Class 4 - MongoDB Queries, Search, Sort, Filter

Factory Design Pattern

1. Create a folder called utils
2. Create a file called crudFactory.js
3. The functions take elementModel as input and return the same async function as before

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

   } catch (err) {
     res.status(500).json({
       message: err.message,
     });
   }
 }
  const createFactory = (elementModel) => async function(req, res){
   try{
     const elementDetails = req.body;
     const data = await elementModel.create(elementDetails);
     res.status(201).json({
       message: "Data created successfully",
       data: data
     })
   }catch(err){
     res.status(500).json({
       message: err.message,
     });
   }
 }
  const getElementByIdFactory = (elementModel) => async function(req,
res) {
   try {
     const { id } = req.params;
     const data = await elementModel.findById(id);
     if (data == undefined) {
       throw new Error("Data not found");
     } else {
       return res.status(200).json({
         message: data,
       });
     }
   } catch (err) {
     return res.status(500).json({
       message: err.message,
     });
   }
 }

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

const deleteElementByIdFactory = (elementModel) => async
function(req, res){
  const id = req.params.id;
  try{
    const data = await elementModel.findByIdAndDelete(id);
    if(data){
      res.status(200).json({
        message: "Data deleted",
        data: data
      })
    } else {
      res.status(404).json({
        message: "Data not found"
      })
    }
  } catch(err){
    res.status(500).json({
      message: err.message,
    });
  }
}
module.exports = {
  getAllFactory,
  createFactory,
  getProductByIdFactory,
  checkInput,
  deleteProductByIdFactory
}
```

4. Now we import these methods in our controllers
5. UserController

```
const User = require("../models/userModel");
const {getAllFactory,
   createFactory,
   getElementByIdFactory,
   checkInput,
   deleteElementByIdFactory} = require('../utils/crudFactory');
/** handlers */

const getUserHandler = getAllFactory(User);
  const createUserhandler = createFactory(User);
  const getUserById = getElementByIdFactory(User);

 const deleteUserById = deleteElementByIdFactory(User);
 module.exports = {
   getUserHandler,
   createUserhandler,
   getUserById,
   checkInput,
   deleteUserById
 }
```

6. ProductController

```
const Product = require("../models/productModel");
const {getAllFactory,
   createFactory,
   getElementByIdFactory,
   checkInput,
   deleteElementByIdFactory} = require('../utils/crudFactory');
/** handlers */

const getProductHandler = getAllFactory(Product);
  const createProducthandler = createFactory(Product);
```

```
 const getProductById = getElementByIdFactory(Product);

const deleteProductById = deleteElementByIdFactory(Product);
module.exports = {
  getProducts,
  createProduct,
  getProductById,
  checkInput,
  deleteProductById
}
```

7. checkInput will be imported in app.js from crudFactpory
8.
```
const {checkInput} = require("./utils/crudFactory");
```

Routing

1. One thing that we can still do is separate our routes
2. Create a folder called router and files - productRouter.js and userRouter.js
3. userROuter

```
const express = require("express");
const userRouter = express.Router();
const {getUserHandler, createUserHandler, getUserById,
deleteUserById} = require("../controllers/userController");
const {checkInput} = require('../utils/crudFactory');
/** Routes for user */
// app.use(checkInput);

userRouter.get("/", getUserHandler);
userRouter.post("/", checkInput, createUserhandler);
userRouter.get("/:id", getUserById);
userRouter.delete('/:id',deleteUserById);
```

```
module.exports = userRouter;
```

4. productRouter

```
const express = require("express");
const productRouter = express.Router();
const {getProducthandler,createProductHandler,getProductById,
deleteProductById} =
require("../controllers/productController");
const {checkInput} = require('../utils/crudFactory');
/*** Routes for Products * */
productRouter.get("/", getProductHandler);
productRouter.post("/", checkInput, createProductHandler);
productRouter.get("/:id", getProductById);
productRouter.delete("/:id", deleteProductById);

module.exports = productRouter;
```

5. Update app.js

```
const userRouter = require("./router/userRouter");
const productRouter = require("./router/productRouter");
require("dotenv").config();
const port = process.env.PORT || 3000;

app.use(express.json());
app.use("/api/users", userRouter);
app.use("/api/products", productRouter);
```

Extending Product Model

1. Add description, stock, brand, discount. Also category should be an array of string as the product like earphone can be both under electronics and audio

```
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"
       }
   },
   description:{
       type:String,
       required:[true,'Please provide description'],
       maxLength:[200,'Description should be less than 200
characters']

   },
   stock:{
       type:Number,
       required:[true,'Please provide stock'],
       validate:{
           validator:function(){
               return this.stock >= 0
           },
           message:"Stock should be greater than 0"
       }
   },
   brand:{
       type:String,
       required:[true,'Please provide brand']
   },
   }
)
```

2. Create new products and see that with a different schema your products will be in different state(will have fields as per new schema and there will be products with older schema as well) and this needs to be handled at the application level
3. However there are few additional checks that we can do

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.

   c. use schema validators for straightforward, synchronous validation and pre-hooks for complex, asynchronous tasks or validations that require external data.

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. Creating a pre hook to remove the confirmPassword field

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

    a. Create a user 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",
"furniture", "stationery"];

productSchema.pre("save", function (next) {
 console.log("pre save hook");
 const invalidCategories =
this.categories.filter((category) => {
   return !validCategories.includes(category);
 });
 if (invalidCategories.length) {
   return next(new Error(`Invalid categories
${invalidCategories.join(" ")}`));
 } else {
   next();
 }
});
productSchema.post("save", function () {
 console.log("post save hook");
});
```

7. 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

Query Params

1. In URL everything that starts after question mark
2. It is used for accessing data that is sent in a non-structured way, typically for filtering, sorting, or specifying certain types of requests in a REST API.
3. For eg, In Postman , if we change the url where google search was done and update it to localhost:3000/search?....

   ```
   localhost:3000/search?q=flipkart&rlz=1C5CHFA_enIN1084IN1085&oq=flipka&gs_lcrp=EgZjaHJvbWUqEAgAEAAYgwEY4wIYsQMYgAQyEAgAEAAYgwEY4wIYsQMYgAQyEwgBEC4YgwEYxwEYsQMY0QMYgAQyBggCEEUYOTIKCAMQABixAxiABDINCAQQABiDARixAxiABDIHCAUQABiABDIKCAYQABixAxiABDIGCAcQBRhA0gEIMjI3MmowajeoAgCwAgA&sourceid=chrome&ie=UTF-8
   ```
4. In app.js, add a route to capture this request

```
app.use('/search', function(req,res){
 console.log(req.query)
 res.status(200).json({
   message:"Search successful",
```

```
   data:req.query
 })
})
```

5. In ProductRouter, update get all products handler

```
function getProductsHandler(req,res){
   console.log(req.query)
   res.status(200).json({
       message:"Get products successful",
       data:req.query
   })
}
```

6. In postman, create this url to pass query in the url
   localhost:3000/api/products?sort=price%20asc&select=name%20description%20price

7. %20 to give spaces between the values and use '&' to pass multiple query values
8. Run in postman
9. Try to log now in the method

```
function getProductsHandler(req,res){
   const sortQuery = req.query.sort;
 const selectQuery = req.query.select;
 console.log("sort query ", sortQuery);
   console.log("select query ", selectQuery);

   res.status(200).json({
       message:"Get products successful",
       data:req.query
   })
}
```

10. We ca similarly also have the pagination params
11. HOw this works is mongoose first gets all the data and then on top pf that does the required operations
12. But all of this is done at db level

Sorting based on Param

1. Put the code in try catch block
2. First we will find all the items and then we will call subsequent methods of sort , select etc

```
async function getProductsHandler(req, res) {
 try {
   const sortQuery = req.query.sort;
   const selectQuery = req.query.select;
   /** sorting logic */
 let queryResPromise = Product.find();
 if (sortQuery) {
   const [sortParam, order] = sortQuery.split(" ");
   console.log("sortParam", sortParam);
   console.log("order", order);
   if (order === "asc") {
     queryResPromise = queryResPromise.sort(sortParam);
   } else {
     queryResPromise =
queryResPromise.sort(`-${sortParam}`);
   }
 }
```

```
   const result = await queryResPromise;

   res.status(200).json({
     message: "search successful",
     data: result,
   });
 } catch (err) {
   res.status(500).json({
     message: "Internal server error",
     error: err.message,
   });
 }
}
```

3. In postman, run queries for both asc and desc query localhost:3000/api/products?sort=price%20desc&select=name%20description%20price

Selecting only few params from the documents

1. There might be situation when we need fewer fields in the response from the document.
2. Let us say we only need name, price, description
3. Just add the below if block

```
if(selectQuery){
       queryResPromise = queryResPromise.select(selectQuery)
   }
```