DMHP : USER GUIDE

Mani Nandadeep
IMT2019051

R Prasannavenkatesh
IMT2019063

Vijay Jaisankar
IMT2019525

M Dhanush
IMT2019049

CONTENTS

1 ADDING A NEW CHART : ANGULAR

The current version of the project uses Apex Charts for graphing procedures. The graphing procedure has the following workflow:

1.1 Call the API

First, we call the required API using the corresponding body parameters. For example, to call */getTraining*, we send the required arguments in a JSON format as the body parameters which in turn accesses the corresponding Node.js route.
The API is called via a method present in the *backend-connector service* which returns a rxjs observable.

1.2 Subscribe to the API observable

The API call returns an *observable* which is to be *subscribed* further. In the subscription body the data is returned and the corresponding logic can be registered and executed. The API observable can then be operated in your required *component*.

1.3 Create a map

Using the data from the API, create an ES6 map with your custom key-value pairs.

You can either create the map in the *subscription body* of your API call itself or you can outsource the logic to one of the *services* which in turn takes in the API response and returns the corresponding map.

1.4 Creating Apex charts options

Once we have created the map, we create an *options* variable which contains the necessary details to create the graph.

The chart options creation can be abstracted to a service or we can create that in the component itself.

1.5 Rendering Apex Charts

Once the options are created use the render method of apex charts to render the graph in the required position. Much like the other steps in this process, the chart rendering can again be abstracted to a service or we can create that in the component itself.

2 ADDING A NEW ROUTE : NODE.JS

To make a new route in *node.js*, here's what we have to do.

2.1 Prerequisites

Before adding a new route, the following information should be ready:

- The HTTP Method of the new request (i.e GET/POST/PUT/PATCH/DELETE/etc)
- The JSON format of the input (if any). In particular, the variable names of the keys in the input parameters should be known.
- Whether the new route needs JWT authentication for access
- If the new request calls a stored procedure, the stored procedure should be added to the database and the argument list and order should be known.
- A Validation function for the input parameters (if any)

2.2 Adding the route

If the route does not need authentication, add the route to the *excludedRoutes* array in *app.js*.

The syntax for adding a new route is expressObject.HTTPMETHOD(ROUTE, function_for_processing)

For example, app.get("/newroute", (req, res) => res.json("HelloWorld"));
specifies a *GET* request to the *newroute* endpoint, which returns "Hello World".

2.3 Processing input JSON

If the input JSON is of the form "key" : "value", key can be accessed by req.body.key, if req is the request object variable.

One can then process the input parameters.

2.4 Validation

If we are going to add a Validation function to the input parameters, that must be abstracted away in validators.js.

The syntax for the same is :
referenceName : functionfunctionName(< parameters >)...

Note that this will be one of the exports in the file, this can then be called in app.js as follows: referenceName(< parameters >)

2.5 Testing the Route

We can use an service like Postman, cURL, or Insomnia to call the route. Do note that, if JWT is used, a *Bearer Token* must be added; this token can be found by calling the /api/auth route.

Please add the successful input JSON as an example call in the /backend/sample − json − calls directory.

3 ADDING A NEW ROUTE : SPRING BOOT

To make a new route in Spring Boot, here's what we have to do.

3.1 Prerequisites

Before adding a new route, the following information should be ready.

- The HTTP Method of the new request (i.e GET/POST/PUT/PATCH/DELETE/etc)
- The JSON format of the input (if any). In particular, the variable names of the keys in the input parameters should be known.
- The column names in the table, and the corresponding flags that will determine decorators for the variables representing column names.
- Methods that fall outside the bounds of JPARepository methods need to be present beforehand.
- Custom Error handling classes, if any, should be created beforehand.

3.2 Entity

In the *entity* package, there are classes corresponding tables. In these classes, variables corresponding to column names, along with decorators that elucidate Hibernate validators are also present (for example, *@NotNull* and *@Id*).

To add a route that needs representation of a new table, a new file should be added in this package, whose name should be the same as that of the table. Note that, if we want to keep the same column nomenclature format as that of MySQL, the following lines should be present in the `application.properties` file :

`spring.jpa.hibernate.naming.implicit-strategy =`
`org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl`

`spring.jpa.hibernate.naming.physical-strategy =`
`org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl`

3.3 Controller

In the *controller* package, we have an enumeration of API routes and permissions for tables.

To add a new route, we must add a new controller and bind it to the new Entity class created.

We can also add *Controller-level Logic*, by allowing access to only certain callers; this is encapsulated in the `@CrossOrigin()` decorator.

We then call the corresponding method of the service layer through *dependency injection*.

For example, a POST request to the `/newroute/` endpoint which allows only *localhost:4200* to access it, which calls SERVICEMETHOD is given by:

```
@PostMapping("/newroute")
@CrossOrigin(origins = "http://localhost:4200")
public returnType functionName(<parameters>) {
// Processing
return boundServiceObject.SERVICEMETHOD(<parameters>);
}
```

3.4 Service

The *service* package consists of pairs of Interface-Implementing classes for tables. Every method outlined in the Controller class should be implemented using this paradigm.

Service Layer methods involve *Service-level logic*, and Calling the Repository Layer methods for the same.

For example, a function in the Service Layer that facilitates adding a new entry to the Table is given by:

```
@Override
public entity_class_name SERVICEMETHOD(entity_class_name var) {
return boundRepositoryObject.save(var);
}
```

The `@Override` stems from the fact that this method is in the implementation class that implements the interface containing `SERVICEMETHOD`.

3.5 Repository

The *repository* package consists of a list of Interfaces that extend `JpaRepository`. We can either use these methods out of the box, or by creating our own. Note that, this could entail making new interfaces created for this purpose.

The syntax for the repository is
`RepositoryClassName < Entity_Class_Name, DataTypeOfPrimaryKeyOfCorrespondingEntityClass`

3.6 Tests

We can add unit tests in the `demo/src/test/TESTINGLAYERPACKAGE`.

We must take care of two variables here:

- Mocking the parent Layer
- Assertions of the test

3.7 Testing the Route

We can use an service like Postman, cURL, or Insomnia to call the route.