MERN-2: Session 4

In this session

- Middleware
- Globally & Selectively Applying Middleware
- Storing data in MongoDB
- Using Mongoose ODM

Recap

- We started validating our routes using Joi
- Also added some level of protection using a key
- And Environment variables to hide those secrets
- But still we need to apply it to each of the routes throughout .

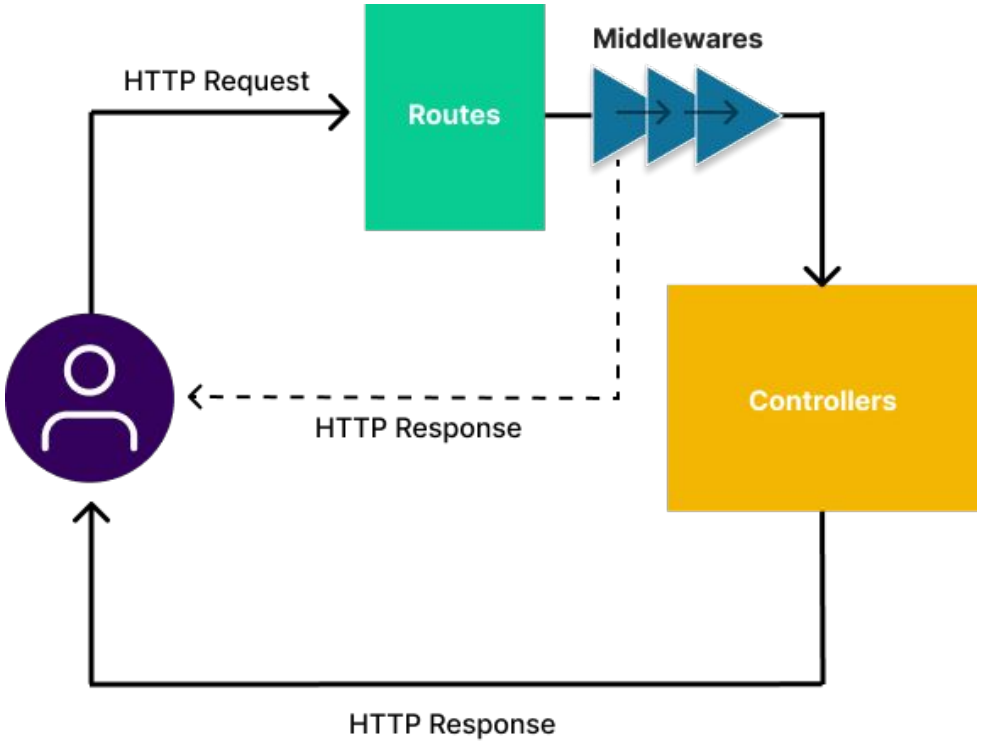
We still need to apply authorization for each and every route of our application

Let’s see how we can do that

Middleware

Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle.

Current App Architecture



Adding a middleware

We already have the verifyAuth function which takes in a req.

We will modify it to act like a middleware

1. A middleware takes in both req and res as its arguments.

```
const verifyAuth = (req, res) => {
  const { authorization } = req.headers;
  if (!authorization) {
    return false;
  }
  if (authorization !== Password) {
    return false;
  }
  return true;
}
```

Adding a middleware

2. A middleware can send response back to the client. It need not return true/false

```
const verifyAuth = (req, res) => {
  const { authorization } = req.headers;
  if (!authorization) {
    return res.status(403).json({ message: "Unauthorized Request" });
  }

  if (authorization !== Password) {
    return res.status(403).json({ message: "Unauthorized Request" });
  }

  return true;
};
```

Adding a middleware

3. If everything goes well, the middleware passes the `req` and the `res` to the next middleware

   (like passing the baton to another player)

To pass and accept the request and the response object to the next middleware, each middleware accepts a callback named as `next()` as an argument

Adding a middleware

To pass and accept the request and the response object to the next middleware, each middleware accepts a callback named as `next()` as an argument

```
//The next callback is from the previous middleware.
const verifyAuth = (req, res, next) ⇒ {
  console.log("Password from Env =", Password);
  const { authorization } = req.headers;
  if (!authorization) {
    return res.status(403).json({ message: "Unauthorized Request" });
  }
  if (authorization ≠= Password) {
    return res.status(403).json({ message: "Unauthorized Request" });
  }
  next(); //passes the request to the next object in line
};
```

Using the middleware in our app

4. To use the middleware import it in our index.js and use it with the help of app.use()

It takes in middleware(s) as an argument and applies them in order

```
const { verifyAuth } = require("./middlewares/verifyAuth");

const app = express();

app.use(verifyAuth); //Called first
app.use("/currencies", currencyRoutes); //Applied after verifyAuth
app.use("/users", userRoutes) //Applied after currencies
```

Voila! Now all our request pass through the validation!

What else could be used as a middleware?

Validators!

Activity - Creating a validator middleware

- You already created a `getQueryErrors` function, we can now move it to a new folder `middlewares/validators`
- Convert this `getQueryErrors` function into a `validateSearchQuery` middleware.
    a. Middleware take in req, res, and next
    b. Can return a response to the client
    c. If everything goes well, they call the next middleware.

But we need to only use this middleware for the /users/search route

Let’s see how to selectively apply middleware

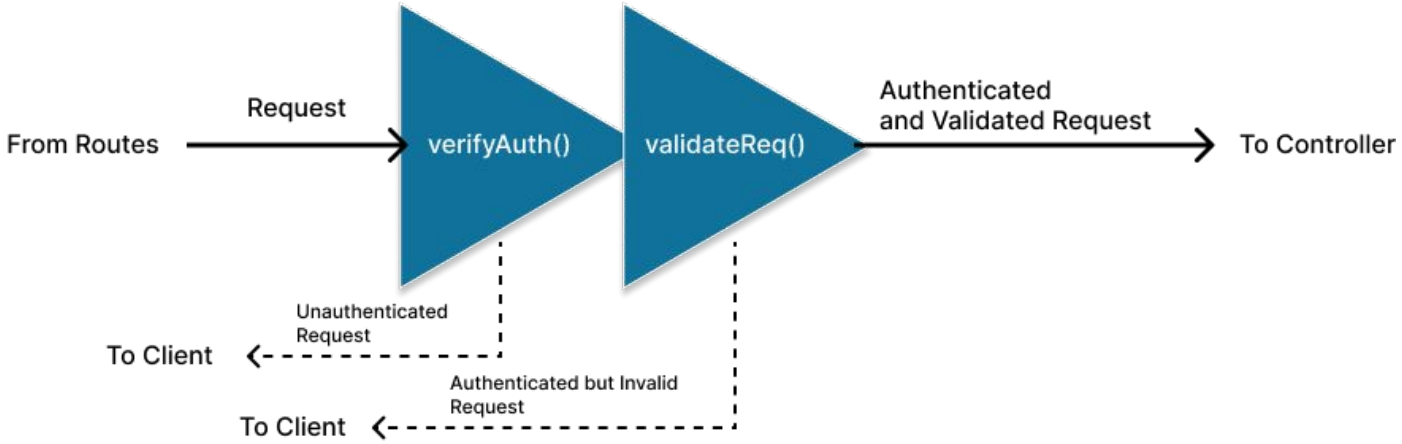
Selectively apply middleware

- We can selectively apply middleware to specific routes by adding it before the controller of that specific route

```
const {
  validateSearchQuery,
} = require("../middlewares/validators/users.validator");

router.get("/search", validateSearchQuery, searchUsersByQuery);
```

Middleware summary



Basically a middleware can

- Execute a piece of code (based on information of the request/response)
- Make changes to the request & response objects
- Return a response to the client (and end the request-response cycle)
- Call the next middleware.

Storing Data in a database!

Storing Stuff in DB

- Till this point we were using plain JSON files to mock our DBs
- This JSON is a format to transfer data between two machines
    - DB to server
    - Server to Client
- Although a JSON response does not represent the way data is stored in a DB
- Today there are two popular DB models available
    - **SQL Type DBs** - Examples: MySQL, PostgreSQL
    - **NoSQL DBs** - Examples: MongoDB, Firestore

SQL vs MongoDB

SQL	MongoDB
Needs to save structured data, with a rigid schema	No strict need to store structured data, flexible schema
Stores data in tables with rows and columns	Stores data in Documents (which resemble JSON format)

Understanding MongoDB

- There are a few concepts that apply to all NoSQL databases, but we will study them in the light of MongoDB
- Data is stored in collections and documents
- A document resembles a JSON Object that can hold data in key-value pairs
    - Example - A document (or object) to store user data, or blog data like comments, content etc.
- A group of documents is known as a collection.

Documents in MongoDB

- Documents are the fundamental units for storing data in a NoSQL DB
- Example - A user document can look something like
    - If you are familiar with SQL DB → A document is equivalent to a record in SQL table.

```
{
   "_id": "dj0jd29kak20jfj391jdjl103jfn",
   "username": "viveknigam3003",
   "roles": ["ADMIN", "USER"],
   "email": "vivek@resuminator.in"
}
```

Documents in MongoDB

- In MongoDB, each document has an auto generated **ObjectId**
- Which is basically a Hexadecimal value to uniquely identify the document inside a collection.
- This **ObjectId** is automatically attached to each new document added to the collection.

```
{
    "_id": ObjectId("507f1f77bcf86cd799439011"),
    "username": "viveknigam3003",
    "roles": ["ADMIN", "USER"],
    "email": "vivek@resuminator.in"
}
```

Format of Data in MongoDB

- The document resembles a JSON but is actually stored in an improved version of it - BSON
- BSON = Binary JSON, that supports more values types like Date, apart from plain String, Boolean, Number, and Array
- Stores Data in Binary Form
    - Read the complete differences between JSON vs BSON - https://www.mongodb.com/basics/bson

Activity - Using Mongo and storing documents

Let’s build our personal blog

```
> mongo
```

```
crio-user@vivek-nigam-crio-users:~/workspace/mern-2$ mongo
MongoDB shell version v3.6.3
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.3
Server has startup warnings:
2021-12-25T09:40:24.660+0530 I STORAGE  [initandlisten]
2021-12-25T09:40:24.660+0530 I STORAGE  [initandlisten] ** WARNING: Using the XFS filesystem is strongly recommended with the WiredTiger sto
rage engine
2021-12-25T09:40:24.660+0530 I STORAGE  [initandlisten] **          See http://dochub.mongodb.org/core/prodnotes-filesystem
2021-12-25T09:40:27.144+0530 I CONTROL  [initandlisten]
2021-12-25T09:40:27.144+0530 I CONTROL  [initandlisten] ** WARNING: Access control is not enabled for the database.
2021-12-25T09:40:27.144+0530 I CONTROL  [initandlisten] **          Read and write access to data and configuration is unrestricted.
2021-12-25T09:40:27.145+0530 I CONTROL  [initandlisten]
>
```

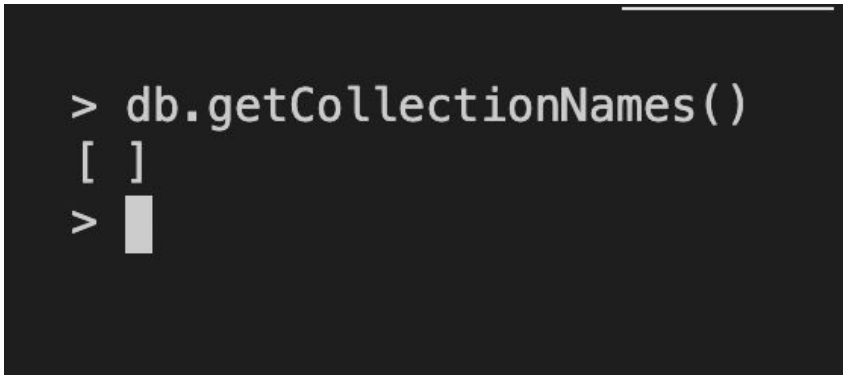
Activity - Using Mongo and storing documents

Use the database test

```
> use test
```

Check to see if it has any collections inside it?

```
> db.getCollectionNames()
```



Activity - Using Mongo and storing documents

Add a new collection named “blogs” if already doesn’t exist

```
> db.createCollection("blogs")
```

```
> db.createCollection("blogs")
{ "ok" : 1 }
> db.getCollectionNames()
[ "blogs" ]
>
```

Activity - Using Mongo and storing documents

Now we will add some documents to this collection

Our first document will have -

```
> db.blogs.insert({publishedAt: null, content: "", author: []})
```

field	type	default value
publishedAt	Date	null
content	string	""
author	Array<string>	[]

Activity - Using Mongo and storing documents

We will add another document

```
> db.blogs.insert({publishedAt: null, content: "", author: "", title: ""})
```

field	type	default value
publishedAt	Date	null
content	string	""
author	string	""
title	string	""

Activity - Using Mongo and storing documents

Try to list down the documents of the blogs collection

```
> db.blogs.find({}).pretty()
```

Notice that an _id is automatically attached to each document.

```
> db.blogs.find({}).pretty()
{
        "_id" : ObjectId("61c6eb127b17694d7be09da5"),
        "publishedAt" : null,
        "content" : "",
        "author" : [ ]
}
{
        "_id" : ObjectId("61c6eb597b17694d7be09da6"),
        "publishedAt" : null,
        "content" : "",
        "author" : "",
        "title" : ""
}
>
```

Do you notice something in the author field here?

The author field

- In one document its an **array**, but in the other document its a **string**.
- Imagine on the frontend we get this data of the the blogs collection
    - And we wish to display the authors using this logic

```
data.map(item ⇒ (
  <div key={item._id}>
    {item.author.map(author ⇒ <p key={author}>{author}</p>}
  </div>
)
```

- Can you identify what problem might arise?

5 mins break!

Storing Data in MongoDB with NodeJS

- Since MongoDB is schema-less, it means that there is **no fixed structure for our document** objects
- This can make our DB very inconsistent to store objects and query them later
- Moreover the data in MongoDB is stored in BSON and we need to convert it to JSON to make it usable.
- Here is where an Object Data Modeling library like Mongoose helps us.

Mongoose

- Mongoose is a preferred ODM library for MongoDB in Node JS
- It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.
- It will define a schema and provide us an interface to perform CRUD (create-read-update-delete) operations on MongoDB using Node JS

Using Mongoose

Installing Mongoose

- Mongoose is available as an npm package that we can install using

```
npm i mongoose
```

Connecting to DB

1. We need to define the DB URI to make a connection between our server and the DB

```
const DB_URI = "mongodb://127.0.0.1:27017";
```

2. Import mongoose to use a function connect that makes an async request to connect MongoDB on the passed URI

```
mongoose
  .connect(`${DB_URI}`)
  .then(() ⇒ console.log("Connected to DB at", DB_URI))
  .catch((e) ⇒ console.log("Failed to connect to DB", e));
```

3. Start the server. If you see the success message, then you are connected to MongoDB installed on your local system (workspace)

Note on specifying a DB

- By default, the local mongo instance saves data in the selected DB (i.e. test), but we can specify the DB specifically by adding the DB name to the URI

```
const DB_URI = "mongodb://127.0.0.1:27017/website"
```

- The above URI now points to the website database inside mongodb and all the operations we perform will be w.r.t. this website database.

Mongoose Schema

- In our previous blog example, we had four main fields - `title, author, content, publishedAt`
- We need to **define a schema** so that we never pollute our DB with unwanted data type values for keys in our DB
- **A Mongoose schema defines the structure of the document, default values, validators, etc.**
- Create a new folder `models` and create a new file `blogs.model.js`

```
//Inside models/blogs.model.js
const mongoose = require("mongoose")
```

Creating a schema

A schema defines document properties through an object where…

The key name corresponds to the property name in the collection.

```
const blogSchema = new mongoose.Schema({
  title: String, //Title is string
  authors: [String], //Authors is an array of strings
  content: String, //Content is string
  publishedAt: Date, //publishedAt is Date
});
```

This schema will also trigger an **internal validator for data types** when storing data to DB.

It may also explicitly typecast values when storing it in DB (we'll explore this later)

Mongoose Model

- A Mongoose model is a wrapper on the Mongoose schema that provides an interface to the database for creating, querying, updating, deleting records, etc.
- Create a model using the above schema to interact with the DB collection

```
const blogModel = mongoose.model("Blogs", blogSchema);
```

Export this model out of the models/blogs.model.js file

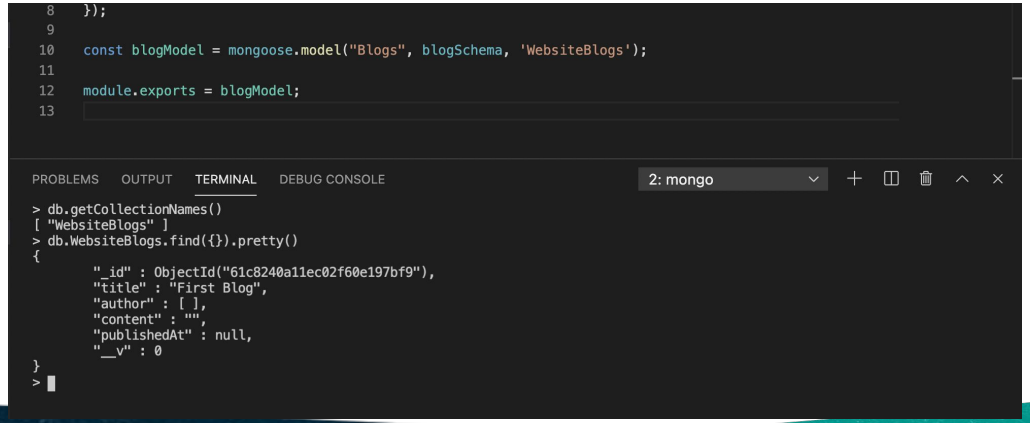
```
module.exports = blogModel;
```

Quick note about model and collection names

- The "`Blogs`" here **ALSO** becomes the **collection name** in Mongo automatically, when we perform any operations using this model.

- Should we need to separate the collection name in mongo from the model identifier, we can pass a third parameter to specify the collection name from which this model should read/write

```
const blogModel = mongoose.model("Blogs", blogSchema, 'WebsiteBlogs');
```



Interview questions

1. What is Express Middleware ?

- Express Middleware are functions that have access to request, and response object, and next function in application's request-response cycle.

2. How data is stored in MongoDB ?

- In MongoDB, data is stored as documents. These documents are stored in MongoDB in JSON-like format

3. What is Mongoose in NodeJS ?

Mongoose is a Node.js-based Object Document Modeling (ODM) library for MongoDB. Mongoose aims to solve is allowing developers to enforce a specific schema at the application layer.

Until next session 🍃

Thank you for joining in today, we’d love to hear your thoughts and feedback

Thank you