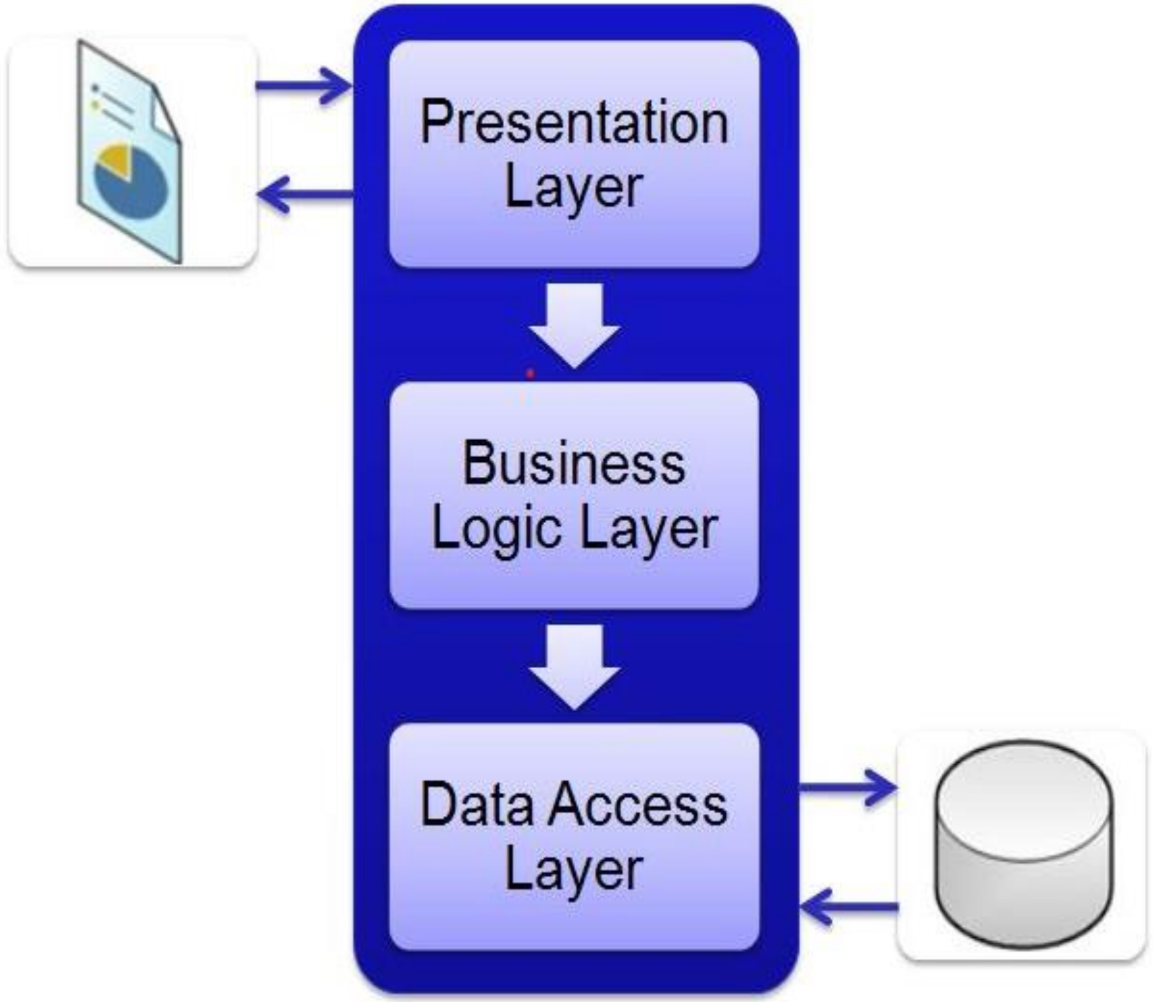
BAAS 360 - REST BUSINESS LAYER

Created Date : 10/26/2023
Created By: GOVE ACADEMY

Table of Content

What is Business Layer?

- Business layer is used to write all the business logics.
- Business layer is also referred to as "Service layer" or "Logic layer".
- This is a software architectural layer that implements the core functionality of an application, encapsulating the business rules and domain logic.
- It depends on the features of the business



Rest Business Folder Structure

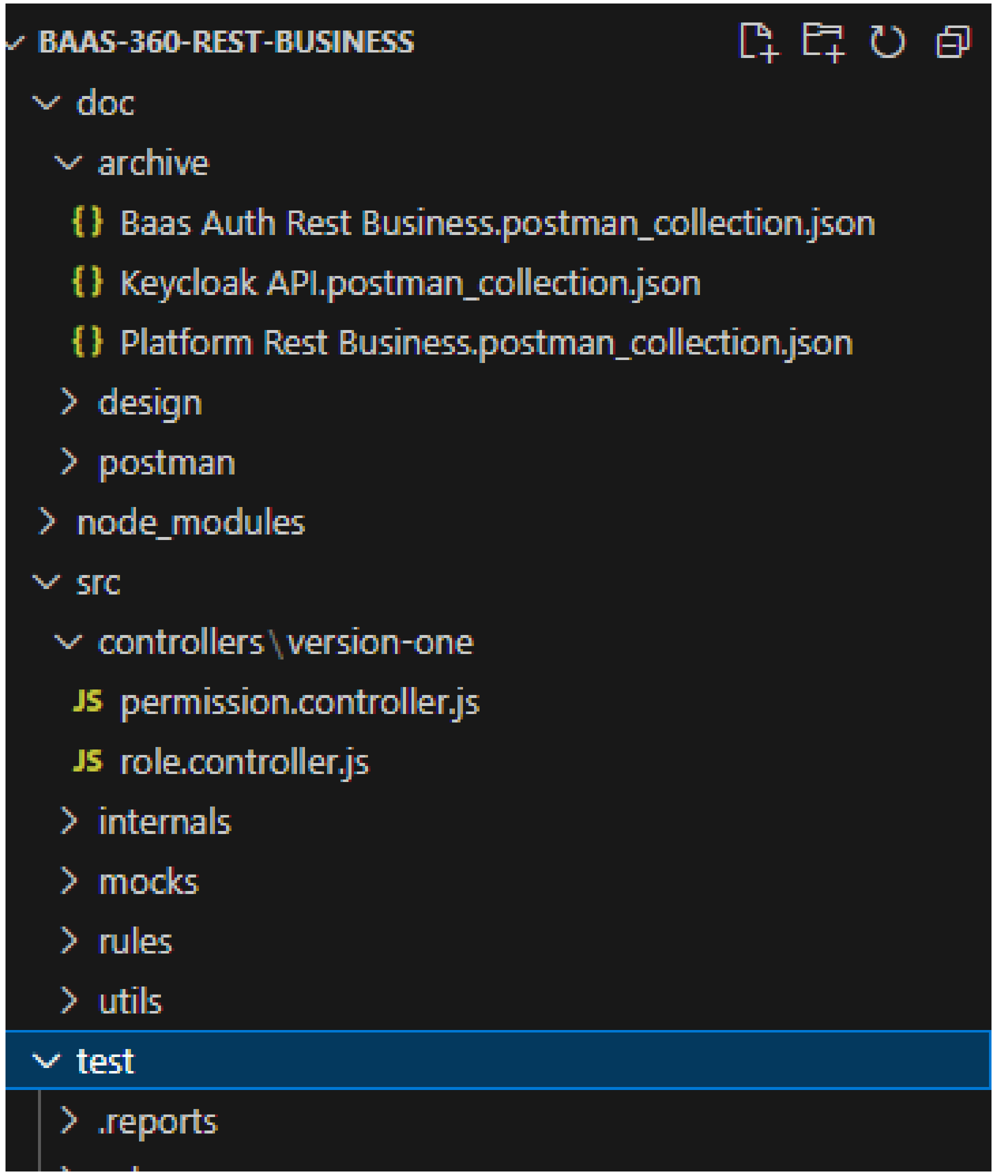
3 MAIN FOLDERS

- **Doc**: Document folder is used to maintain a details, in this there are 3 in built folders are there which is used store Postman collections and Keycloak API'S.

- **Src:** Source folder is used to hold the primary source files for the project, the significance of the source folder vary depending on the context of the framework in use. The extension for the files is **".js"**

- **Test:** Test folder is used to hold unit test reports for every component in html format. The structure is like src folder but the extension is **".test.js"**.

Structure of above mentioned Folders



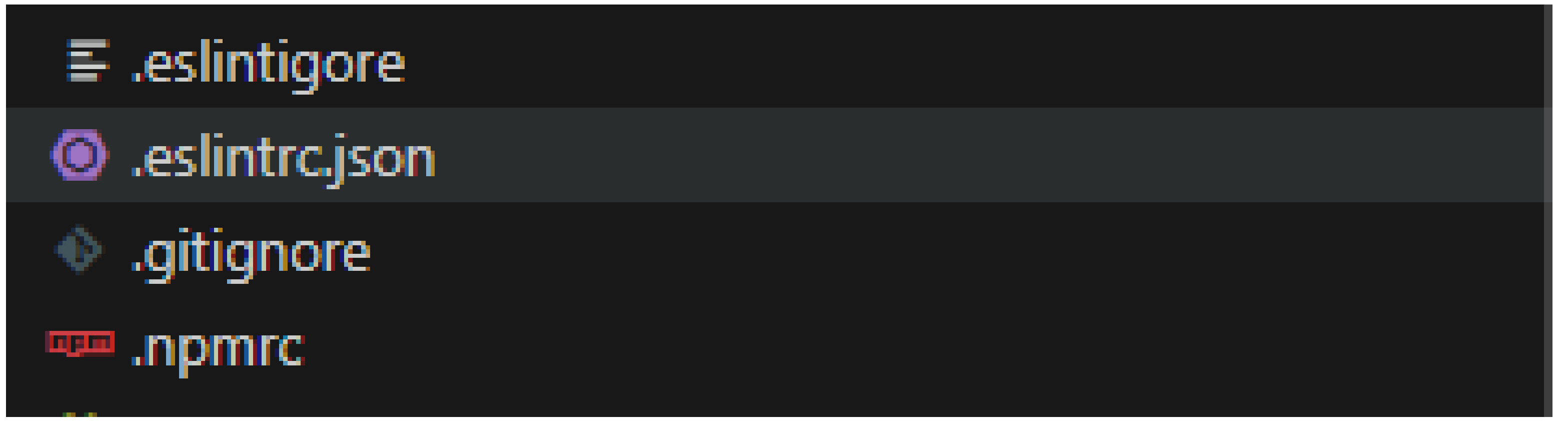
```
BAAS-360-REST-BUSINESS
  doc
    archive
      Baas Auth Rest Business.postman_collection.json
      Keycloak API.postman_collection.json
      Platform Rest Business.postman_collection.json
    design
    postman
  node_modules
  src
    controllers \ version-one
      permission.controller.js
      role.controller.js
    internals
    mocks
    rules
    utils
  test
    .reports
```

Folder Structure

.eslintignore: Used to ignore the files , the lint rules will not be applied to the files which are mentioned in this file.

.eslintrc.json: It is static code analysis of JavaScript, used for software verification analysis and have automated code reviews.It is used to control the configuration file and recommends to follow the structure and also used to add our custom structure

.npmrc: It hosts all packages. Used to setup private registry in project.



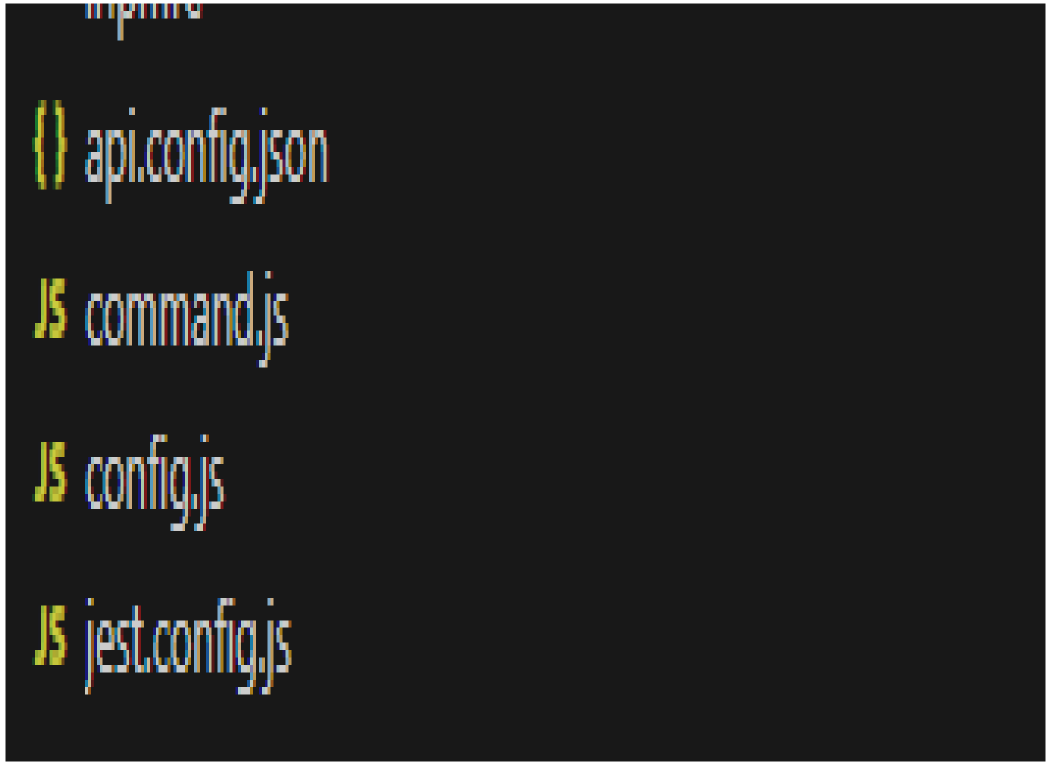
```
.eslintigore
.eslintrc.json
.gitignore
.npmrc
```

Folder Structure

- **Api.config.json:** This folder contains all configuration regarding the project. Used to declare the URL for the API, Paths for various endpoints and versioning information.

- **config.js**: Used to control the configuration of remote services. Like it gives
  - port number in which the service should run
  - Basic authentication
  - Remote configuration
  - Connector configuration.

- **jest.config.js:** is a configuration file used by Jest, a popular JavaScript testing framework. This file allows developers to customize and configure how Jest behaves in their project.



Main.js: This folder serves as the primary entry point for the application. This means that when the application starts, **main.js** is the first file that gets executed or imported. From here, other modules or dependencies might be loaded.

Package-lock.json: It is an automatically generated file for Node.js projects that use npm (Node Package Manager) as a package management tool. This file plays a critical role in ensuring consistent installation of dependencies.

Package.json: The file contains metadata about the project and various configuration settings.

Server.js: It is where the server application is typically initialized. This includes setting up middleware, defining routes, initializing database connections, importing controller classes and injecting the endpoints to the respective controller classes.

Creation of Folders and files in each stage

Controller Folder: We have to create a controller folder in source folder in which the controller object is created. In this we have to write all the API's endpoint to function mapping for modules.

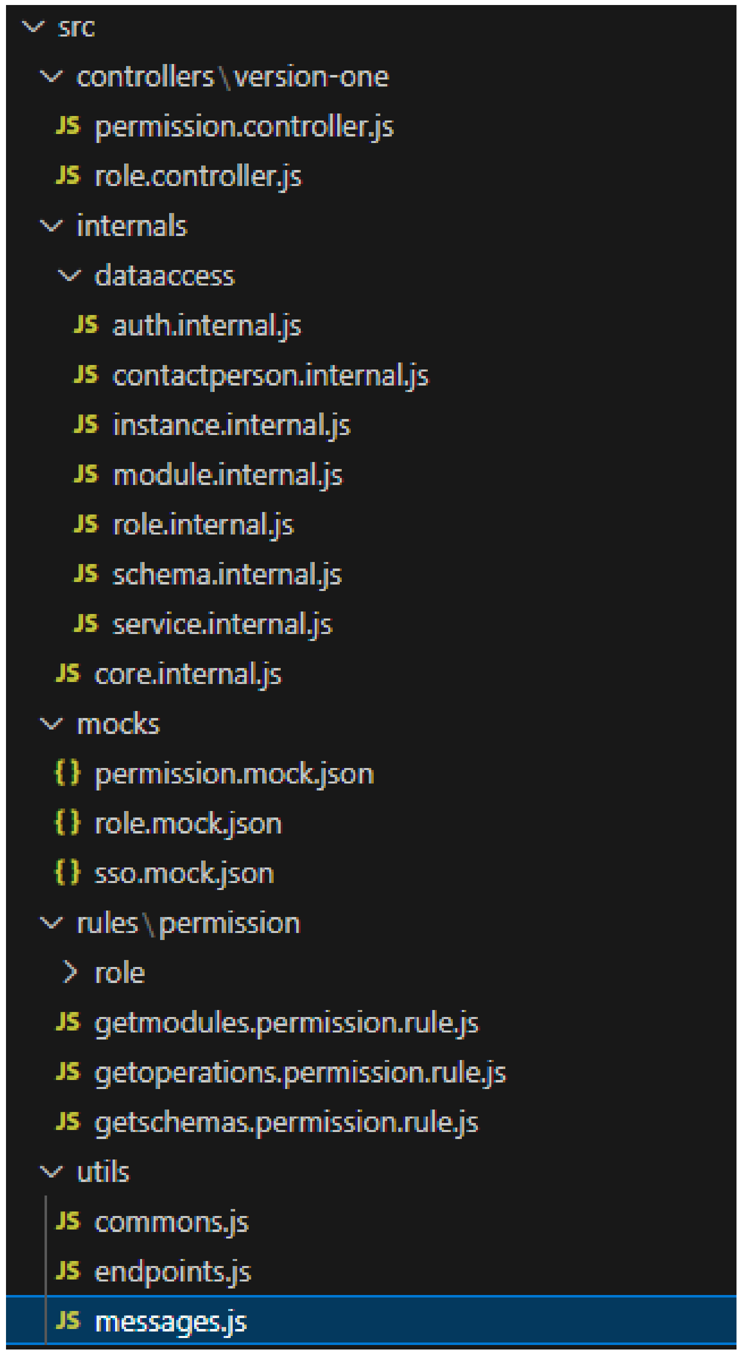
Internals folder: This is main folder, in this folder we have to create dataaccess folder.

Dataaccess folder: For one module in dataaccess layer one folder should be created. It is used to give detail explanation about each operation which we are going to do on tables.

Core-internal: This file is the main base for all the files in the dataaccess folder. First, we have to create core-internal class, in that we are using Axios to call dataaccess layer API. In all the files we are using the core-internal object.

Creation of Files

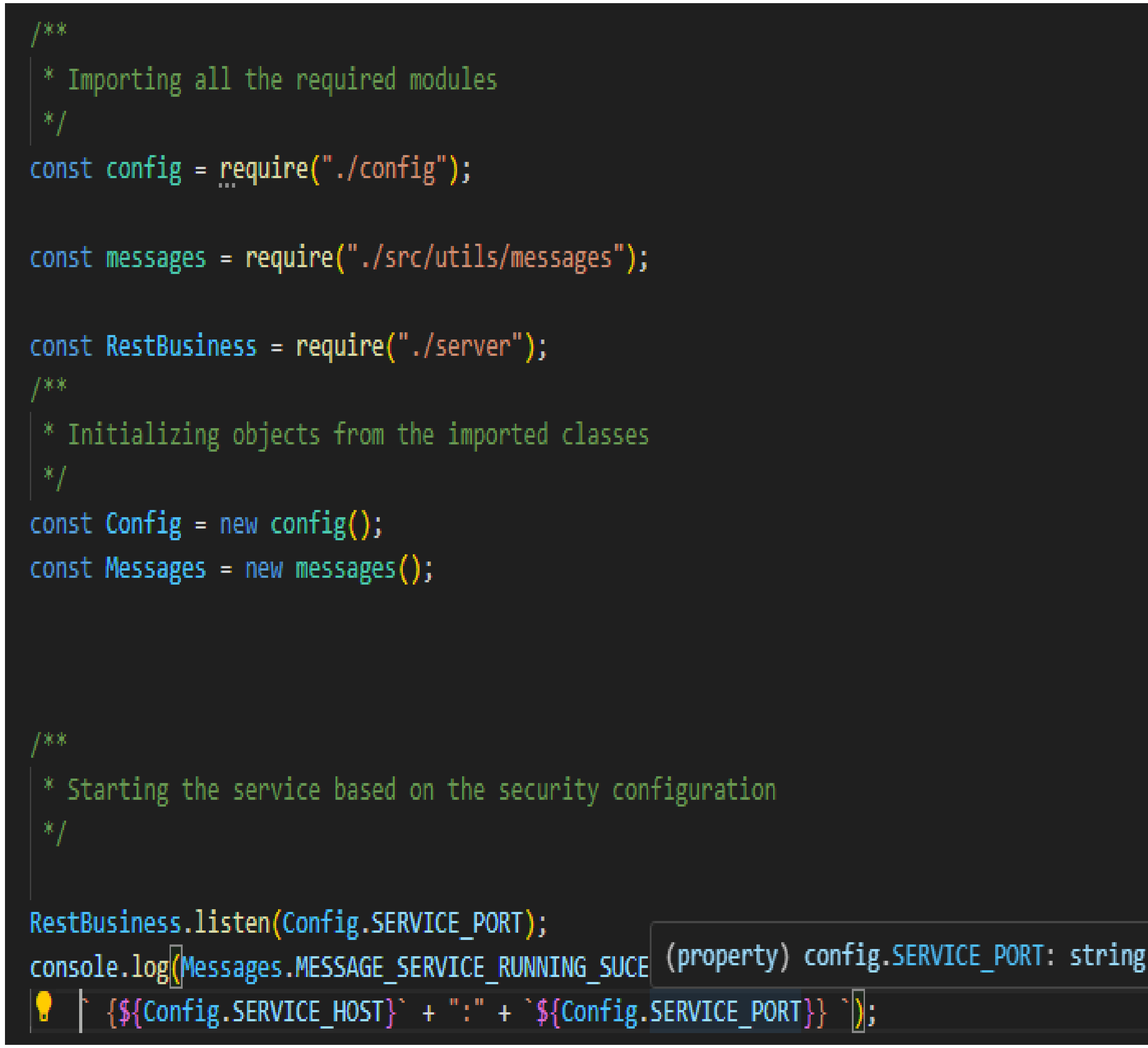
- **Mock folder**: "mock" is conventionally used to store mock implementations and configurations. In this folder we have to give correct path, data, status and message.
- **Rules**: We have to create folder in rules folder, in that we are creating some files, in that files we have to write business logic. When this particular function calls that logic will execute.
- **Utils**: In this there are 3 files are there,
  - **Commons.js**: file contains all common functions related to project.
  - **Endpoints.js**: contains all the entry points like auth rest services, modules, versions and functionalities.
  - **Messages.js**: it contains all messages for applications.



```
v src
  v controllers \ version-one
    JS permission.controller.js
    JS role.controller.js
  v internals
    v dataaccess
      JS auth.internal.js
      JS contactperson.internal.js
      JS instance.internal.js
      JS module.internal.js
      JS role.internal.js
      JS schema.internal.js
      JS service.internal.js
    JS core.internal.js
  v mocks
    {} permission.mock.json
    {} role.mock.json
    {} sso.mock.json
  v rules \ permission
    > role
    JS getmodules.permission.rule.js
    JS getoperations.permission.rule.js
    JS getschemas.permission.rule.js
  v utils
    JS commons.js
    JS endpoints.js
    JS messages.js
```

Code Walkthrough

- First, start reading from the ***main.js*** - This is the entry point for all the services.
  - In main.js file we are importing some configurations which is needed for the modules.
  - Creation of controller object.
- After that it redirects to ***server.js*** file, there also we are importing required modules and object creation.
  - Adding middle wares based on authentication.
  - Importing all the control classes.
  - Injecting the endpoints regarding all the control classes.



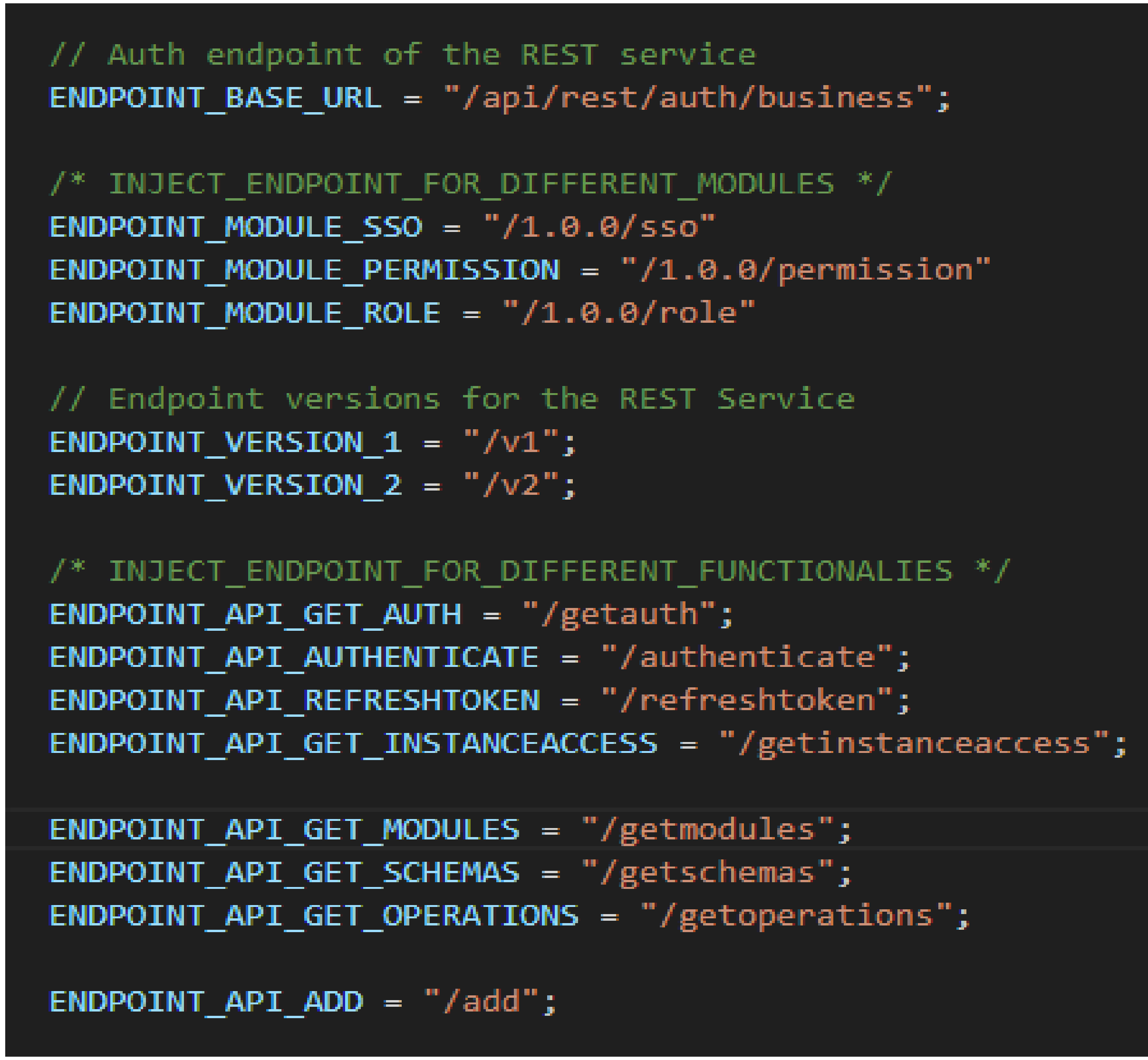
```
/**
 * Importing all the required modules
 */
const config = require("./config");

const messages = require("./src/utils/messages");

const RestBusiness = require("./server");
/**
 * Initializing objects from the imported classes
 */
const Config = new config();
const Messages = new messages();



/**
 * Starting the service based on the security configuration
 */

RestBusiness.listen(Config.SERVICE_PORT);
console.log(Messages.MESSAGE_SERVICE_RUNNING_SUCE (property) config.SERVICE_PORT: string
  ` {${Config.SERVICE_HOST}` + ":" + `${Config.SERVICE_PORT}} `);
```

Controllers, endpoints and rules

- ***Controllers*** file have all the API endpoint to function mapping for all modules.
  - Mainly it has controller object and write the all API's .
  - Create a file with ***.controller.js*** extension.

- Go to utils folder, there is file called ***endpoints.js*** in this file we are giving paths for URL like:
  - Auth endpoint
  - Modules endpoints
  - Versions
  - Different functionalities.

- Then go for the ***rules folder,*** create one more folder there , in that create a file with ***..rule.js extension***, that file contains all the business rules for the modules.



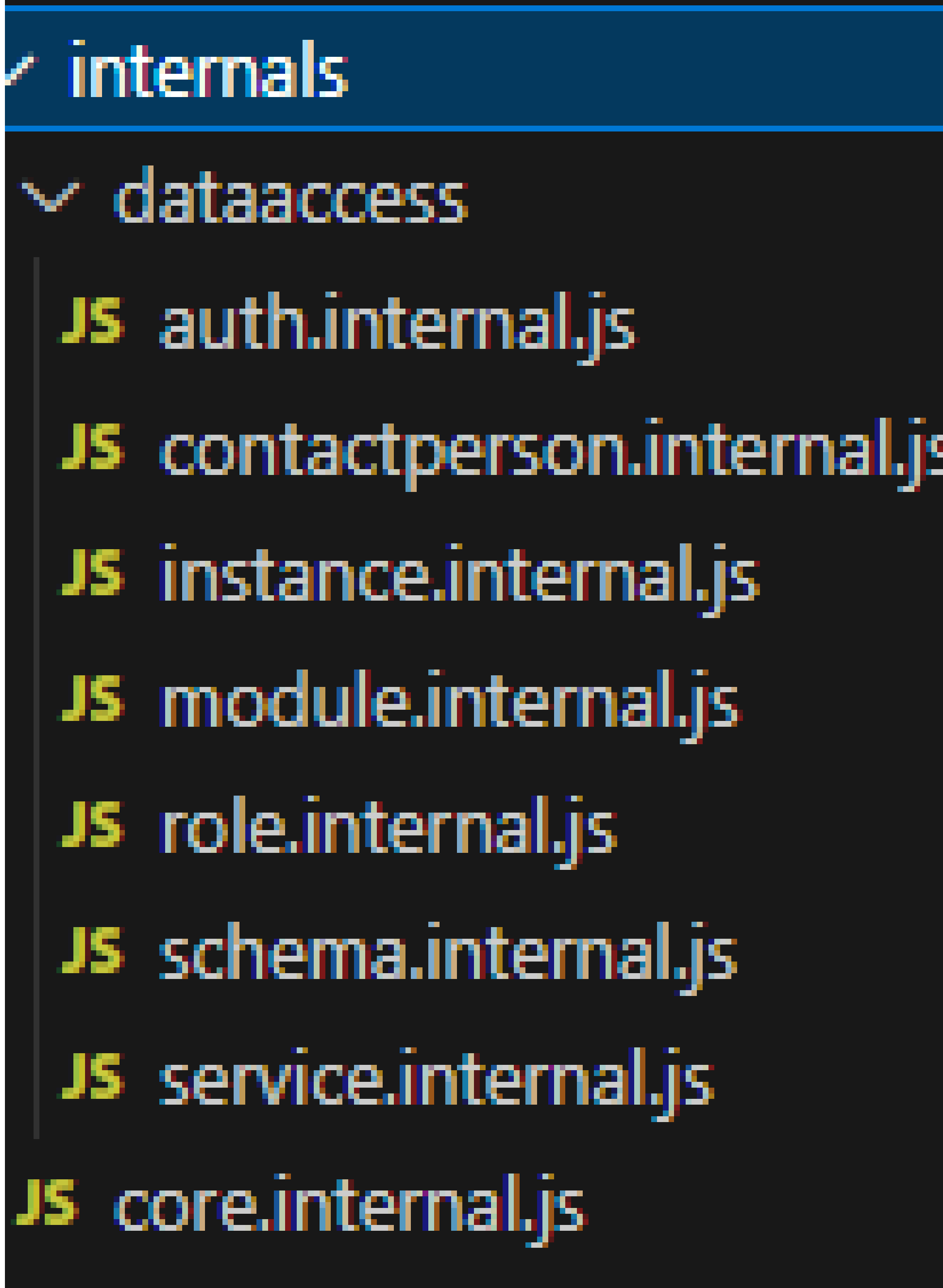
```
// Auth endpoint of the REST service
ENDPOINT_BASE_URL = "/api/rest/auth/business";

/* INJECT_ENDPOINT_FOR_DIFFERENT_MODULES */
ENDPOINT_MODULE_SSO = "/1.0.0/sso"
ENDPOINT_MODULE_PERMISSION = "/1.0.0/permission"
ENDPOINT_MODULE_ROLE = "/1.0.0/role"

// Endpoint versions for the REST Service
ENDPOINT_VERSION_1 = "/v1";
ENDPOINT_VERSION_2 = "/v2";

/* INJECT_ENDPOINT_FOR_DIFFERENT_FUNCTIONALIES */
ENDPOINT_API_GET_AUTH = "/getauth";
ENDPOINT_API_AUTHENTICATE = "/authenticate";
ENDPOINT_API_REFRESHTOKEN = "/refreshtoken";
ENDPOINT_API_GET_INSTANCEACCESS = "/getinstanceaccess";

ENDPOINT_API_GET_MODULES = "/getmodules";
ENDPOINT_API_GET_SCHEMAS = "/getschemas";
ENDPOINT_API_GET_OPERATIONS = "/getoperations";

ENDPOINT_API_ADD = "/add";
```

Internals and core.internal

This file redirects to ***internals folder***, in that data access folder is there, in that folder create a file with ***.internal.js*** extension.

This file contains all the information related to tables.

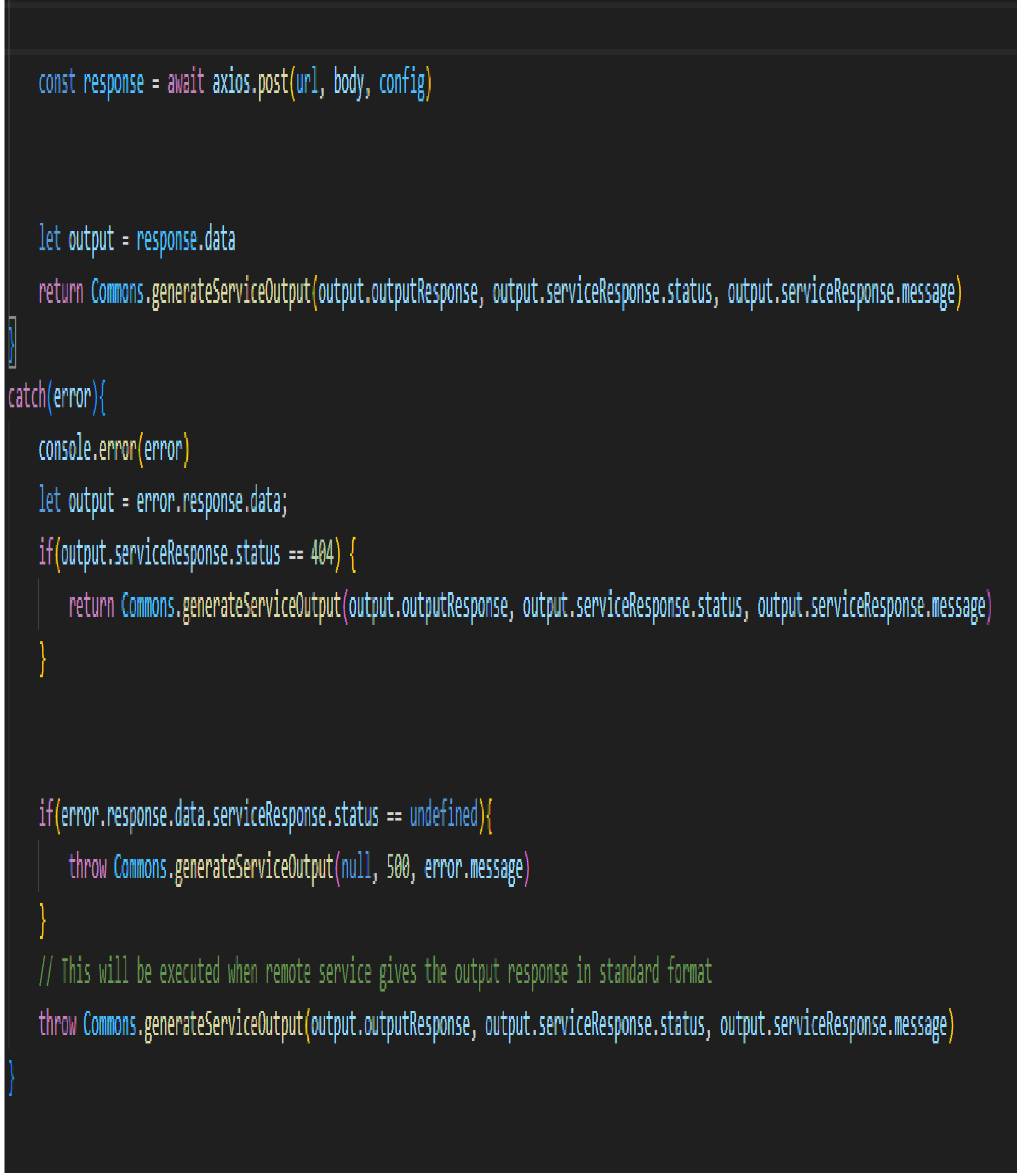
In this one common file is there ***core.internal.js***, which is base for all the internal files. We are using Axios .



```
internals
  dataaccess
    auth.internal.js
    contactperson.internal.js
    instance.internal.js
    module.internal.js
    role.internal.js
    schema.internal.js
    service.internal.js
  core.internal.js
```

Axios

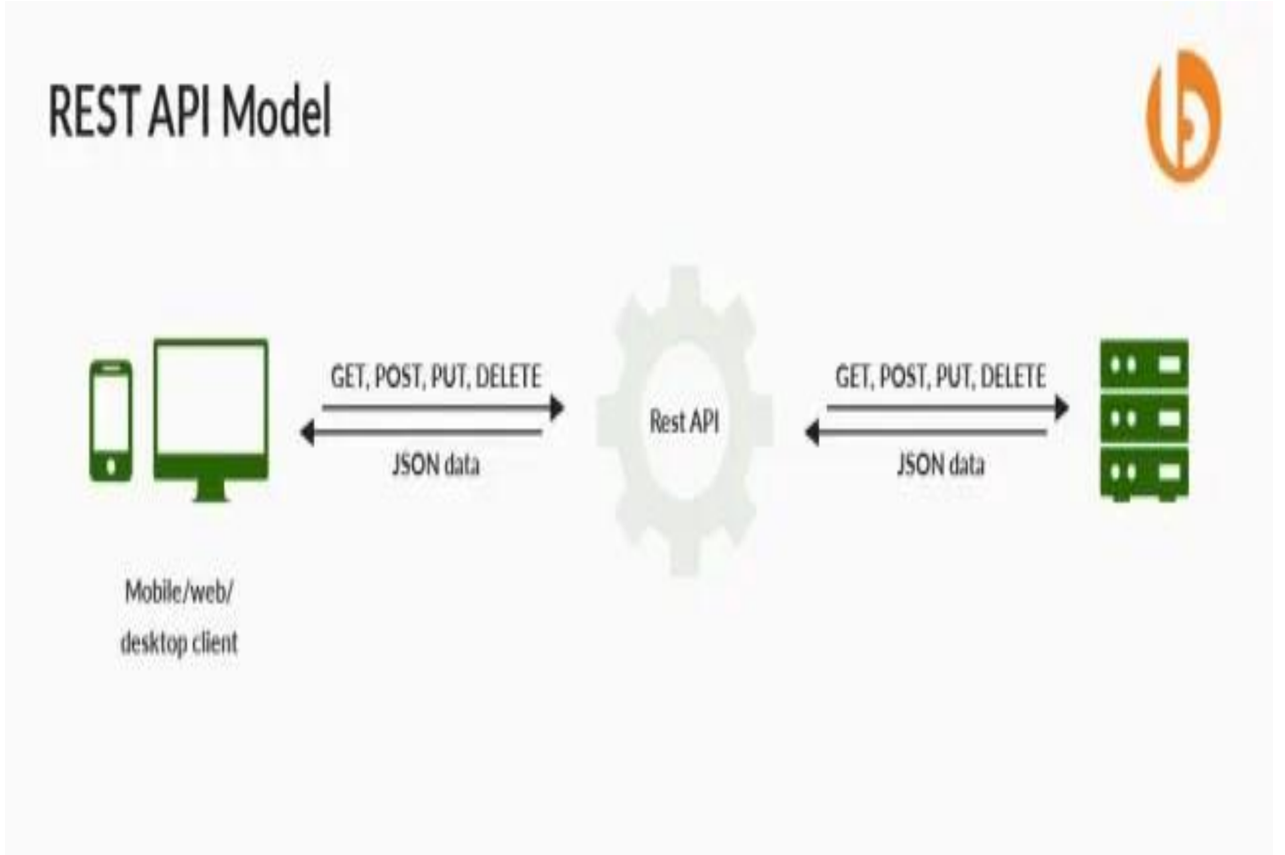
- It is a papoluar JavaScript library to make HTTP requests.It works both browser and Node.js environments.
- Axios provides a consistent way of error handling.
- It converts request and response data to/from JSON.
- It allows cancelling requests.

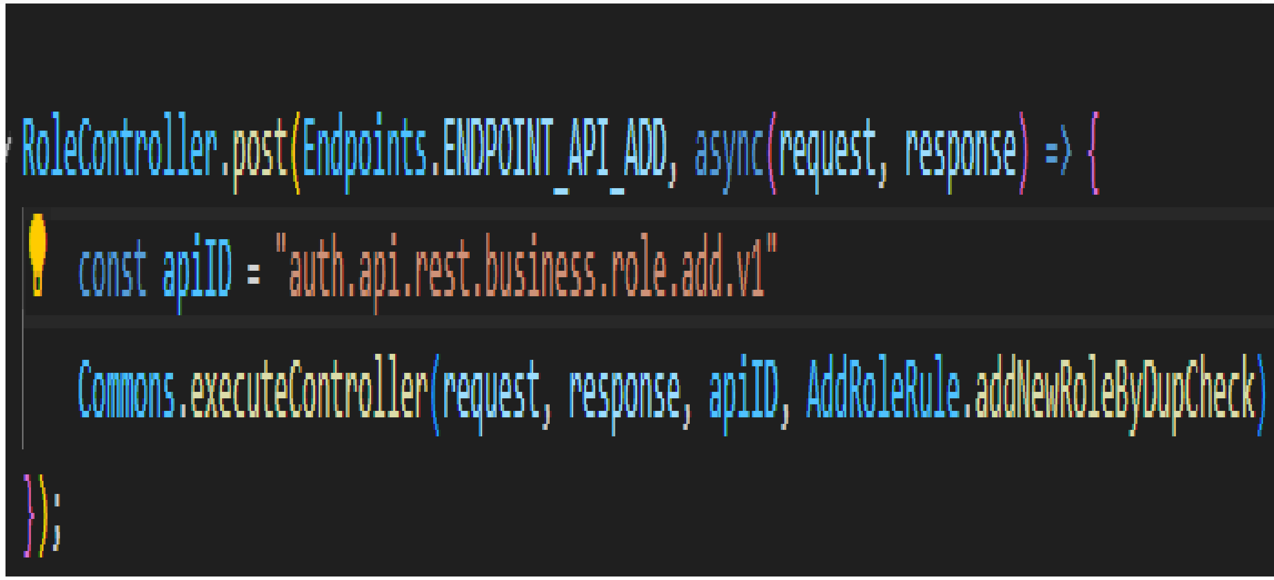


```
    const response = await axios.post(url, body, config)



    let output = response.data
    return Commons.generateServiceOutput(output.outputResponse, output.serviceResponse.status, output.serviceResponse.message)
}
catch(error){
    console.error(error)
    let output = error.response.data;
    if(output.serviceResponse.status == 404) {
        return Commons.generateServiceOutput(output.outputResponse, output.serviceResponse.status, output.serviceResponse.message)
    }


    if(error.response.data.serviceResponse.status == undefined){
        throw Commons.generateServiceOutput(null, 500, error.message)
    }
    // This will be executed when remote service gives the output response in standard format
    throw Commons.generateServiceOutput(output.outputResponse, output.serviceResponse.status, output.serviceResponse.message)
}
```

How to make API

- **API: Application Programming Interface**
- API functions use ‘requests’ and ‘responses’ to authenticate and transfer information. When an API requests information from a web application or web server, it will receive a ‘response.’ The place that APIs send requests or where resources live are called endpoints. An endpoint is a specific address that gives you certain features. In simple terms, it is one end of a communication channel. Each endpoint is a location where APIs can access the resources needed to carry out various operations.
- Here we are dealing with CRUD operations and writing a logic for implementation.

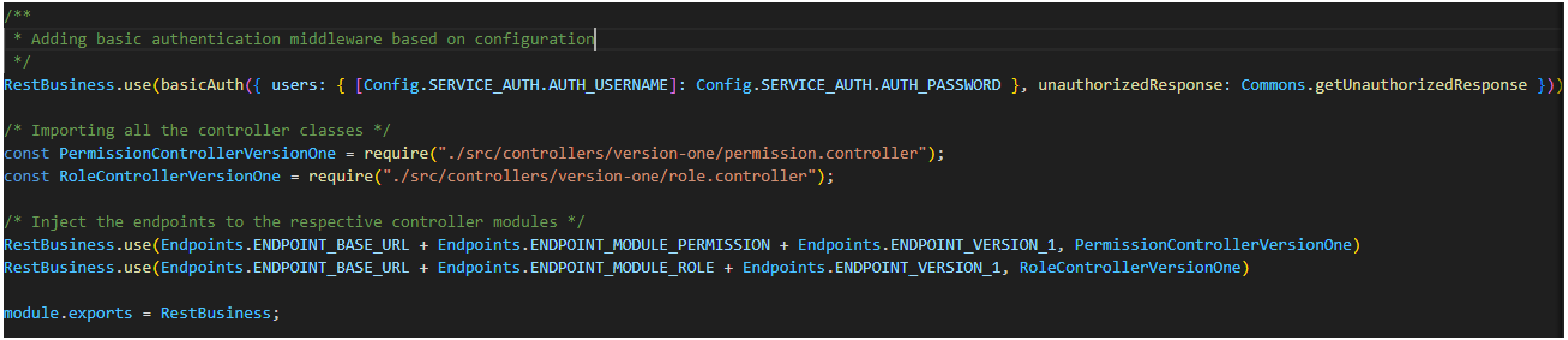




```
RoleController.post(Endpoints.ENDPOINT_API_ADD, async(request, response) => {
    const apiID = "auth.api.rest.business.role.add.v1"
    Commons.executeController(request, response, apiID, AddRoleRule.addNewRoleByDupCheck)
});
```

How to add middlewares and endpoints

- Add Basic authentication Middleware based on the configuration files.
- Here , first we have to initialize object for the imported classes , by using that we can import controller class object and representing the path to the module you want to import .
- After that add endpoint for the respective controller modules with 'use' method, this is used to mount a middleware function or another router at a specific path. This endpoint will give the entire URL path to use.



```
/**
 * Adding basic authentication middleware based on configuration
 */
RestBusiness.use(basicAuth({ users: { [Config.SERVICE_AUTH.AUTH_USERNAME]: Config.SERVICE_AUTH.AUTH_PASSWORD }, unauthorizedResponse: Commons.getUnauthorizedResponse }))

/* Importing all the controller classes */
const PermissionControllerVersionOne = require("./src/controllers/version-one/permission.controller");
const RoleControllerVersionOne = require("./src/controllers/version-one/role.controller");

/* Inject the endpoints to the respective controller modules */
RestBusiness.use(Endpoints.ENDPOINT_BASE_URL + Endpoints.ENDPOINT_MODULE_PERMISSION + Endpoints.ENDPOINT_VERSION_1, PermissionControllerVersionOne)
RestBusiness.use(Endpoints.ENDPOINT_BASE_URL + Endpoints.ENDPOINT_MODULE_ROLE + Endpoints.ENDPOINT_VERSION_1, RoleControllerVersionOne)

module.exports = RestBusiness;
```

Explanation about Endpoints:

Endpoints.ENDPOINT_BASE_URL + Endpoints.ENDPOINT_MODULE_ROLE + Endpoints.ENDPOINT_VERSION_1:

•This is a concatenation of strings or routes. The code is building a URL path by combining several predefined constants.

•**Endpoints.ENDPOINT_BASE_URL**: This likely represents the base URL for your endpoints. It could be something like '/api'.

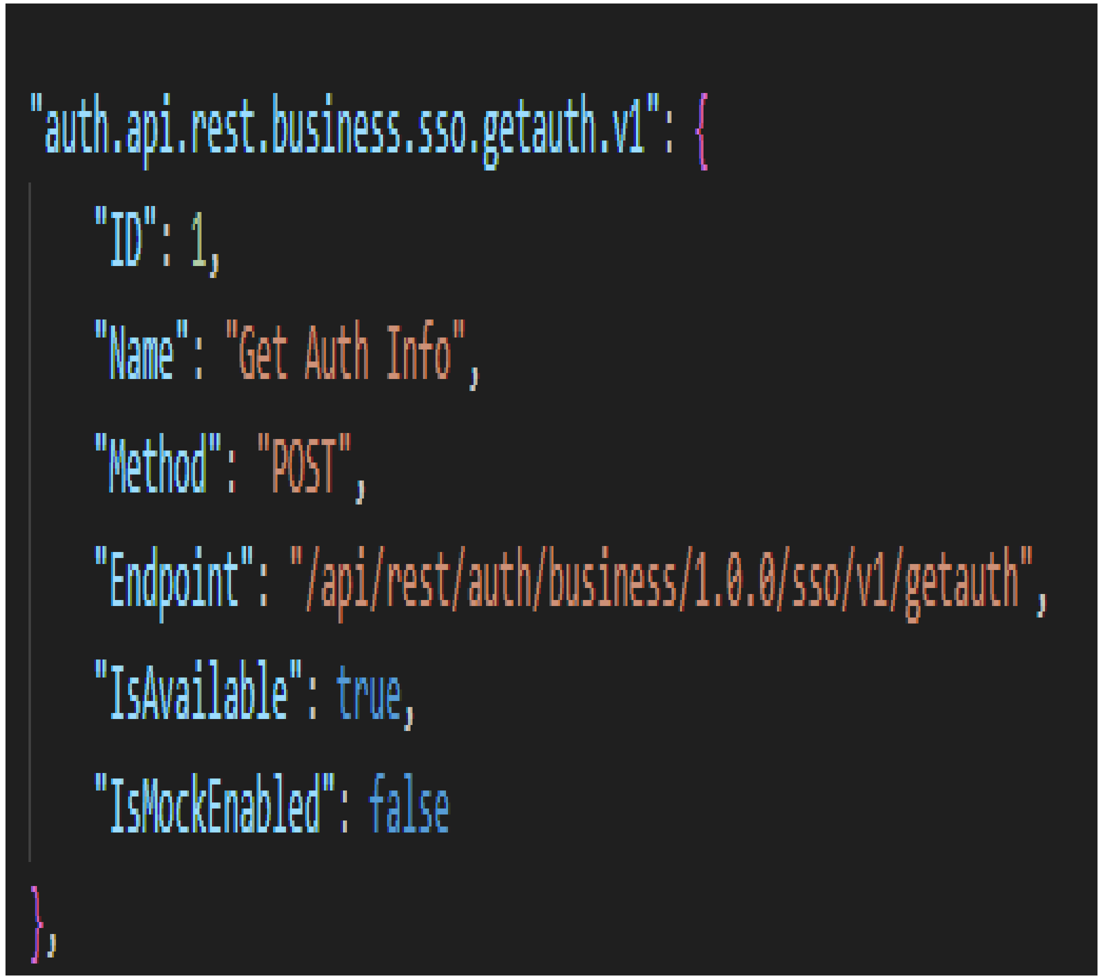
•**Endpoints.ENDPOINT_MODULE_ROLE**: This represents the specific module for the endpoint, probably indicating that these endpoints deal with roles. It could be something like '/roles'.

•**Endpoints.ENDPOINT_VERSION_1**: This indicates the version of the endpoint, allowing for versioning in APIs. It might be something like '/v1'.

If we take the example values above, the concatenated URL path would become '/api/roles/v1'.

How to write configuration

- **Api.config.json** - isn't a standard or universally recognized configuration file by default, but the naming convention suggests that it's a JSON configuration file used for setting up an API. Configuration files, in general, are used to centralize configurations, parameters, and settings that an application or service will utilize. This allows for easier adjustments without diving into the codebase.
- If we create one API we have to give a configuration about method , endpoints and mock enabling and disabling will also done here.



```
"auth.api.rest.business.sso.getauth.v1": {
    "ID": 1,
    "Name": "Get Auth Info",
    "Method": "POST",
    "Endpoint": "/api/rest/auth/business/1.0.0/sso/v1/getauth",
    "IsAvailable": true,
    "IsMockEnabled": false
},
```

What is Executecontroller ?

- Mainly this is a function which is used to control all API's
- This function executes controller functions.
- Here we get execution start time , end time , current date and Header keys.

THANK YOU