Class 5 - Data validation, aliasing

Pagination

1. Open flipkart and go for mens tshirt and then on page 2 of the listing
2. Copy the url and paste in the postman
   a. Notice the page query param
   b. Choose brand and discount and see the url in postman
   c. Similarly choose the price range and see in postman
3.

```
/**
  * filter the product -> as per the query params
  * sorting the products -> as per the query params
  * select the fields -> as per the query params
  * pagination -> as per the query params
  */
```

4. As per above we will accept the params and update the data
5. Pagination will be done using limit and skip methods on mongoose

```
/**
    * pagination logic will be implemented using limit and skip
    * limit -> number of documents to be returned
    * skip -> number of documents to be skipped
    */
```

6. Update postman url to pass these params

```
localhost:3000/api/products?sort=price%20desc&select=name%20description%20price&page=2&limit=2
```

```
const page = req.query.page  || 1;
    const limit = req.query.limit || 1;
    const skip = (page - 1) * limit;
    console.log("skip",skip)
    queryResPromise = queryResPromise.skip(skip).limit(limit);
```

7. In the postman can check two products in each page
   `localhost:3000/api/products?page=2&limit=2`

Filtering

Mongoose is a wrapper / tool to assist in DB operations

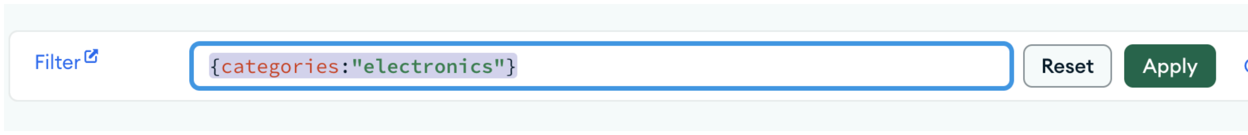
Using mongoose, we can manage schemas
Have methods like find, create, findByID etc that we saw for CRUD

1. For filtering , we will also use MongoDB's rich set of query operators - https://www.mongodb.com/docs/manual/reference/operator/query/
2. Basic filtering with find
   a. const results = await MyModel.find({ fieldName: value });
3. Using query operators
   a. const results = await MyModel.find({ age: { $gt: 18 } });

   b. Basically search for age greater than 18
4. For more granular filtering such chaining expressions can be written

```
const results = await MyModel.find()
                           .where('age').gt(18)
                           .where('name').equals('John Doe');
```

5. Filtering with regular expressions
   a. const results = await MyModel.find({ name: /doe$/i });
   b. This will find all documents where the name field ends with 'doe', case-insensitively.
6. Let us see filtering first in the mongodb itself
   a. Create few more products under different categories



```
{categories:"electronics"}
```

7. Add products with diff stock quantities and apply a filter with stock quantity less than 10
   {"stock":{$lt : 10}}

8. Let us implement the category filter
   a. {categories:"electronics"} // first pass categories without the quotes
   b. POSTMAN - localhost:3000/api/products?filter={"categories":"electronics"}
   c. Comment the limit and skip related code

```
const getProductsHandler = async function(req, res) {
 console.log("getting products")
 /**
  * filter the product -> as per the query params
  * sorting the products -> as per the query params
  * select the fields -> as per the query params
  * pagination -> as per the query params
  */
 try {
   const sortQuery = req.query.sort;
   const selectQuery = req.query.select;
   const filterQuery = req.query.filter;
   // console.log(sortParams, selectParams);
   // sort logic
   let queryResPromise = Product.find();
   if (sortQuery) {
     const [sortParam, order] = sortQuery.split(" ");
     if (order === "asc") {
       queryResPromise = queryResPromise.sort(sortParam);
     } else {
       queryResPromise =
queryResPromise.sort(`-${sortParam}`);
     }
   }
   if(selectQuery){
       queryResPromise =
queryResPromise.select(selectQuery)
   }
   if(filterQuery){
     console.log("filterQuery beofre",filterQuery)
       // const filterObj = JSON.parse(filterQuery);
       queryResPromise = queryResPromise.find(filterQuery)
       // queryResPromise =
queryResPromise.find(JSON.parse(filterQuery))
   }
   /**
    * pagination logic will be implemented using limit and
skip
    * limit -> number of documents to be returned
    * skip -> number of documents to be skipped
    */
   const page = req.query.page  || 1;
   const limit = req.query.limit || 1;
   const skip = (page - 1) * limit;
   console.log("skip",skip)
   // queryResPromise =
queryResPromise.skip(skip).limit(limit);


   const result = await queryResPromise;

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

d. Impo pont to note is that without the JSON.parse for the filter, it will give an error
e. Parameter filters are string and we need to convert them to json objects

9. Let us try adding a filter where stock quantity is less than 10

a. In mongodb, add this filter - {stock:{$lte:10}}
b. Now pass this filter through postman and note that $ signs are not parsed thru JSON.parse
c. POSTMAN - localhost:3000/api/products?filter={"stock":{$lte:10}}
d. How we pass such params is that we dont pass the dollar sign as part of query params
e. And since we need the dollar sign to pass it to mongo db, we replace gt with $gt lt with $lt etc
f. The other issue is When passing JSON in a URL, it needs to be URL-encoded. Special characters in JSON, like {, }, and :, can cause issues if not encoded. Your JSON string in the URL should be encoded.
g. What is URL Encoding?
    i. URL encoding, also known as percent-encoding, is a method used to encode information in a Uniform Resource Identifier (URI) under certain circumstances.
    ii. It involves replacing unallowable characters with a % symbol followed by two hexadecimal digits that represent the ASCII code of the character.
    iii. A URL like http://example.com/user name would break because spaces are not allowed in URLs.
        1. Encoded: It should be http://example.com/user%20name.
    iv. Example: Passing JSON in a URL like http://example.com/api?filter={"name":"John Doe"}

would break because of the special characters in JSON ({, }, ", etc.).

1. Encoded: It should be something like http://example.com/api?filter=%7B%22name%22%3A%22John%20Doe%22%7D.

h.

i. Open this application - https://www.urlencoder.org/

j. Pass this filter query - {"stock":{"lte":10}}

k. URL after encoding -

localhost:3000/api/products?filter=%7B%22stock%22%3A%7B%22lte%22%3A10%7D%7D

```
if(filterQuery){
     console.log("filterQuery beofre",filterQuery)
     // console.log(encodeURI(filterQuery))
     // Parse the filterQuery into an object
     const filterObj = JSON.parse(filterQuery);
     console.log("filterObj",filterObj)

       // const filterObj = JSON.parse(filterQuery);
       // queryResPromise =
queryResPromise.find(filterQuery)
       // handling $gt, $gte, $lt, $lte
       // const filterObj = JSON.parse(filterQuery);
       // replacing lt with $lt and so on
       const filterObjStr =
JSON.stringify(filterObj).replace(
         /\b(gt|gte|lt|lte)\b/g,
         (match) => `$${match}`
       );
       console.log("filterObjStr",filterObjStr)
       // converting back to object
       const filterObjFinal = JSON.parse(filterObjStr);
```

```
        console.log("filterObjFinal",filterObjFinal)
        queryResPromise =
queryResPromise.find(filterObjFinal)
    }
```

l. So, while it might seem a bit roundabout, this method of parsing, modifying, and then parsing back is actually a safer and more accurate way to manipulate JSON data for database queries in JavaScript.
m. Format Validation: Parsing into an object first can act as a validation step, ensuring that the filterQuery is a valid JSON.
n. Targeted Replacement: By converting to an object and then back to a string, it ensures that the replacements occur specifically in the structure of a JSON object.
o. Directly replacing on the query params string can be a more efficient approach, especially when you're sure about the structure and content of the query string. However, if you need to ensure the structure's integrity or if the query string might contain complex or nested JSON, the parse-stringify-replace-parse approach can be safer

Aliasing

1. In the context of a Node.js Express API project, "aliasing" generally refers to a technique used to simplify or give more meaningful names to routes or parameters in your API.

2. This can make the routes easier to understand and use, both for developers working on the API and for clients consuming it.
3. Aliasing is not a built-in feature of Express but is a design pattern that can be implemented in various ways depending on the requirements.
4. For eg: This might involve creating simpler or more descriptive paths for complex or commonly used routes.
5. If For example, you might have a route like /api/v1/products?sort=price&order=asc&limit=10. You could create an alias like /api/v1/products/cheapest
6. Let us create something like a bigBillionDay route
7. What can it contain
    a. Products that are less in stock (for faster clearance)
    b. Ratings greater than 4.6
    c. Maybe show 10 products
    d. Designing for the big billion sale say page
8. Think about it
    a. In the previous exercise, we created an endpoint where we are passing filter query and then a handler to process this,
    b. Do we need to redo the same thing again or is there a way in which we can create a new route handler but still re-use the previous getAllProducts handler
    c. One more imp point to note is that all the middlewares share the same req, response object. So if you modify in one , the changes are reflected on the other one as well
    d. In the product router , create a route for bigBillionDay

```
productRouter.get("/", getProductsHandler);
productRouter.post("/", checkInput, createProduct);
productRouter.get("/:id", getProductById);
productRouter.delete("/:id", deleteProductById);
productRouter.get('/bigBillionDay',getBigBillionProducts)
```

e. Let us create getBigBillionProducts function
f. Initially we will see with the rating and limit as well

```
const getBigBillionProducts = async function(req, res,
next) {
/**
 * create a query with stock less than 30
 * rating grater than eq 4.6
 * limit query to 10
 *   */
req.query.filter = JSON.stringify({
  stock:{lt:30}
})
next()
}
```

g. Now imp point to reflect here is that in the getAllProduct handler, we are handling for the filter query.
h. If we chain this to the getAllProducts then the filter set here can be used to get the filtered data

```
productRouter.get("/", getProductsHandler);
productRouter.post("/", checkInput, createProduct);
productRouter.get("/:id", getProductById);
productRouter.delete("/:id", deleteProductById);
productRouter.get('/bigBillionDay',getBigBillionProducts,
getProductsHandler)
```

i. This should give an error because as per the route order produts/:id is matched before this route and it is trying to find a product with id - bigBillionDay

j. Correct order

```
productRouter.get("/", getProductsHandler);
productRouter.post("/", checkInput, createProduct);
productRouter.get('/bigBillionDay',getBigBillionProducts,
getProductsHandler)
productRouter.get("/:id", getProductById);
productRouter.delete("/:id", deleteProductById);
```

Central Error handler

1. Instead of catch block which returns 500 everywhere in the code, we can have a centralised way of handling this
2. In app.js, after the routes add below

```
/** routes */
app.use("/api/user", userRouter);
app.use("/api/product", productRouter);
app.use("/search",(req,res)=>{
 res.status(200).json({
   message:"search",
   data:req.query
 });
})
```

```
// Central error handler
app.use((err, req, res, next) => {
 const statusCode = err.statusCode || 500;
 const message = err.message || 'Internal Server Error';
 res.status(statusCode).json({
     status: statusCode,
     message: message,
 });
});
```

3. In the crudfactory for getElementById, update in the catch block

```
const getElementByIdFactory = (elementModel) =>
 async function (req, res,next) {
   try {
     const { id } = req.params;
     const data = await elementModel.findById(id);
     if (!data) {
       throw new Error("No data found");
     } else {
       res.status(200).json({
         status: 200,
         message: "Data found",
         data: data,
       });
     }
   } catch (err) {
     // res.status(500).json({
     //   status: 500,
     //   message: err.message,
     // });
     console.log("in the error handler")
```

```
    next(err)
  }
};
```

4. Pass a wrong id in the postman for get user by id and see that the error handler works
5. The next function is a part of Express's routing mechanism. It essentially tells Express to continue with the next middleware in line. When you call next() with no argument, Express moves to the next non-error handling middleware.
6. However, if you call next with an argument (like next(err)), Express treats it as an error, and it starts looking for error handling middleware.
7. Error handling middleware in Express are defined just like regular middleware but with four arguments instead of three: (err, req, res, next). When an error is passed to next, Express skips all non-error handling middleware and forwards the error to the next error handling middleware.