Spring Data Workshop
Lab 3

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
- MySQL 8.0
- 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 Designing REST API

In this lab, we will repeat the implementation of a REST API that we had designed in an earlier lab from scratch. This time around we will use a MySQL database table as the underlying persistence storage, rather than just a basic in-memory list.

The first thing we will do again is to create the underlying classes that model the business domain and then determine the REST API endpoints that access and manipulate a collection of objects from these classes.

The REST service that we had previously created was intended to maintain a list of developers with the following schema:

Developer
`id: Integer`
`name: String`
`age: Integer`
`languages: Array[String]`
`married: Boolean`

It will expose the following API endpoints for consumption by a service:

Method	**Endpoint**	**Description**
GET	`/api/developers`	Get the list of developers
GET	`/api/developers/{id}`	Retrieve the developer with the specified id
GET	`/api/developers?language=XXX`	Retrieve the developers with capability in the specified language
GET	`/api/developers?married=XXX`	Retrieve the developers with the specified marital status
GET	`/api/developers?married=XXX&language=YYY`	Combination of the two previous conditions
GET	`/api/developers?age>XX|age<XX`	Retrieve the developers whose age is more or less than XX
GET	`/api/developers?limit=X&page=Y`	Pagination functionality to retrieve page no Y containing a total of X developers
POST	`/api/developers`	Add a new developer to the list of developers
PUT	`/api/developers/{id}`	Make a modification to a developer with the specified id
DELETE	`/api/developers/{id}`	Delete a developer with the specified id

We have left out the PATCH operation at the moment as this requires JSON PATCH or specialized logic to perform correctly.

3 Creating the application package structure

The main folder for this lab is `JPA-Rest-App`

Start up STS. Switch to the Java perspective.

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

Name: `JPARestApp`
Group: `com.workshop.jpa`
Artifact: `JPARestApp`
Version: `0.0.1-SNAPSHOT`
Description: `REST API with Spring Data JPA using MySQL app for developers`
Package: `com.workshop.jpa`

Add the following dependencies:
```
Web -> Spring Web
SQL -> Spring Data JPA
SQL -> MySQL Driver
```

Replace `pom.xml` with the version in `changes` and perform a Maven -> Update Project

In `src/main/java`, create the following packages which are going to have the following purposes:

Package name	Purpose
`com.workshop.jpa.model`	Holds all the classes for the business domain model. This will be stored in the database table via Spring Data JPA, and will thus be @Entity classes.
`com.workshop.jpa.dto`	Holds all the DTO classes that encapsulate the data to be exchanged between the client and service. This includes custom error messages.
`com.workshop.jpa.repository`	Holds the user-defined interfaces that extend on the standard Spring Data JPA Repositories (CrudRepository, PagingAndSortingRepository, etc)
`com.workshop.jpa.service`	Holds the classes that extract data from the repository interfaces and perform any required business logic on them
`com.workshop.jpa.controller`	Holds all the @RestController classes that implement the various @XXXMapping methods for the REST API utilizing the various Service classes.
`com.workshop.jpa.exception`	Holds all the user defined exceptions and the primary @ControllerAdvice exception handling class containing the individual @ExceptionHandler methods

The main @SpringBootApplication class (JpaRestAppApplication) will reside in the top level package: `com.workshop.jpa` so that Spring will scan all its subpackages to locate @Component / @Service / @Repository classes for DI.

4 Creating initial classes and interfaces

In `com.workshop.jpa.model`, place:

`Developer`

In `com.workshop.jpa.dto`, place:

`DeveloperDTO`

In `com.workshop.jpa.repository`, place:

`DeveloperRepository`

In `com.workshop.jpa.service`, place:

`DeveloperService`

In `com.workshop.jpa.controller`, place:

`DeveloperController`

We will use a Data Transfer Object (DTO) to encapsulate the data to be transferred between the client and the REST service we are implementing. The DTO is necessary since the structure of the domain class (marked with @Entity) is slightly different from the structure of the class that we are intend to serialize into JSON to be returned to the client.

In the domain class Developer, the languages field (which represents the languages that the developer is capable of coding in) is stored as a single String where the different languages are separated by commas. This is because MySQL does not easily incorporate the storage of an array as a basic data type. On the other hand, the JSON that is exchanged between the client and service will list these languages as an array as this is a more natural way to represent them.

We will thus use the DTO to represent the structure of the data to be exchanged between the client and service, and then perform conversion between DTO / domain object when storing to or retrieving from the underlying database table.
We have the skeleton now for the main controller class that will provide the logic for mapping all the API endpoints. This will be fleshed out step by step.

5 Initializing the app with SQL tables

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

```
logback-spring.xml
application.properties
data.sql
schema.sql
```

Ensure that you have completed the previous lab on creating a MySQL user account with a password that you will now be using in `application.properties`. The remaining properties set in this file have been already discussed at length in a previous lab.

Start the MySQL server and run the app from the Boot dashboard in the usual way.

Open the MySQL command line client and check that the Developers table has been created and populated with sample data:

```
USE workshopdb;

SHOW TABLES;

SELECT * FROM developers;
```

6 Implement GET /api/developers

In `src/main/resources`, make the change:

`application.properties-v2`

In `com.workshop.jpa.controller`, make the change:

`DeveloperController-v2`

In `com.workshop.jpa.service`, make the change:

`DeveloperService-v2`

In `com.workshop.jpa.dto`, make the change:

`DeveloperDTO-v2`

Notice that the @GetMapping returns a DeveloperDTO object to the client (in order to produce serialized JSON where the `languages` key has a value of an array of strings). However, the objects retrieved from the database are in the Developer class format: so there needs to be a conversion between these two classes. This is accomplished by having a new constructor in the DeveloperDTO class that initializes a new object from a Developer parameter.

DeveloperService in turn uses the existing `findAll` functionality of the repository to retrieve the existing list of developers (of type Developer) from the table, which we have already seen in action in a previous lab. It then creates a list of DeveloperDTOs from this list and returns that to the calling method in DeveloperController.

Restart the app.

Start up Postman and create a collection to group requests for this app: `Developer REST requests`

Make a GET request to:

```
localhost:8080/api/developers
```

and verify that the initial list of developers in Developers table is returned correctly in JSON.

7 Implement POST /api/developers

In `com.workshop.jpa.controller`, make the change:

```
DeveloperController-v3
```

In `com.workshop.jpa.service`, make the change:

```
DeveloperService-v3
```

In `com.workshop.jpa.model`, make the change:

```
Developer-v2
```

In `com.workshop.jpa.repository`, make the change:

```
DeveloperRepository-v2
```

In `com.workshop.jpa.dto`, place the following file:

```
CustomErrorMessage
```

In `com.workshop.jpa.exception`, place the following file:

```
IncorrectJSONFormatException
```

In `com.workshop.jpa.exception`, place the following file:

```
DeveloperControllerExceptionHandler
```

The primary issue in performing a POST correctly is to return the URL to retrieve the newly created resource (which will be a new row in the Developers table). This URL will include the id of the new created row. To obtain the id of this new row (which will automatically be the last row in the table if we use an autoincrement attribute for the primary key id), we create a special native SQL query in DeveloperRepository.

We also create a user defined IncorrectJSONFormatException, a custom class CustomErrorMessage to return to the client and a DeveloperControllerExceptionHandler to house all the future exception handling methods.

Restart the app.

Make a POST request to:

```
localhost:8080/api/developers
```

with the following raw JSON content in the body:

```
{
  "name": "Ryan",
  "age": 42,
  "languages": [
      "php",
      "python",
      "java"
  ],
  "married": true
}
```

Check that a status 201 Created is returned with the following URL in the Location header:

http://localhost:8080/api/developers/xxx
where xxx is the id of the newly created record in the Developers table.

You can verify this in the MySQL command line client.

You can also verify it by making another GET request to retrieve all the developers:

```
localhost:8080/api/developers
```

Introduce an error into the POST submission by leaving out an important field (either name, age or languages). For e.g. sent a POST to the same URL with this content:

```
{
  "name": "Aaron",
  "age": 42,
  "married": true
}
```

Verify that the correct exception handling method is invoked server side and an appropriate error message is returned.

Make another GET request to the same URL (or check through the MySQL command line client) to verify that this information was not added as a new developer.

```
localhost:8080/api/developers
```

Introduce an error by submitting invalid JSON using a POST for e.g.

```
{
  "name": "Debbie"
  "age": 29
  "languages": [
      "JavaScript",
      "Python",
      "Java"
  ]
  "married": false
}
```

Verify that the correct exception handling method is invoked server side and an appropriate error message is returned.

Make another GET request to the same URL (or check through the MySQL command line client) to verify that this information was not added as a new developer.

`localhost:8080/api/developers`

8 Implement PUT /api/developers/{id}

In `com.workshop.jpa.repository`, make the change:

`DeveloperRepository-v3`

In `com.workshop.jpa.controller`, make the change:

`DeveloperController-v4`

In `com.workshop.jpa.exception`, make the change:

`DeveloperControllerExceptionHandler-v2`

In `com.workshop.jpa.service`, make the change:

`DeveloperService-v4`

In `com.workshop.jpa.exception`, create the files:

`IncorrectURLFormatException`

In DeveloperRepository, we create a new native SQL update statement and this will need to be annotated with @Modifying and @Transactional for it to work properly.

DeveloperController and DeveloperService include additional logic to check for errors in the request and will throw appropriate exceptions if this occurs. We have created a new Exception class: `IncorrectURLFormatException` to cater for URL-related errors such as invalid id type or id out of range. This is catered for appropriately in `DeveloperControllerExceptionHandler`

Note: Technically speaking, specifying a developer id that does not exist in the table should be a case of `DeveloperNotFoundException`, rather than a `IncorrectURLFormatException`. However, we use `IncorrectURLFormatException` since the semantics of this exception covers a wide range of other URL-related error issues as well which minimizes the number of user-defined exceptions we might need to create.

Restart the app.

Make a PUT request to:

```
localhost:8080/api/developers/3
```

with the following raw JSON content in the body:

```
{
  "name": "Edwin",
  "age": 21,
  "languages": [
      "python",
      "c++"
  ],
  "married": false
}
```

Verify that this returns with status 200 OK.

To verify that the developer with id 3 has had the details changed accordingly, make a GET request to:

```
localhost:8080/api/developers
```

Alternatively, verify from the MySQL command line client.

You can repeat the operation several times to change the details of a few other random developers.

Let's introduce some potential errors in the request.

Make a PUT request to:

```
localhost:8080/api/developers/888
```

using some valid JSON content. Verify that the error message returned points out that no developer with such id exists.

Make a PUT request to:

```
localhost:8080/api/developers/3sw
```

Verify that the error message returned points out that the developer id needs to be specified correctly as a number.

Make a PUT request to:

`localhost:8080/api/developers/3`

with the following raw JSON content in the body:

```
{
  "age": 21,
  "languages": [
      "Python",
      "C++"
  ],
  "married": false
}
```

Verify that the correct error message is returned regarding incorrect format for developer

9 Implement DELETE /api/developers/{id}

In `com.workshop.jpa.controller`, make the change:

`DeveloperController-v5`

In `com.workshop.jpa.service`, make the change:

`DeveloperService-v5`

The implementation here is reasonably straight forward since there is already an existing `deleteById` method from CrudRepository that we can call directly from DeveloperService.

Restart the app.

Make a DELETE request to:

`localhost:8080/api/developers/3`

Verify that this returns with status 200 OK.

To verify that the developer with id 3 has had the details changed accordingly, make a GET request to:

`localhost:8080/api/developers`

Alternatively, verify from the MySQL command line client.

You can repeat this operation to delete a few other random records and verify accordingly.

We can check for similar errors as in the case of the PUT operation.

Make a DELETE request to:

```
localhost:8080/api/developers/888
```

Verify that the error message returned points out that no developer with such id exists.

Make a DELETE request to:

```
localhost:8080/api/developers/3sw
```

Verify that the error message returned points out that the developer id needs to be specified correctly as a number.

If you have deleted too many rows from the table, you can always restore the original table by changing the settings in `application.properties` as shown in step 5 previously and running the application to load from the *.sql tables on the classpath. Then change it back again as show in step 6.

10 Implementing GET /api/developers/{id}

In `com.workshop.jpa.controller`, make the change:

```
DeveloperController-v6
```

In `com.workshop.jpa.service`, make the change:

```
DeveloperService-v6
```

The implementation here is reasonably straight forward since there is already an existing `findById` method from CrudRepository that we can call directly from DeveloperService.

Restart the app.

Make a GET request to:

```
localhost:8080/api/developers/x
```

where x is valid developer ID. Verify that the correct developer is returned by inspecting from MySQL command line client if necessary.

We can check for similar errors as in the case of the PUT operation.

Make a DELETE request to:

```
localhost:8080/api/developers/888
```

Verify that the error message returned points out that no developer with such id exists.

Make a DELETE request to:

```
localhost:8080/api/developers/3sw
```

Verify that the error message returned points out that the developer id needs to be specified correctly as a number.

11 Implementing GET /api/developers with query parameters

In `com.workshop.jpa.controller`, make the change:

```
DeveloperController-v7
```

In `com.workshop.jpa.service`, make the change:

```
DeveloperService-v7
```

Since there are many query parameters that can be passed via a GET to `/api/developers`, we will refactor the existing code base to streamline our implementation by having the DeveloperService determine the underlying operation to perform on the DeveloperRepository based on the query parameters being passed.

We initially make distinction between a GET request without any query parameters (which should return the list of all developers) and a request with some query parameters (which we initially list and will subsequently process one at a time).

Restart the app.

Make a GET request to:

```
localhost:8080/api/developers
```

and verify that the entire contents of the Developers table is returned.

Make a GET request to the same URL but with some random query parameters, for e.g:

```
localhost:8080/api/developers?hero=ironman&age=33
```

and verify that nothing is returned now (even though status remains as 200 OK) and the key value pairs are logged on the server side

11.1 Query parameter: language=XXX

In `com.workshop.jpa.service`, make the change:

```
DeveloperService-v8
```

In `com.workshop.jpa.repository`, make the change:

`DeveloperRepository-v4`

We declare the derived query method `findByLanguagesContaining` in DeveloperRepository to perform a filtering on the language `column`, and provide appropriate logic in DeveloperService to call this method when the appropriate query parameter is present.
An issue that we need to watch out for is for the language values: C++ and C#. The symbols + and # are reserved characters, so trying to incorporate them directly into a URL, for e.g:

localhost:8080/api/developers?language=c++

will result in an error in retrieving them on the server side. In order to retrieve them correctly, they would need to be encoded first using the standard URL encoding scheme:
https://www.url-encode-decode.com/

Thus, for the parameter
`language=c++`
we would need to specify it as:
`language=c%2B%2B`

and for the parameter
`language=c#`
we would need to specify it as:
`language=c%23`

The other alternative is to require the client to just use alphanumeric characters for specifying parameter values e.g.
`language=cplusplus`
or
`language=sharp`
and then on the server side (in DeveloperService match to this to the actual corresponding language values of C++ or C#).

To keep things simple here, we will just use the first approach of requiring the client to specify the values using the correct URL encoding although the second approach is probably preferable in a production grade application.

Restart the app.

Make multiple GET requests to:

`localhost:8080/api/developers ?language=XXXX`

where `XXX` can be any of the valid languages associated with the existing developers: e.g. python, java, c++, etc. Keep in mind the issue about c++ and c# and URL encoding as explained earlier. Try entering a random string for `XXX` to verify that an empty list is returned.

Verify that the correct subset of developers is returned with a status 200 OK and double checking at the MySQL command line if necessary.

Also verify that making a GET request to:

```
localhost:8080/api/developers
```

without any query parameters should still return the entire contents of the Developers table.

11.2 Query parameter: married=XXX

In `com.workshop.jpa.service`, make the change:

```
DeveloperService-v9
```

In `com.workshop.jpa.repository`, make the change:

```
DeveloperRepository-v5
```

Again, we declare the derived query method `findByMarried` in DeveloperRepository to perform a filtering on the column `married`, and provide appropriate logic in DeveloperService to call this method when the appropriate query parameter is present. We also return an error message to the client if a value other than `true` or `false` is supplied for this parameter.

Restart the app.

Make GET requests to:

```
localhost:8080/api/developers?married=false
```

and

```
localhost:8080/api/developers?married=true
```

Verify that the correct subset of developers is returned with a status 200 OK.

Make a GET request with an invalid value for the `married` parameter, for e.g.

```
localhost:8080/api/developers?married=sdf
```

Verify that an appropriate error message is returned with a status 400 BAD REQUEST.

Also verify that making a GET request to:

```
localhost:8080/api/developers
```

without any query parameters should still return the entire contents of the Developers table.

11.3 Combine query parameters: language=XXX&marriage=YYY

In `com.workshop.jpa.service`, make the change:

`DeveloperService-v10`

In `com.workshop.jpa.repository`, make the change:

`DeveloperRepository-v6`

Again, we declare the derived query method `findByMarriedAndLanguagesContaining` in DeveloperRepository to perform a filtering on the columns `languages` and `married`, and provide appropriate logic in DeveloperService to call this method when both query parameters are present.

Restart the app.
Make a GET request to:

`localhost:8080/api/developers?married=true&language=java`

Verify that the correct subset of developers is returned with a status 200 OK.

Notice that changing the order of appearance of the parameters does not affect the result:

`localhost:8080/api/developers?language=java&married=true`

Experiment around with a few other combinations for the `married` and `language` parameters and verify that the correct subset of developers is returned with a status 200 OK.

11.4 Query parameter: age>XX or age<XX

In `com.workshop.jpa.service`, make the change:

`DeveloperService-v11`

In `com.workshop.jpa.repository`, make the change:

`DeveloperRepository-v7`

Note that the format: `age>XX` or `age<XX` is counted as a key without a value as the = sign is used to separate keys and values.
Again, we declare the derived query method `findByAgeLessThan` and `findByAgeGreaterThan` in DeveloperRepository to perform a filtering on the `age` column, and provide appropriate logic in DeveloperService to call either of these methods depending on whether the symbol > or < is used. We also incorporate logic to check the expression for non-integer or negative values and return a suitable error message to the client if this occurs.

Restart the app.
Make a GET request to:

`localhost:8080/api/developers?age<30`

Verify that the correct subset of developers is returned with a status 200 OK.

Make a GET request to:

```
localhost:8080/api/developers?age>50
```

Verify that the correct subset of developers is returned with a status 200 OK.

Make a GET request to the following incorrect URLs:

```
localhost:8080/api/developers?age>-200
localhost:8080/api/developers?age<-1
localhost:8080/api/developers?age>asdf
localhost:8080/api/developers?age=kewlstuff
```

and verify that an error message is returned.

11.5 Query parameters: limit=X&page=Y

In `com.workshop.jpa.service`, make the change:

```
DeveloperService-v12
```

For pagination, we can simply use the existing `findAll(pageable)` method available from PagingAndSortingRepository demonstrated in a previous lab. We just need to incorporate logic to extract the values for the `limit` and `page` parameters correctly, as well as logic to ensure that both of them are supplied at the same time and have positive integer values. If this is not the case, we return a suitable error message to the client.

Restart the app.

Make a GET request to:

```
localhost:8080/api/developers?page=1&limit=5
```

Verify that the correct subset of developers is returned with a status 200 OK.

Make a GET request to:

```
localhost:8080/api/developers?page=3&limit=3
```

Verify that the correct subset of developers is returned with a status 200 OK.

Make a GET request to:

```
localhost:8080/api/developers?page=4&limit=5
```

Verify that the correct subset of developers is returned with a status 200 OK.

Make a GET request to:

```
localhost:8080/api/developers?page=5&limit=10
```

Notice that even though the page and limit specified exceeds the total remaining number of developers in the list, no error is flagged and an empty list is returned instead.

Errors will be flagged if:

- Non-numeric or negative values are specified for either the `page` or `limit` parameters
- The `page` or `limit` parameter is supplied individually without the other (both are required for the pagination functionality to work)

Make GET requests to these URLs to verify that all of these errors are flagged with appropriate error message responses:

```
localhost:8080/api/developers?page=xxx&limit=10

localhost:8080/api/developers?page=2&limit=yyy

localhost:8080/api/developers?page=-1&limit=5

localhost:8080/api/developers?page=2

localhost:8080/api/developers?limit=6
```

At this point, it is also good to verify that making a GET request to:

```
localhost:8080/api/developers
```

without any query parameters should still return the entire contents of the Developers table.

11.6 Further filtering / sorting functionality

Query parameters used with GET requests are ideal for performing a variety of filtering and sorting functionality. For e.g. we could sort the list of developers returned on ascending or descending value of a particular field (such as age or name). The query parameter might look like this:

```
GET /api/developers?sort=-age
```

which might mean to sort on descending order of the age field of the developers.

```
GET /api/developers?sort=+name
```

which might mean to sort on ascending order of the name field of the developers.

```
GET /api/developers?sort=-age,+name
```

which might mean to sort on descending order of the age field of the developers first (primary sort), and then sort on ascending order of the name field for the case of developers who have the same age (secondary sort).

See whether you can implement this sorting functionality using either native SQL queries or the sorting functionality provided by PagingAndSortingRepository: both of which have already been demonstrated in a previous lab.