Class 3 - MVC Architecture and Factory design pattern

Patch request

1. Add the routes and handlers

```
app.patch("/api/users/:id", updatedUserByIdHandler);

app.delete("/api/users/:id", deleteUserById);
```

2. Update the patch method

```
async function updatedUserByIdHandler(req, res) {
 try {
   const { id } = req.params;
   const updatedUserData = req.body;
   const updatedUser = await User.findByIdAndUpdate(id,
updatedUserData, {
     new: true,
   });
   if (!updatedUser) {
     // throw new Error("No user found")
     res.status(404).json({
       message: "user not found",
     });
   } else {
     res.status(200).json({
       message: "success",
       data: updatedUser,
     });
   }
 } catch (err) {
   res.status(500).json({
    message: "error",
    data: err.message,
  });
 }
}
```

Note the await User.findByIdAndUpdate(id, updatedUserData, {

   new: true,

  });

This new:true will return the updated object

3. Update the delete handler

```
async function deleteUserById(req, res) {
 try {
   const { id } = req.params;
   const deletedUser = await User.findByIdAndDelete(id);
   if (!deletedUser) {
    // throw new Error("No user found");
     res.status(400).json({msg:"no user found"})
   } else {
     res.status(200).json({
       message: "success",
       data: deletedUser,
     });
   }
 } catch (err) {
   res.status(500).json({
     message: "error",
     data: err.message,
   });
 }
}
```

MVC Architecture

1. MVC as an architecture addresses these challenges by organizing code into three interconnected components, each with a specific responsibility. This separation makes the application more manageable, scalable, and maintainable.
2. **Model**: This is the component that deals with the data . In a Node.js/Express app, models often interact with a database to store and retrieve data.
3. **View**: The View component is used for all the UI logic of the application. In the context of Node.js and Express, views are typically templated files which are rendered and sent to the client's browser.
4. **Controller**: Controllers act as an intermediary between models and views. They control the flow of data between the model and view components. In Node.js/Express, controllers are often route handlers, which process incoming requests, perform operations through the models,

Separating our models and schemas

1. Create a new folder models
2. Create a new file userModel.js
3. Copy the schema and the model

4. Import the mongoose
5. Export the User model

```
const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    unique: true
  },
  phone: String,
  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,
  });

  const User = mongoose.model("User", userSchema);

  module.exports = User;
```

6. Import the User in the app.js file
7. const User = require("./models/userModel");

Controller functions

1. What are controller functions
2. The route handlers that we had created earlier in actual are referred to as controller functions
3. Create another folder called as controller
4. Create a new file called userController.js
5. Paste the handler functions in the file.
6. Also make the functions as function expressions
7. Finally export them

```
const User = require("../models/userModel");
/** handlers */

const getUser = async function(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,
     });
   }
 }
  const createUser = async function(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,
     });
   }
 }
  const getUserById = async function(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,
    });
  }
}
module.exports = {
  getUser,
  createUser,
  getUserById
}
```

8. Need to import these methods in the app.js file

```
const { getUser, createUser, getUserById } =
require("./controllers/userController");
```

Chaining

1. Lets say we had to create a check to ensure that the client does not send an empty object while making a POST request.
   a. We can create a middleware function and use it in app.use
   b. Inside userController file, create this method

```
const checkInput = function(req, res, next){
   const userDetails = req.body;
```

```
  const isEmpty = Object.keys(userDetails).length === 0;
  if(isEmpty){
    return res.status(400).json({
      message: "Input fields cannot be empty"
    })
  } else {
      next();
  }
}
module.exports = {
  getUser,
  createUser,
  getUserById,
  checkInput
}
```

c. In the app.js file, befire the routes, add app.use

```
/** Routes */
app.use(checkInput);
app.get("/api/user", getUser);
app.post("/api/user", createUser);
app.get("/api/user/:id", getUserById);
```

e. Now the problem with this is that for every request this middleware will be run

f. **We can chain middlewares**

```
app.post("/api/user", checkInput, createUser);
```

g. different use cases that open up because of this chaining

h. Authentication of routes, rate limiting, parsing input and validation, logging, etc.

Pretext to REST APIS

API, or application programming interface, is a set of rules that define how applications or devices can connect to and communicate with each other

The Problem was each service was being developed differently Different response formats (XML, JSON) . No standard way to define endpoints, HTTP methods were not adopted and used consistently

REST APIs

1. An. A REST API is an API that conforms to the design principles of the REST, or representational state transfer architectural style. For this reason, REST APIs are sometimes referred to RESTful APIs.

2. First defined in 2000 by computer scientist Dr. Roy Fielding in his doctoral dissertation, REST provides a relatively high level of flexibility and freedom for developers. This flexibility is just one reason why REST APIs have emerged as a common method for connecting components
3. Some APIs, such as SOAP or XML-RPC, impose a strict framework on developers. But REST APIs can be developed using virtually any programming language and support a variety of data formats. The only requirement is that they align to the

following six REST design principles - also known as architectural constraints:

1. Uniform interface.
    a. All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). Resources shouldn't be too large but should contain every piece of information that the client might need.
    b. One more thing about the request (although not mentioned as a rule but is followed) is that the routes should be nouns like /users /products . etc.
    c. For example if you have to get reviews for iphone15
        i. Instead of routes like /getReviewsForIphone15
        ii. We have app.get('/reviews/mobiles/:iphone15)
    d. Http methods are verbs like get , put, post

2. Client-server decoupling.
    a. In REST API design, client and server applications must be completely independent of each other. The only information the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application

shouldn't modify the client application other than passing it to the requested data via HTTP.

3. Statelessness.
    a. REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it. In other words, REST APIs do not require any server-side sessions. Server applications aren’t allowed to store any data related to a client request.
    b. Example of Stateless Nature:
        i. Suppose you have a REST API for an online bookstore. A client application wants to place an order for a book.
        ii. Stateful Scenario: If the API were stateful, the client might first log in with a request, and the server might then store the user's authentication state. Subsequent requests for browsing books or placing an order would rely on this stored state for details about the user's authentication.
        iii. Stateless Scenario (RESTful): In a REST API, each request must be self-contained. When the client sends a request to browse books, it includes all necessary information (like an authentication token) within that request. Similarly, when placing an order, the client sends another request containing all the

required data (user credentials, book ID, payment details, etc.) to process the order. The server does not rely on any previous interaction; it treats each request independently

iv. Now why do you think the stateless architecture is scalable. Because if you saved authentication on one server instance, you would need to replicate this on other server instances like payments, orders, etc.

v. With stateless architecture, if one server fails , the other can easily take up the work because it is client agnostic

c.

4. Cacheability.
    a. When possible, resources should be cacheable on the client or server side. Server responses also need to contain information about whether caching is allowed for the delivered resource. The goal is to improve performance on the client side, while increasing scalability on the server side.

5. Layered system architecture.
    a. In REST APIs, the calls and responses go through different layers. As a rule of thumb, don’t assume that the client and server applications connect directly to each other. There

may be a number of different intermediaries in the communication loop. REST APIs need to be designed so that neither the client nor the server can tell whether it communicates with the end application or an intermediary.

6. Code on demand (optional).
    a. REST APIs usually send static resources, but in certain cases, responses can also contain executable code (such as Java applets). In these cases, the code should only run on-demand.

Product Schema and Model

1. Let us add a Product Schema and model in our project and some routes

```
const mongoose = require("mongoose");

const productSchema = new mongoose.Schema({
   name: {
     type: String,
     required: [true, "Product name is required"],
     unique: [true, "Product name should be unique"],
     maxLength: [40, "Product name should be less than 40
characters"],
   },
   price: {
     type: Number,
     required: [true, "Product price is required"],
     validate:{
       validator:function(){
           return this.price > 0
       },
       message: "Price should be greater than 0"
     }
   },
   categories:{
       required:true,
       type: String
   },
   images:{
       type:[String]
   },
   averageRating:Number,
   discount:{
       type:Number,
       validate:{
           validator:function(){
               return this.discount < this.price
           },
           message:"Discount should be less than price"
       }
   },

})

const Product = mongoose.model("Product", productSchema);

module.exports = Product;
```

2. Copy userController.js and create a new productController.js

```
const Product = require("../models/productModel");
/** handlers */

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

  } catch (err) {
    res.status(500).json({
      message: err.message,
    });
  }
}
 const createProduct = async function(req, res){
  try{
    const productDetails = req.body;
    const product = await Product.create(productDetails);
    res.status(201).json({
      message: "Product created successfully",
      data: product
    })
  }catch(err){
    res.status(500).json({
      message: err.message,
    });
  }
}
 const getProductById = async function(req, res) {
  try {
    const { id } = req.params;
    const product = await Product.findById(id);
    console.log("user", user);
    if (product == undefined) {
      throw new Error("User not found");
    } else {
      return res.status(200).json({
        message: product,
      });
    }
  } catch (err) {
    return res.status(500).json({
      message: err.message,
    });
  }
}

const checkInput = function(req, res, next){
  const productDetails = req.body;
  const isEmpty = Object.keys(productDetails).length === 0;
  if(isEmpty){
    return res.status(400).json({
      message: "Input fields cannot be empty"
    })
  } else {
      next();
  }
}

const deleteProductById = async function(req, res){
  const id = req.params.id;
  try{
    const product = await Product.findByIdAndDelete(id);
    if(product){
      res.status(200).json({
        message: "Product deleted successfully",
        data: product
      })
```

```
     } else {
       res.status(404).json({
         message: "Product not found"
       })
     }
   } catch(err){
     res.status(500).json({
       message: err.message,
     });
   }
 }
 module.exports = {
   getProducts,
   createProduct,
   getProductById,
   checkInput,
   deleteProductById
 }
```

3. Now create routes for products same as that of user

```
/*** Routes for Products * */
app.get("/api/products", getProducts);
 app.post("/api/products", checkInput, createProduct);
 app.get("/api/products/:id", getProductById);
 app.delete("/api/products/:id", deleteProductById);
```

Problem Statement

1. Same set of code needs to be done for every resource
2. Code duplication - only model is changing

Next lecture: will continue from here with Factory Design pattern