**Mini Capstone Project - 1**

**Developing a Server-Side application using Node & Express**

**Instructions to use the project file provided:**

* Read the problem statement, examples and the other details provided carefully and implement the solution.
* Download the project **EOWebService\_toTrainee** into your system and unzip it.
* Install **node\_modules** using the command **npm install** and **start** your server.
* Once the server started successfully, hit the **URL** **http://localhost:5000/set-up-db**  **(GET request)** to setup your database with initial set of documents.
* **DO NOT** alter the function name or the argument list of the function that is provided to you.
* **DO NOT** add any new functions apart from the those given.
* **DO NOT** write codes that result in infinite loops/infinite recursive calls, because it just won’t work!

**Problem Description:**

The Infy HR department wants to automate the employee onboarding and management process. To do this, they want to implement the following features:

* See a list of all employees.
* Add a new employee to the list.
* Update the information of an existing employee on the list.
* Remove an employee from the list.

**In EOWebService\_toTrainee Folder:**

|  |  |
| --- | --- |
| **File Name** | **Description** |
| src/model/setup\_db.js | **Implemented** |
| src/model/employee.js |
| src/utilities/connection.js |
| src/utilities/requestLogger.js |
| src/routes/routing.js | **Partially Implemented** |
| src/utilities/errorLogger.js |
| src/utilities/validator.js |
| src/service/allocate.js |
| src/model/allocate.js |
| src/app.js |

**model/employee.js : (Implemented)**

This file contains a Class **Employee**, which converts a generic object to **Employee** object.

**utilities/requestlogger.js : (Implemented)**

This file consists of a middleware function which logs all the requests made by the users into a text file **RequestLogger.txt.**

**utilities/connection.js : (Implemented)**

This file contains the database schema and getDBModel() method which gives model object of Employee collection.

The database consists of the Employee schema as given below :

|  |  |  |
| --- | --- | --- |
| **Schema** | **Field Name** | **Type** |
| **Employee** | empId | Number |
| empName | String |
| empAge | String |

**model/setup\_db.js : (Implemented)**

This file contains the initial set of data for the collection and a method to insert the data into the respective collection in database.

**utilities/errorlogger.js : (Partially Implemented)**

* This file contains a function **errorLogger** which should log the full stack trace of the error that may get thrown during execution of the program.
* If there is some error in the code the entire error stack should be appended to the **ErrorLogger.txt** file with the timestamp of when the error had occurred.
* If there is any error in appending the error to the **Errorlogger.txt**, it should display the message **“Failed in logging error”** in console.
* If error object’s **status** property is set, then the **response status** should be setas the error object’s **status** value.
* Else, the **response status** should be setto **500** and the error message should be sent as a **JSON** in the given format **{“message” :<<message>>}**
* **errorLogger** should be exported as a module.

**utilities/validator.js : (Partially Implemented)**

This file contains a validator object with three methods.

* **validateEmpId** –
  + This method should accept **empId** as a parameter and validate it.
  + **empId** should be of 4 digits.
  + If the validation fails, it should throw an error with message “**Employee Id should be of 4 digits**” after setting error status as **400**.
* **validateEmpName** –
  + This method should accept **empName** as a parameter and validate it.
  + **empName** should start with capital letter and contain only alphabets having minimum length as 3 letters.
  + If the validation fails, it should throw an error with message “**Employee name should start with capital letter and contain only alphabets having minimum length as 3 letters**” after setting error status as **400**.
* **validateEmpAge** –
  + This method should accept **empAge** as a parameter and validate it.
  + **empAge** should between the range of 20 to 60 years.
  + If the validation fails, it should throw an error with message “**Employee age should between the range of 20 to 60** **years**” after setting error status as **400**.

**model/allocat.js : (Partially Implemented)**

* Instance of **connection** module is created by importing it.
* This file contains **allocate** object with the following methods :
  + addSingleEmployee()
  + findEmployees()
  + deleteEmployee()
  + updateEmployee()

**Note** : Details of the mentioned methods are given in the next section.

* Finally, **allocate** object is exported as a module.

**allocate.addSingleEmployee() : (To Be Implemented)**

* This method takes Employee’s Object as an argument which contains details about Employee to be add.
* Connection to the database should be established by invoking the **getDBMethod()** method of the connection object.
* If Employee added successfully then return an **Employee Id** as a response else throw an error as “**Failed to add an Employee**” with status as **500**.

**allocate.findEmployees() : (To Be Implemented)**

* This method is responsible for fetching details of all the Employees present in the database.
* Connection to the database should be established by invoking the **getDBMethod()** method of the connection object.
* If the details of all the employees fetched successfully then return fetched data as a response else throw an error as “**Failed to fetch the Employees**” with status as **500**.

**allocate.deleteEmployee() : (To Be Implemented)**

* This method takes **Employee Id** as an argument.
* Connection to the database should be established by invoking the **getDBMethod()** method of the connection object.
* Based upon **Employee Id** provided, If the Employee’s details deleted successfully then return an **Employee Id** as a response else throw an error as “**Failed to delete an Employee**” with status as **500**.

**allocate.updateEmployee() : (To Be Implemented)**

* This method takes **Employee Id** and **Employee Object** as the arguments.
* Connection to the database should be established by invoking the **getDBMethod()** method of the connection object.
* Based upon **Employee Id** provided, update the details of **Employee** with the details provided through an **Employee Object**. If the Employee’s details updated successfully then return an **Employee Id** as a response else throw an error as “**Failed to update an Employee**” with status as **500**.

**service/allocat.js : (Partially Implemented)**

* Import all the required modules.
* **allocate** object is created with the following methods.
  + findAllEmp()
  + addSingleEmp()
  + updateEmp()
  + deleteSingleEmp()

**Note** : Details of above methods are given below.

* Finally, export **allocate** objectas a module.

**allocate.findAllEmp() : (To Be Implemented)**

* This method should not take any parameter and return fetched data about all the Employees.
* If failed, then throw an error.

**allocate.addSingleEmp() : (To Be Implemented)**

* This method should take single parameter which contains details about Employee to be add.
* Before adding the details, validate Employee’s Id, Name and Age.
* If failed, then throw an error.

**allocate.updateEmp() : (To Be Implemented)**

* This method should take two parameters i.e., Employee’s Id and Employee’s Object which contains details about Employee to be update.
* Before updating the details, validate Employee’s Id.
* If failed, then throw an error.

**allocate.deleteSingleEmp() : (To Be Implemented)**

* This method should take single parameter which contains Employee’s Id of Employee to be delete/remove from the database.
* Before deleting the Employee, validate Employee’s Id.
* If failed, then throw an error.

**routes/routing.js : (Partially Implemented)**

* Import all required modules.
* Configure the instance of Router to handle the **POST, GET, DELETE** and **PATCH/PUT** request for the given URI.
* The URIs should be configured for adding, fetching, deleting and updating details of the Employees respectively.

**URIs :**

1. **/employee –**
   * It handles the **POST** request for the given URI.
   * Once the request is received, it should create **Employee** object with the values present inside the **request body**.
   * It should invoke **addSingleEmp()** method of **allocateService** by passing the **Employee’s object**, which returns **Employee’s Id** wrapped inside a promise.
   * If the promise is successful, then it should populate the **JSON** response in the given format: ***{"message": "*** ***Employee with empId: <<empId>> added successfully !”}****.*
   * Else if the promise fails, it should forward the control to next handler by passing the error object as parameter.
2. **/employees -**
   * It handles the **GET** request for the given URI.
   * Once the request is received, it should invoke **findAllEmp()** method of **allocateService** which returns **all the** **Employee’s** details present in the database wrapped inside a promise.
   * If the promise is successful, then it should populate the **JSON** response with the data received about all the Employees*.*
   * Else if the promise fails, it should forward the control to next handler by passing the error object as parameter.
3. **/employee/:empId –** 
   * It handles the **DELETE** request for the given URI.
   * Once the request is received, it should invoke **deleteSingleEmp()** method of **allocateService** by passing the **Employee’s Id** which we want to delete.
   * If the promise is successful, then it should populate the **JSON** response in the given format: ***{"message": "*** ***Employee with id: <<empId>> deleted successfully !”}****.*
   * Else if the promise fails, it should forward the control to next handler by passing the error object as parameter.
4. **/employee/:empId –**
   * It handles the **PATCH** request for the given URI.
   * Once the request is received, it should create **Employee** object with the values present inside the **request body**.
   * It should invoke **updateEmp()** method of **allocateService** by passing the **Employee’s Id** which passed through the URI as a parameter and **Employee’s object** created.
   * If the promise is successful, then it should populate the **JSON** response in the given format: ***{"message": "*** ***Employee with id: <<empId>> updated successfully !”}****.*
   * Else if the promise fails, it should forward the control to next handler by passing the error object as parameter.

**app.js : (Partially Implemented)**

* All necessary modules should be imported.
* Middleware should be organized appropriately for its desired functionality.
* The application should listen for requests on port **5000**.

**---------------------------- ALL THE BEST -----------------------------**