CLass 2 - Middlewares and CRUD in DB

Revision

1. Routes and handlers
    a. Like for button click, we had handlers for the click event similarly we have handlers for routes
    b. So in the code that we had written

```
app.get("/api/users", (req, res) => {
  console.log("get request");
  res.status(200).json({
    message: "User list",
    data: {
      name: "John Doe",
      age: 25,
    },
  });
});
```

We can simply have a handler function for better readability

```
app.get("/api/users", getUserHandler);
function getUserHandler(req, res) {
  console.log("get request");
  res.status(200).json({
    message: "User list",
    data: {
      name: "John Doe",
```

```
      age: 25,
    },
  });
}
```

c. Similarly we can create handlers for all the routes

```
/** ROUTES */
app.get("/api/user", getUsers);
app.post("/api/user", createUser);
app.get("/api/user/:id", getUserByID);

/** Route handlers */
function getUsers(req, res) {
  (req, res) => {
    try {
      let msg = "";
      if (userData.length === 0) {
        msg = "No data found";
        // throw new Error('No data found')
      } else {
        msg = "Data found";
      }
      res.json({
        status: 200,
        data: userData,
        message: msg,
      });
```

```
    } catch (err) {
      res.status(500).json({
        message: err.message,
      });
    }
  };
}

function createUser(req, res) {
  const userInput = req.body;
  const isEmpty = Object.keys(userInput).length === 0;
  if (isEmpty) {
    return res.status(400).json({
      status: 400,
      message: "No data found",
    });
  } else {
    const id = short.generate();
    const userDetails = req.body;
    userDetails.id = id;
    console.log(userDetails);
    userData.push(userDetails);
    // write to file
    fs.writeFile("./data.json", JSON.stringify(userData), (err)
=> {
      if (err) {
```

```
      console.log(err);
    }
   });

   res.json({
    status: 200,
    data: req.body,
    message: `User created with id ${id}`,
   });
  }
}

function getUserByID(req, res) {
 try {
  const { id } = req.params;
  console.log("64", req.params);
  const user = userData.find((user) => user.id == id);
  console.log("user", user);
  if (user == undefined) {
   throw new Error("User not found");
  } else {
   return res.status(200).json({
    message: user,
   });
  }
 } catch (err) {
```

```
      return res.status(500).json({
        message: err.message,
      });
    }
  }
```

ENV file

1. Used to store different keys and secret passphrases and passwords, connection URLs in env file which is not shipped as part of the code
2. WE need a package called dotenv and the below code
    a. require("dotenv").config();
    b. Now our variables in env files are accessed using **process.env**
3. We will be integrating a payment gateway and we will keep the keys for that in the env file

Databases

1. Why do we need
    a. Store
    b. Retrieval
    c. Search, index
    d. Validation (some fields cannot be empty, structure of data being stored)

2. MOngoDB
    a. When you use MongoDB, you typically interact with the database using documents that look very much like JSON
    b. MongoDB stores data in a format called BSON, which stands for Binary JSON. BSON is a binary representation of JSON-like documents.
    c. MongoDb is a type of NoSql database
        i. What is a NoSql database
            1. NoSQL databases are not primarily structured as a set of tables. They can store and manage data in formats such as key-value pairs, documents, graphs, or wide-column stores.
            2. Unlike SQL databases, which require a predefined schema and structured data, NoSQL databases allow the storage of data without a predefined schema. This makes them ideal for handling semi-structured or unstructured data.
            3. Types of NoSQL Databases:
                a. Document Databases: Store data in documents similar to JSON objects. Each document contains pairs of fields and values. Examples include MongoDB and CouchDB.
                b. Key-Value Stores: Store data as a collection of key-value pairs. Examples include Redis and DynamoDB.

c. Wide-Column Stores: Store data in tables, rows, and dynamic columns. They are excellent for querying large datasets. Examples include Apache Cassandra and Google Bigtable.
d. Graph Databases:

ii. In MongoDB, which is a NoSQL database, data is organized into **collections** and **documents**:

iii. Documents:

1. In MongoDB, a document is a single record in a collection, analogous to a row in a relational database table.
2. They represent a single object like a user

iv. Collections:

1. A collection is a grouping of MongoDB documents. It is the equivalent of a table in a relational database system. Collections are schema-less, meaning the documents within a collection can have different fields.

Mongoose

1. "Mongoose is a tool for Node.js that helps you work with MongoDB (a type of database). It makes some of the more complicated parts of using MongoDB easier.

2.
3. Here's what Mongoose does:
    a. Defining Data Structure / Schemas : It lets you set rules for your data, like what kind of information can be stored and how it should look. This is like creating a form with specific fields that need to be filled out.
    b. Creating Data Models: Think of Mongoose models as templates for your data. They help you create and read data that follows the rules you set.
    c. Checking Data: Before you save your data, Mongoose checks it to make sure it follows your rules. This helps prevent mistakes.
    d. Finding and Using Data: Mongoose makes it easier to search for specific pieces of data and do things with them, like updating or deleting.
    e. In short, Mongoose is like an assistant that helps you manage and use your data in MongoDB, making sure everything is organized and correct."
    f. Just like how express makes writing node code easier, similarly

MongoDB signup / setup

1. Signup
2. Create an org
3. Create a product

4. Create something called as cluster where your data will be stored
5. Follow steps one after another
6. Network access - where you need to add the server;s ip address from where your request to DB will be sent
7. For development purpose, we allow from anywhere
8. Next for database access
    a. Creating a new user
    b. Copy the generated password to env file
9. Go to database to get the db url
    a. Click on connect
    b. Select driver
    c. Copy the url
    d. Can paste it to env

MOngoose

1. Install mongoose package

```
/** Database connection */
mongoose.connect(process.env.DB_URL).then((connection) => {
  console.log("DB connected");
}).catch((err) => {
  console.log("DB connection failed");
})
/** DB connection ends */
```

MongoDB Entity, Schema and Models

1. Let us understand a little more about schemas and models
2. Entity
    a. An entity is like an object or a thing in the real world that has information we want to store. For example, if we are making a database for a school, student data can be an entity.
    b. For ecommerce website, products can be an entity, users, reviews can be different entities
    c. Different category of data
3. Schema
    a. A schema in MongoDB defines the structure of the data for an entity.
    b. Consider a schema as a form that a student fills out on their first day at school.
    c. In addition, It has certain rules and requirements that has to be followed like certain fields are required, data type constraints , etc

4. Model
    a. It's like an inerface to create, read, update, and delete entities in the database.
5. Summary - In MongoDB: We first define what information we want to store about each thing (entity) using a schema. Then, we

use a model to actually work with this data - like adding new data, changing it, or finding it when we need it.

DEFINING SCHEMAS

1. After the connection is made, we then define a schema

```
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  password: String,
confirmPassword: String,
createdAt: Date,
  id: String,
});
```

2. See the mongoose docs for different schema types
3. What are different validations that we can think of for this schema
    a. Name , email and some other fields as required
    b. Pwd and confirmPassword to match

```
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
```

```
    required: true,
    unique: true
  },
  address: String,
  password:{
    type: String,
    required: true,
    minlength: 8
  },
  confirmPassword:{
    type: String,
    required: true,
    validate: {
      validator: function(){
        return this.password === this.confirmPassword
      },
      message: "Password and confirm password should be
same"
    }
  },
  id: String,
});
```

c.

4. We then create models based on the schema defined
5. const User = mongoose.model("User", userSchema);

6. So when we use this model, entries will be added/ updated to the users collection

Mongoose Queries

1. use queries that mongoose provides for CRUD operations

2. Remove the fs module and uuid dependency
3. Create User
4. Make your createUser function as async

```
async function createUser(req, res){
 try{
  const userDetails = req.body;
  const user = await User.create(userDetails);
  res.status(201).json({
   message: "User created successfully",
   data: user
  })
 }catch(err){
  res.status(500).json({
   message: err.message,
  });
 }
}
```

5. Save and check data in mongodb
6. Note that collection name is pluralised form of the model

7. Password encryption
    a. Right now we are storing the password directly
    b. We will later use a library for encrypting the password
8. Getuser

```
async function getUser(req, res) {
 try {
  const userData = await User.find();
  if(userData.length === 0){
   throw new Error("No user found")
  } else {
   res.status(200).json({
    message: userData,
   });
  }

 } catch (err) {
  res.status(500).json({
   message: err.message,
  });
 }
}
```

    a.
9. getUserById

```
async function getUserById(req, res) {
 try {
  const { id } = req.params;
```

```
    console.log("64", req.params);
    const user = await User.findById(id);
    console.log("user", user);
    if (user == undefined) {
      throw new Error("User not found");
    } else {
      return res.status(200).json({
        message: user,
      });
    }
  } catch (err) {
    return res.status(500).json({
      message: err.message,
    });
  }
}
```