DOP: / /2023 **DOS: / /2023**

Experiment No: 07

Title Build a REST-ful API(CRUD) using MongoDB.

Theory:

🔷API:

API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user—establishing the content required from the consumer (the call) and the content required by the producer (the response).

Mongoose is an ODM (Object Document Mapping) tool for Node.js and MongoDB. It helps you convert the objects in your code to documents in the database and vice versa.

Before proceeding to the next section, Please install MongoDB in your machine if you have not done already. Checkout the official MogngoDB installation manual for any help with the installation.

Selecting the appropriate database for your REST API is crucial, and based on current trends, MongoDB is one of the most popular databases for web applications. MongoDB REST API is simple to set up and allows you to store and retrieve documents, making it great for Unstructured Data. Using Express JS as the backend web server with MongoDB as the document store is a common way of implementing the MongoDB REST API strategy. This approach conveniently links MongoDB's Document Model to the JSON-based REST API payloads. You can use Express to build a backend middle tier that runs on Node.js and exposes REST API routes to your application. The Node.js Driver also connects the Express.js server to the MongoDB Atlas cluster.

🔷Steps:

Step 1: Setting up the Project If you have Node.js installed, you can run the following command to start the application from the command line:

```
npm init -y
```

The above command will create a package.json file.

Step 2: Installing Application Dependencies For MongoDB REST API to run, you need a file that will serve as the application's command center. When you ask npm to run your application, it will initially run this file. This file can include object instances of both your modules and third-party modules installed from the npm directory. You created a file named app.js, which will be the application's main entry point, and installed a few dependencies that are required to run your application using the commands above.

```
touch app.js

npm install express mongodb body-parser –save
```

These are the dependencies:

- Express: A framework for Node.js.
- MongoDB: The MongoDB team has supplied an official module to let your Node.js application communicate with MongoDB.
- Body-parser: It will handle request bodies with Express.

Step 3: Run Code Here, you are importing the dependencies you downloaded before. Use the Express object to initialize the express framework, which will utilize the express framework to start the server and run your application on a certain port, as well as configure the body-parser, which is a middleware that parses incoming chunks of data.

```
const Express = require("express");
const BodyParser = require("body-parser");
const MongoClient = require("mongodb").MongoClient;
const ObjectId = require("mongodb").ObjectID;
var app = Express();
app.use(BodyParser.json());
app.use(BodyParser.urlencoded({ extended: true }));
app.listen(5000, () => {});
```

Step 4: Testing Application for MongoDB REST API

node app.js

Step 5: Establishing Connection with MongoDB REST API For this, you will need the MongoDB REST API connection string. Choose Clusters from the Atlas dashboard, then the Overview page, and then the Connect button. You will need to add the string to the app.js file and perform the following code adjustments.

```
const Express = require("express");
const BodyParser = require("body-parser");
const MongoClient = require("mongodb").MongoClient;
const ObjectId = require("mongodb").ObjectID;
const CONNECTION_URL = ;
const DATABASE_NAME = "accounting_department";
var app = Express();
app.use(BodyParser.json());
app.use(BodyParser.urlencoded({ extended: true }));
var database, collection;
app.listen(5000, () => {
    MongoClient.connect(CONNECTION_URL, { useNewUrlParser: true },
(error, client) => {
        if(error) {
            throw error;
        }
        database = client.db(DATABASE_NAME);
        collection = database.collection("personnel");
        console.log("Connected to `" + DATABASE_NAME + "`!");
    });
});
```

Step 6: Build MongoDB REST API Endpoint Next, you will need to establish and query endpoints for the data. To add the data, we'll need to construct an endpoint. To app.js, add the following code:

```
app.post("/personnel", (request, response) => {
    collection.insert(request.body, (error, result) => {
        if(error) {
            return response.status(500).send(error);
        }
        response.send(result.result);
    });
});
```

Step 7: Testing the MongoDB REST API

```
curl -X POST \
    -H 'content-type:application/json' \
    -d '{"firstname":"John","lastname":"Doe"}' \
    http://localhost:5000/personnel
```

You'll see the personnel record for John Doe added into the MongoDB database of 'accounting department.'

GET

Now let's create an endpoint to retrieve all the records data. Add the following code to app.js:

```
app.get("/personnel", (request, response) => {
    collection.find({}).toArray((error, result) => {
        if(error) {
            return response.status(500).send(error);
        }
        response.send(result);
    });
});
```

The goal here is to return all data in our collection representing people. We have no query conditions, hence the empty {} in the find command, and the results get converted into an array. Let's test this out using cURL, a command-line tool for transferring data and supports HTTP; a very good ad-hoc tool for testing REST services.

```
curl -X GET http://localhost:5000/personnel
```

Conclusion: - We learned how to create a REST-ful API using MongoDB.