MERN 4 - MVC architecture and Pre/ Post hooks

MVC Architecture
Refactoring Previous code with MVC Architecture
Pre and Post hook

Model:

Represents the data and business logic of the application.
In a Node.js app, the Model could be a Mongoose schema or any other data access layer that interacts with a database. The model is unaware of the user interface.

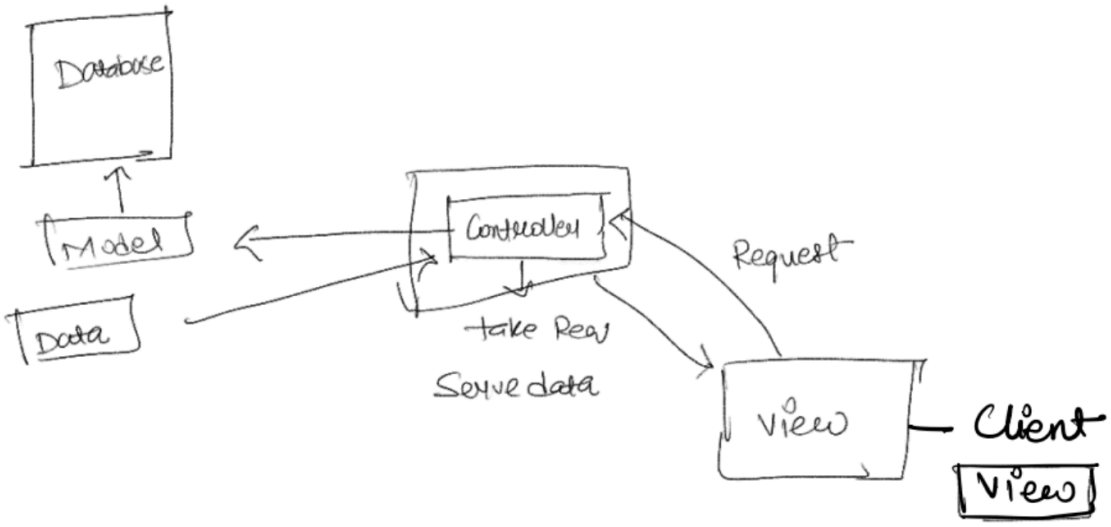
View:

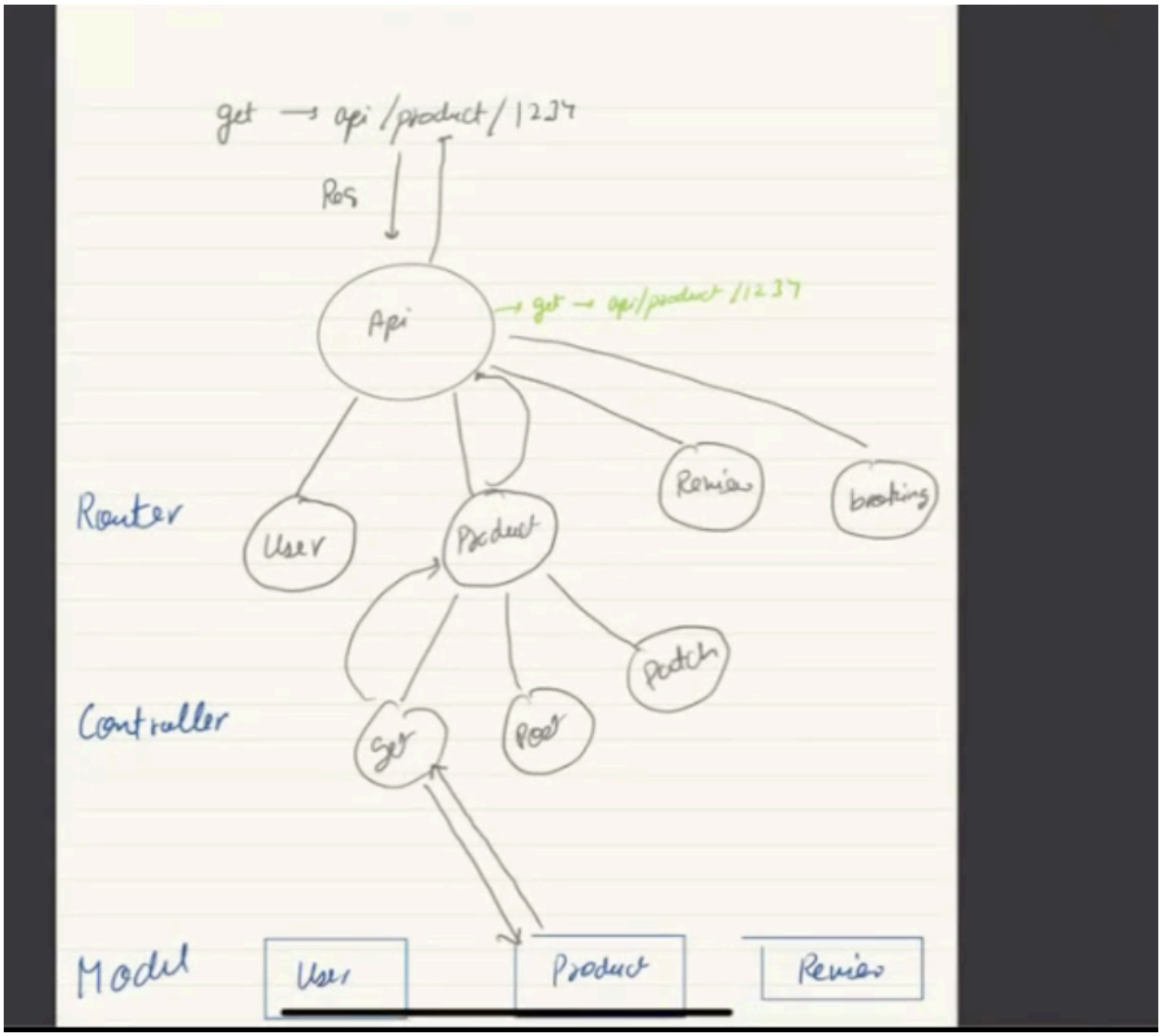
The part of the application that users see and interact with.
In Node.js, this could be HTML templates rendered using a template engine like EJS, Pug, or even a front-end framework like React. Passes user input to the controller.

Controller:

Manages the flow of the application, handles user input, and updates the Model and View.
In Node.js, controllers are usually functions or classes that handle HTTP requests and responses.

The following image gives an idea about MVC architecture:





In essence, MVC separates the application into three distinct components, making it easier to manage and maintain.

The view handles the presentation and user interaction, the model manages the data and logic, and the controller orchestrates the communication between the view and the model.

This separation of concerns enhances code organization and promotes scalability and maintainability in software development.

title: Benifits of MVC Architecture

a. Separation of Concerns:

Divides the application into Model, View, and Controller for clear separation of responsibilities.

b. Modular Development:

Supports development and maintenance of separate, reusable modules for each component.

c. Improved Code Reusability:

Allows reuse of Models, Views, and Controllers in different parts of the application or other projects.

d. Enhanced Maintainability:

Changes in one component have minimal impact on the others, simplifying maintenance and debugging.

e. Scalability:

Facilitates parallel development and the addition of new features without major rework.

f. User Interface Flexibility:

Adapts to various user interfaces while keeping the core logic intact.

g. Efficient Testing and Debugging:

Enables isolated unit testing for each component, easing issue identification and resolution.

h. Parallel Development:

Supports multiple developers or teams working on different components simultaneously.

i. Support for Multiple Views:

Utilizes the same Model and Controller with multiple Views for diverse user interfaces.

j. Long-Term Maintainability:

Promotes organized and understandable code, reducing technical debt over time.

title: Refactoring the Code with MVC

So in the last class we wrote this code with Express and Mongoose

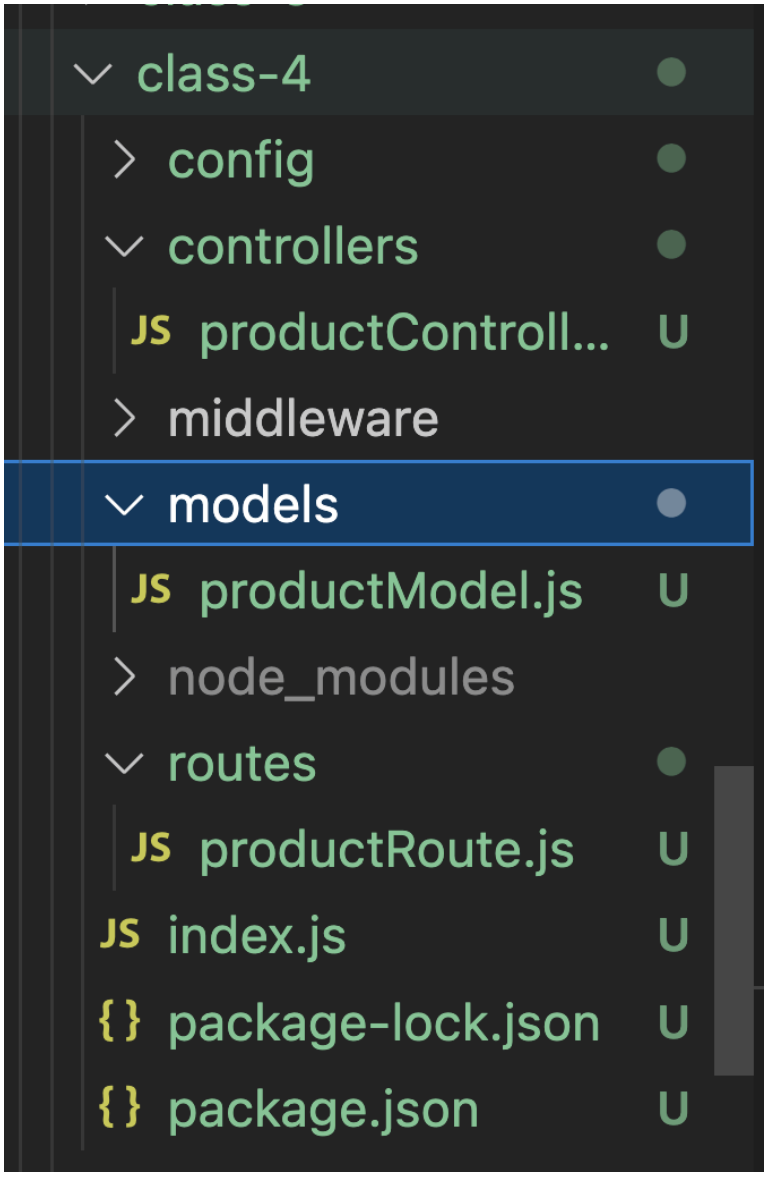
```
const mongoose = require("mongoose");
const express = require("express");

const app = express();
app.use(express.json());

const dbURL =
`mongodb+srv://ayushrajsd:28Wca5DmXC6Q9BDW@cluster0.vg5saeo.mongodb.net/`;

// connect to DB
mongoose
 .connect(dbURL)
 .then(function (connection) {
   console.log("Connected to MongoDB");
 })
 .catch(function (err) {
   console.log(err);
 });

// create a schema

const productSchema = new mongoose.Schema({
 product_name: {
   type: String,
   required: true,
 },
```

```
 product_price: {
   type: String,
   required: true,
 },
 isInStock: {
   type: Boolean,
   default: true,
 },
 category: {
   type: String,
   required: true,
 },
 password: {
   type: String,
   required: true,
   minLength: 8,
 },
 confirmPassword: {
   type: String,
   required: true,
   minLength: 8,
   validate: {
     validator: function () {
       return this.password === this.confirmPassword;
     },
     message: "Password and confirm password should be same",
   },
 },
});

const ProductModel = mongoose.model("products", productSchema);

app.post("/api/products", async function (req, res) {
 const body = req.body;
 console.log(body);
 try{
   const product = await ProductModel.create({
       product_name: body.product_name,
       product_price: body.product_price,
       category: body.category,
       isInStock: body.isInStock,
       password: body.password,
```

```
      confirmPassword: body.confirmPassword,
    });
    console.log(product);
    return res.status(201).json({ message: "Product creted successfully" });
 }catch(err){
   console.log(err);
   return res.status(400).json({message: err.message});
  }
 });

app.get("/api/products", async (req, res) => {
 const allProducts = await ProductModel.find();
 // const allProducts = await ProductModel.find({category: "electronics"});
 console.log(allProducts);
 return res.status(200).json(allProducts);
});

app.get("/api/products/:id", async (req, res) => {
 const id = req.params.id;
 const product = await ProductModel.findById(id);
 res.status(200).json(product);
});

app.put("/api/products/:id", async (req, res) => {
 await ProductModel.findByIdAndUpdate(req.params.id, req.body);
 res.status(200).json({ message: "Product updated successfully" });
});

app.delete("/api/products/:id", async (req, res) => {
 await ProductModel.findByIdAndDelete(req.params.id);
 res.status(200).json({ message: "Product deleted successfully" });
});

app.listen(3000, function () {
 console.log("Server is running");
});
```

Create Project Structure:

1. index.js (main file)
2. config/ (for database connection)
    a. db.js
3. models/ (for Mongoose schemas)
    a. product.js
4. controllers/ (for request handlers)
    a. productController.js
5. routes/ (for route definitions)
    a. productRoutes.js
6. middlewares/ (for any middlewares, if needed)



Refactor

1. Move the Database Connection to config/db.js:

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

// driver
```

```
const dbURL =
`mongodb+srv://ayushrajsd:28Wca5DmXC6Q9BDW@cluster0.vg5saeo.mongodb.net/`;
// once

const connectDB = async () => {
 try {
   await mongoose.connect(dbURL);
   console.log("connected to db");
 } catch (err) {
   console.log(err);
 }
};

module.exports = connectDB;
```

2. Move the Schema and Model to models/product.js:

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

const productSchema = new mongoose.Schema(
 {
   product_name: {
     type: String,
     required: true,
   },
   product_price: {
     type: String,
     required: true,
   },
   isInStock: {
     type: Boolean,
     required: true,
   },
   category: {
     type: String,
     required: true,
   },
   password: {
     type: String,
     required: true,
     minLength: 8,
```

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

const ProductModel = mongoose.model("products", productSchema);

module.exports = ProductModel;
```

3. Move Controllers to controllers/productController.js:
    a. Cut the method definition from index.js and paste it here

```
const ProductModel = require("../models/productModel");

const getAllProducts = async (req, res) => {
 const allProducts = await ProductModel.find();
 // const allProducts = await ProductModel.find({isInStock :true})
 console.log(allProducts);
 return res.status(200).json(allProducts);
};

const getProductById = async (req, res) => {
 const id = req.params.id;
 const product = await ProductModel.findById(id);
 res.status(200).json(product);
};

const updateProductById = async (req, res) => {
 await ProductModel.findByIdAndUpdate(req.params.id, req.body);
 return res.status(201).json({ message: "Resources Updated" });
```

```
};

const createProduct = async (req, res) => {
 const { product_name, product_price, isInStock, category } = req.body;
 try {
   const product = await ProductModel.create({
     product_name: product_name,
     product_price: product_price,
     isInStock: isInStock,
     category: category,
     password: password,
     confirmPassword: body.confirmPassword,
   });

   console.log(product);

   return res.status(201).json({ message: "Product Created" });
 } catch (err) {
   console.log(err);
   return res.status(400).json({ message: "Something went wrong", err });
 }
};

const deleteProductById = async (req, res) => {
 await ProductModel.findByIdAndDelete(req.params.id);
 return res.status(201).json({ message: "Resource Deleted" });
};

module.exports = {
   getAllProducts,
   getProductById,
   updateProductById,
   createProduct,
   deleteProductById,
}
```

4. Move Routes to routes/productRoutes.js:

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

const productRouter = express.Router();
```

```
const {
 getAllProducts,
 getProductById,
 updateProductById,
 createProduct,
 deleteProductById,
} = require("../controllers/productController");

productRouter.get("/", getAllProducts);
productRouter.get("/:id", getProductById);
productRouter.post("/", createProduct);
productRouter.put("/:id", updateProductById);
productRouter.delete("/:id", deleteProductById);

module.exports = productRouter;
```

5. Update index.js

```
const express = require("express");
const connectDB = require("./config/db");
const productRouter = require("./routes/productRoute");

const app = express();

// connect to db
connectDB();

// middlewares
app.use(express.json());

// routes
app.use("/api/products", productRouter);

app.get("/", (req, res) => {
 res.send("Welcome to the product API");
});

app.use((req, res) => {
 res.status(404).json({ message: "Route not found" });
});
```

```
app.listen(3000, () => {
 console.log(`The server is running in port 3000`);
});
```

title:Pre and Post hook

Mongoose, an Object Data Modeling (ODM) library for MongoDB and Node.js, provides middleware (also called hooks) that can be executed at various points in the lifecycle of a document. Pre and post hooks are two types of middleware that allow you to run code before and after certain operations.

Hooks in Mongoose

1. Intuition:
    a. Now there are situations where we want to run some validation before saving the data or run some logging, process after saving the data
    b. Mongoose hooks, often considered as "middleware," are functions that run before or after certain events happen in Mongoose. Think of them like "triggers" in a process.
    c. If we want to process / execute something before the data save for example or after the data save then these hooks come in handy
2. Complementing Schema Rules

a. Complex Validations: While Mongoose schemas are great for basic validations (like type checking, required fields, and simple constraints), pre hooks can handle more complex, custom validations that might be too intricate or specific for the schema definition.
b. Separation of Concerns: By using hooks, you can keep your schema definitions clean and focused on the structure and basic constraints of your data, while hooks can manage the operational or business logic aspects.

3. Pre Hooks
    a. Data Validation and Sanitization: Beyond the built-in validation rules in Mongoose, pre hooks can be used for custom validations or to sanitize inputs before they are saved to the database.
    b. Password Hashing: In user models, pre-save hooks are commonly used to hash passwords before storing them in the database.
    c. Setting Default Values: Automatically setting values for certain fields before saving, especially when these values aren't provided in the input.
    d. Timestamping: Although Mongoose supports automatic timestamping, in some cases, you might need custom timestamp logic that can be implemented in a pre-save hook.

e. Data Transformation: Altering data before it's persisted, like formatting strings, converting units, or setting complex derived fields.
f. Logging and Auditing: Recording activities or changes for auditing purposes just before a document is modified or created.

4. Post Hooks
    a. Logging: Post hooks are useful for logging operations after they have occurred, such as logging the creation or modification of documents
    b. Data Aggregation or Analysis: Performing aggregations or data analysis tasks after a certain operation, like recalculating averages or metrics post-update.
5. in last class we discussed about confirmPassword in the database. Do we need this field to be saved in DB ?

```
productSchema.pre("save", function () {
 this.confirmPassword = undefined;
})
```

    a. Create a product and see the confirmPassword field no more present
6. Let us see one more example where we will restrict creation of products with only certain categories

```
const validCategories = ["electronics", "clothes", "stationery",
"furniture"];
productSchema.pre("save", function (next) {
 const invalidCategories = this.category.filter((category) => {
   return !validCategories.includes(category);
 });
```

```
 if (invalidCategories.length) {
   return next(new Error(`Invalid categories
${invalidCategories.join(",")}`));
 } else {
   next();
 }
});
```

7. Change category in schema to be array of strings

```
category: {
    type: [String],
    required: true,
  },
```

8. Now how the flow looks like
    a. Before the save method is done in the product Controller , this hook is run.
    b. If it runs successfully the save part is done otherwise the catch block will run
    c. Show demo

Let us create a user model and implement a timestamp logger via hook

1. Create a file userModel.js

```
const mongoose = require('mongoose');
// Define a schema
const userSchema = new mongoose.Schema({
   name: String,
   email: String,
   createdAt: Date,
```

```
  updatedAt: Date,
});
 // Pre-save hook to add timestamps
userSchema.pre('save', function (next) {
  const now = new Date();
  this.updatedAt = now;
  if (!this.createdAt) {
    this.createdAt = now;
  }
  next();
});
 // Post-save hook to log a message
userSchema.post('save', function (doc, next) {
  console.log(`User ${doc.name} has been saved.`);
  next();
});
 // Create a model from the schema
const User = mongoose.model('User', userSchema);

  module.exports = User;
```

2. Controller

```
const UserModel = require('../models/userModel');

const createUser = async (req, res) => {
   const { name, email } = req.body;
   try {
       const user = await UserModel.create({
           name: name,
           email: email,
       });
       return res.status(201).json({ message: "User Created" });
   } catch (err) {
       return res.status(400).json({ message: "Something went wrong", error:
err.message });
   }
}
module.exports = {createUser};
```

3. User route

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

const userRoute = express.Router();

const {createUser} = require("../controllers/userController");

userRoute.post("/", createUser);

module.exports = userRoute;
```

4. index.js

```
const express = require("express");
const connectDB = require("./config/db");
const productRouter = require("./routes/productRoute");
const userRouter = require("./routes/userRoute");

const app = express();

// connect to db
connectDB();

// middlewares
app.use(express.json());

// routes
app.use("/api/products", productRouter);
app.use("/api/users", userRouter);

app.get("/", (req, res) => {
 res.send("Welcome to the product API");
});

app.use((req, res) => {
 res.status(404).json({ message: "Route not found" });
});

app.listen(3000, () => {
 console.log(`The server is running in port 3000`);
});
```

Order of hooks

Add few console.logs

```
userSchema.pre('save', function (next) {
   console.log("Pre-save hook")
   const now = new Date();
   this.updatedAt = now;
   if (!this.createdAt) {
     this.createdAt = now;
   }
   next();
 });
  // Post-save hook to log a message
 userSchema.post('save', function (doc, next) {
   console.log(`User ${doc.name} has been saved.`);
   next();
 });
```

In the controller as well

```
const createUser = async (req, res) => {
   console.log("creating user")
   const { name, email } = req.body;
   try {
       const user = await UserModel.create({
           name: name,
           email: email,
       });
        console.log("saved user");
       return res.status(201).json({ message: "User Created" });
   } catch (err) {
       return res.status(400).json({ message: "Something went wrong", error:
err.message });
   }
}
```

Hit the api again and see the order of the logs

```
creating user
Pre-save hook
User  asd1 has been saved.
saved user
```