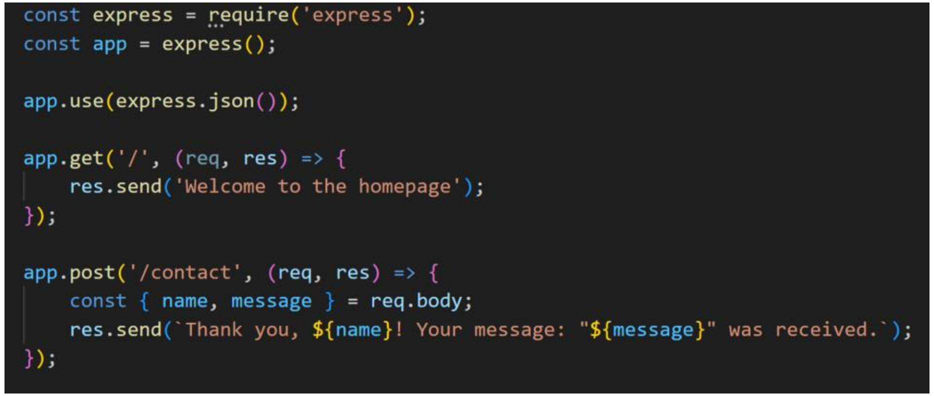
Web Development

CSE343
Lab 9

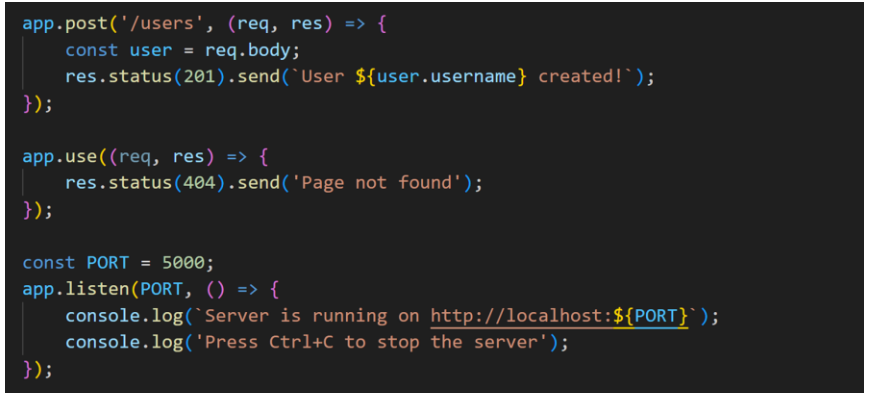
Express Continued

Express

- Last time we looked at what is express and how to use express framework to build our server.
- We also tried building a server and implementing some endpoints. Here is an example to refresh your mind a bit.



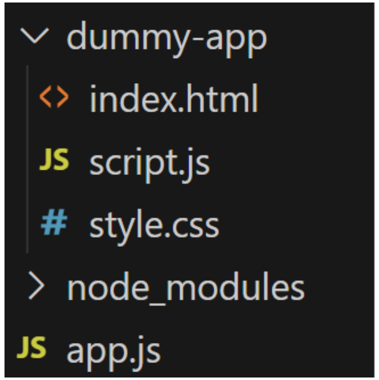
```javascript
const express = require('express');
const app = express();

app.use(express.json());

app.get('/', (req, res) => {
    res.send('Welcome to the homepage');
});

app.post('/contact', (req, res) => {
    const { name, message } = req.body;
    res.send(`Thank you, ${name}! Your message: "${message}" was received.`);
});
```



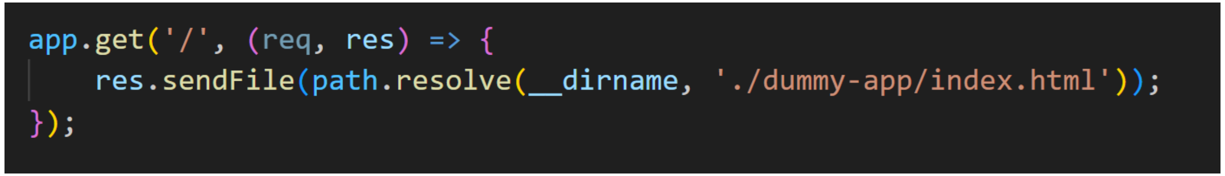
```javascript
app.post('/users', (req, res) => {
    const user = req.body;
    res.status(201).send(`User ${user.username} created!`);
});

app.use((req, res) => {
    res.status(404).send('Page not found');
});

const PORT = 5000;
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
    console.log('Press Ctrl+C to stop the server');
});
```

Sending back a File

- What if I want to implement an endpoint that returns actually an html file instead of returning some text. To test this, I have created a dummy app as follows:
- I also refactored the '/' endpoint to send the index.html in the response using the path module to resolve the absolute path



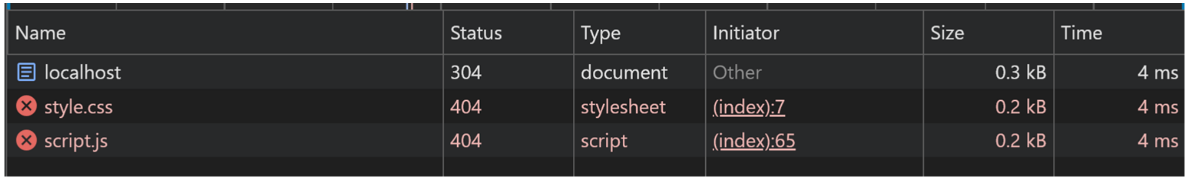
```
dummy-app
    index.html
    script.js
    style.css
node_modules
app.js
```



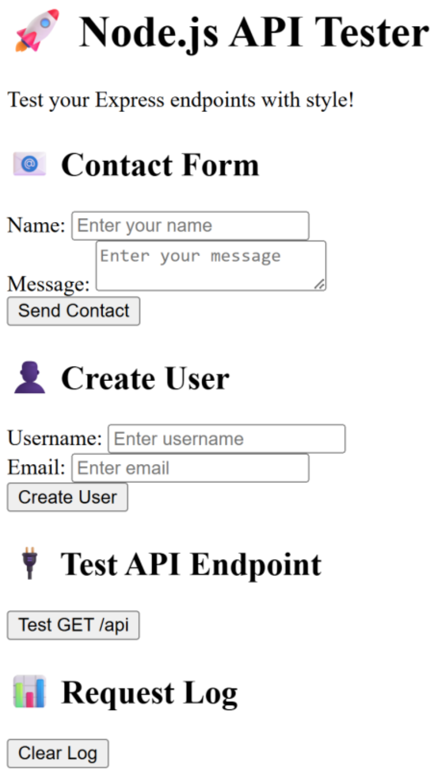
```
app.get('/', (req, res) => {
    res.sendFile(path.resolve(__dirname, './dummy-app/index.html'));
});
```

Sending back a File (cont.)

- The problem is that when we try to access this endpoint, the html will be sent back correctly, but the css and js files are not sent back so the webpage will be loaded as follows:
- If you take a look at the network tab in the browser you will find that the browser finds links to css and js files in the index.html file so it requests them from the backend



Name	Status	Type	Initiator	Size	Time
localhost	304	document	Other	0.3 kB	4 ms
style.css	404	stylesheet	(index):7	0.2 kB	4 ms
script.js	404	script	(index):65	0.2 kB	4 ms



🚀 Node.js API Tester

Test your Express endpoints with style!

📧 Contact Form

Name: Enter your name
Message: Enter your message
Send Contact

👤 Create User

Username: Enter username
Email: Enter email
Create User

🔌 Test API Endpoint

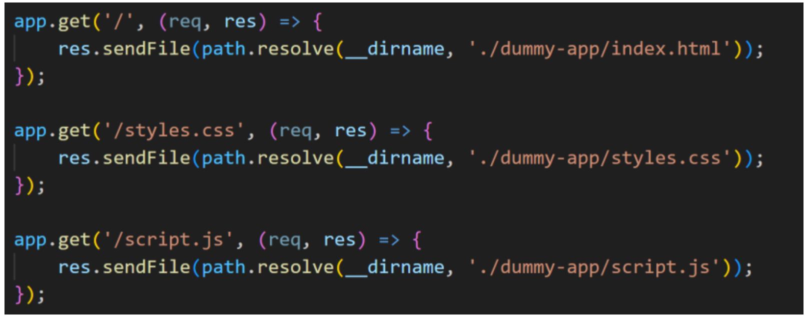
Test GET /api

📊 Request Log

Clear Log

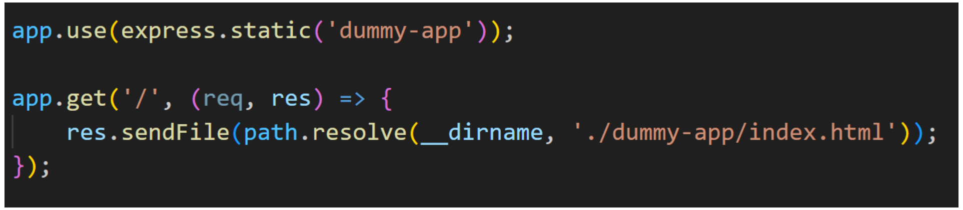
Sending back a File (cont.)

- Without express you will have to create separate endpoints for that as follows:



```
app.get('/', (req, res) => {
    res.sendFile(path.resolve(__dirname, './dummy-app/index.html'));
});

app.get('/styles.css', (req, res) => {
    res.sendFile(path.resolve(__dirname, './dummy-app/styles.css'));
});

app.get('/script.js', (req, res) => {
    res.sendFile(path.resolve(__dirname, './dummy-app/script.js'));
});
```

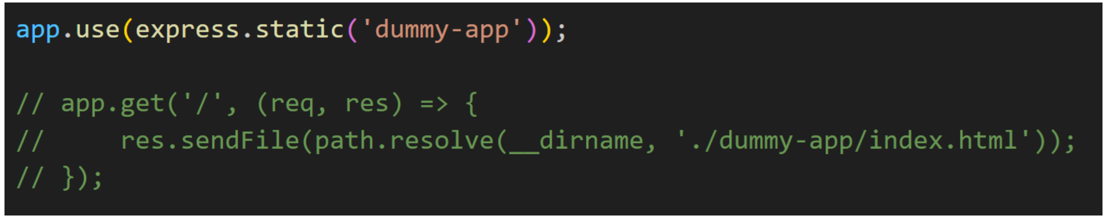
- With express, you can use the express.static function. This is middleware that serves static files 'dummy-app' is the folder name where your static files live. Express will look for files in this folder when a request comes in



```
app.use(express.static('dummy-app'));

app.get('/', (req, res) => {
    res.sendFile(path.resolve(__dirname, './dummy-app/index.html'));
});
```

Sending back a File (cont.)

- This will also work if we remove the endpoint that sends the file and keep it like:



```
app.use(express.static('dummy-app'));

// app.get('/', (req, res) => {
//     res.sendFile(path.resolve(__dirname, './dummy-app/index.html'));
// });
```

- When you use express.static(), you don't need res.sendFile() for static files anymore
- Express automatically serves all files from the public folder.
- Without express.static():
  - Browser requests: GET /
  - Express checks routes
  - Finds: app.get('/', ...)
  - Executes: res.sendFile('public/index.html')
  - Sends file to browser

Sending back a File (cont.)

- With express.static():
  - Browser requests: GET /
  - Express checks middleware FIRST
  - express.static() says: "I'll handle this!"
  - Looks in 'public' folder
  - Finds index.html
  - Automatically sends it to browser
  - Never reaches your routes!
- What counts as static files?
- Examples include: HTML files, CSS files, JavaScript files, Images (PNG, JPG, SVG, etc.), Fonts, PDFs
- You give Express a folder path, and it will automatically make all files inside that folder publicly accessible via HTTP.

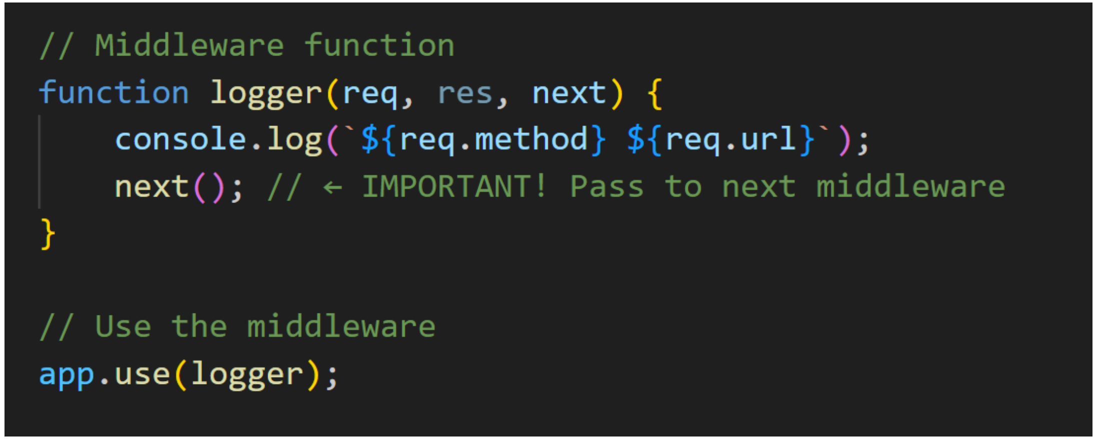
Middleware

What is Middleware?

- Middleware is a function that has access to:
  - req (request object)
  - res (response object)
  - next (function to pass control to the next middleware)
- Think of middleware as checkpoints or filters that every request passes through before reaching your routes.
- Each checkpoint:
  - Inspects you (reads req)
  - May modify your luggage (modifies req or res)
  - Decides if you can proceed (next()) or stops you (res.send())

Basic Middleware structure

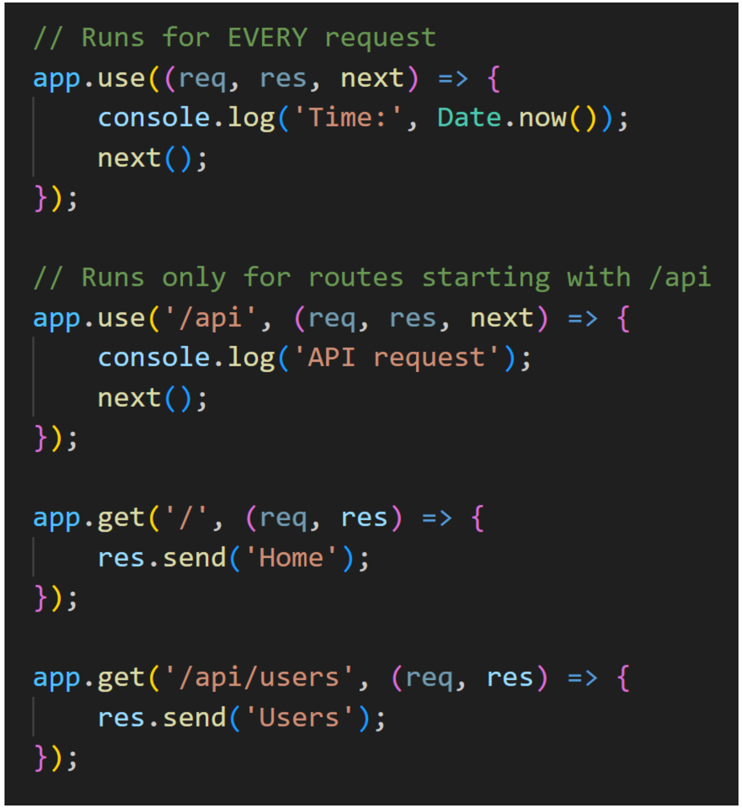
- What happens when you visit http://localhost:5000/:
    - Request arrives: GET /
    - logger middleware runs:
        - Logs: "GET /"
        - Calls next()
    - Route handler runs:
    - Sends: "Home Page"



```
// Middleware function
function logger(req, res, next) {
    console.log(`${req.method} ${req.url}`);
    next(); // ← IMPORTANT! Pass to next middleware
}

// Use the middleware
app.use(logger);
```

Types of Middleware

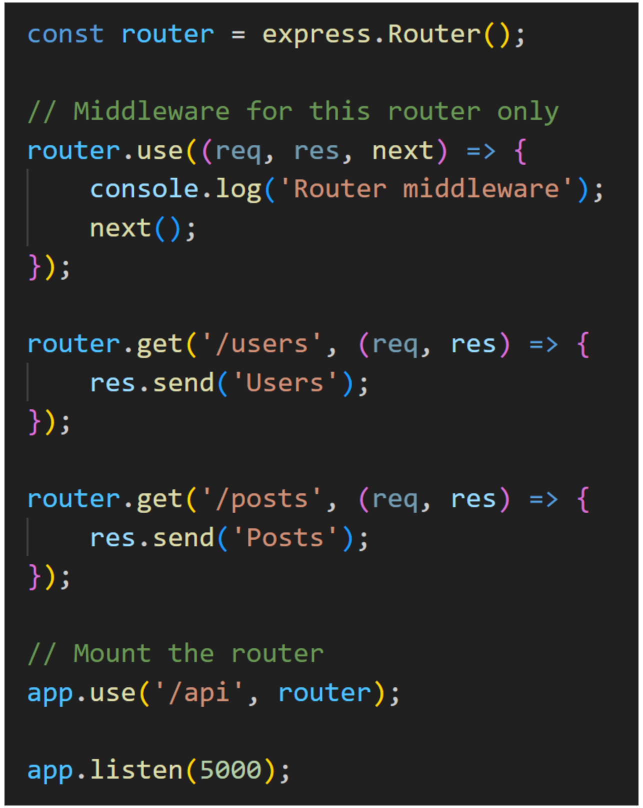
- There are several types of middleware:
  - Application-Level Middleware -> Applies to all routes (or specific paths). Middleware that works for the entire application or specific paths.
  - Bound to app using app.use()
  - Runs for all routes (or specific paths you choose)



```
// Runs for EVERY request
app.use((req, res, next) => {
    console.log('Time:', Date.now());
    next();
});

// Runs only for routes starting with /api
app.use('/api', (req, res, next) => {
    console.log('API request');
    next();
});

app.get('/', (req, res) => {
    res.send('Home');
});

app.get('/api/users', (req, res) => {
    res.send('Users');
});
```

Types of Middleware

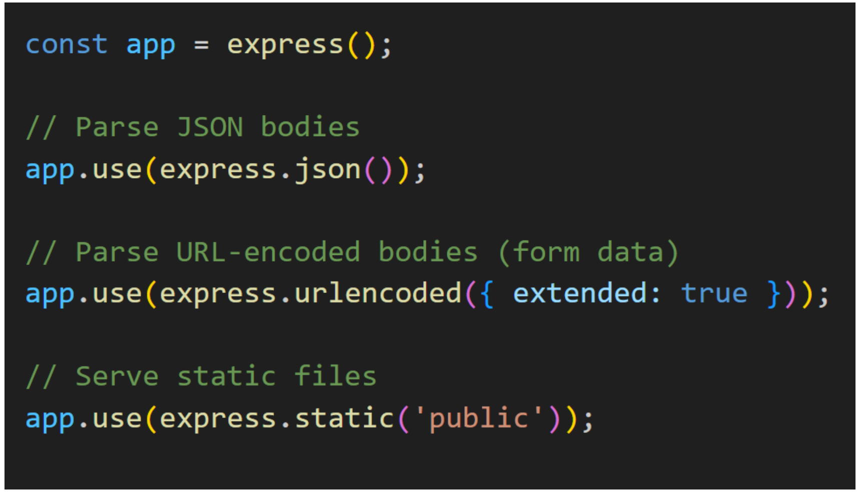
- Router-Level Middleware -> Middleware that works only for a specific group of routes (a router).
- Bound to router using router.use()
- Only affects routes in that specific router
- Helps organize large applications



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

// Middleware for this router only
router.use((req, res, next) => {
    console.log('Router middleware');
    next();
});

router.get('/users', (req, res) => {
    res.send('Users');
});

router.get('/posts', (req, res) => {
    res.send('Posts');
});

// Mount the router
app.use('/api', router);

app.listen(5000);
```

Types of Middleware

- Built-in Middleware -> Middleware that comes with Express (you don't need to install anything extra).



```
const app = express();

// Parse JSON bodies
app.use(express.json());

// Parse URL-encoded bodies (form data)
app.use(express.urlencoded({ extended: true }));

// Serve static files
app.use(express.static('public'));
```

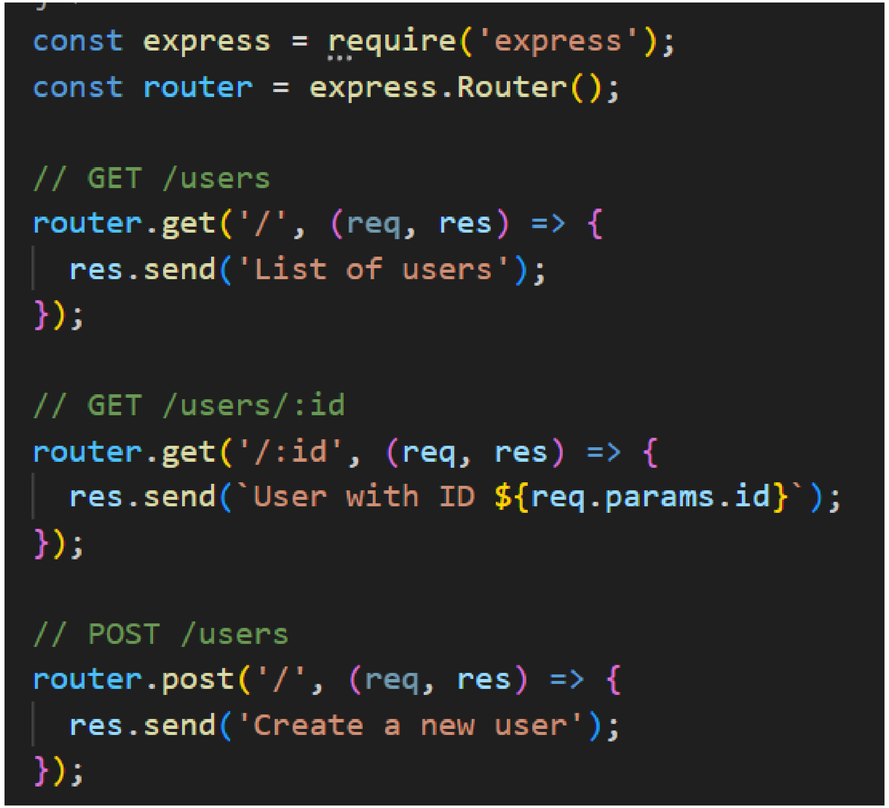
Routers

What is a Router?

- In Express, a router is a mini-application that can handle routes. It helps you organize your routes and separate them into modules instead of having all your routes in a single app.js or server.js file.
- Instead of having all the routes in one file, you can have a router for /users, a router for /posts
- Why Use Routers?
    - To Keep your code clean by splitting routes into separate files.

How create a router

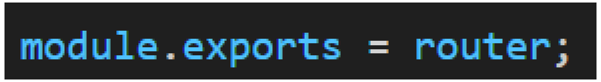
- We start by creating a separate file where you will put the specific route in.
- We do not type /users, /users/:id, the router will be known for the specific route as shown.
- We just create all the endpoints specific to this route



```
const express = require('express');
const router = express.Router();

// GET /users
router.get('/', (req, res) => {
  res.send('List of users');
});

// GET /users/:id
router.get('/:id', (req, res) => {
  res.send(`User with ID ${req.params.id}`);
});

// POST /users
router.post('/', (req, res) => {
  res.send('Create a new user');
});
```

Exporting router

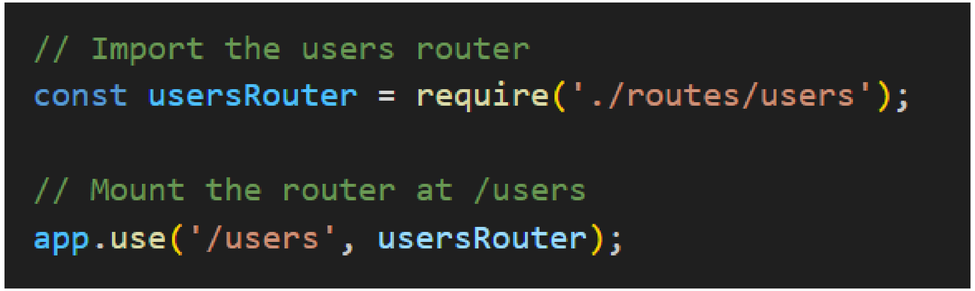
- We then export our router to be used in app.js



```
module.exports = router;
```

How to use router in app.js

- After exporting the router in the specific routes file, we import the router in app.js and mount it using the use middleware function as follows



```
// Import the users router
const usersRouter = require('./routes/users');

// Mount the router at /users
app.use('/users', usersRouter);
```

MongoDB

MongoDB

- Until now, we have implemented endpoints that fetch data from static arrays defined directly in our code. While this approach works well for learning and small-scale applications, it comes with several critical limitations that make it unsuitable for real-world production environments.
- The main problems with using static arrays are:
  - Data Volatility: All data stored in arrays exists only in the server's memory (RAM). This means that whenever the server restarts—whether due to a crash, deployment, or maintenance—all the data is permanently lost.
  - As the application grows and the number of records increases, storing everything in memory becomes impractical. Arrays must be loaded entirely into RAM, which consumes significant resources and slows down the application.
  - In modern applications, you often need multiple server instances running simultaneously (for load balancing or high availability). With static arrays, each server instance maintains its own separate copy of the data, leading to inconsistency. Changes made on one server aren't reflected on others.

MongoDB

- That's why we need a database—a dedicated system designed specifically for storing, managing, and retrieving data efficiently and reliably. For this course, we will be using MongoDB, a popular NoSQL database that works exceptionally well with Node.js and Express applications.

What is MongoDB?

- MongoDB is a document database which is often referred to as a non-relational database. This does not mean that relational data cannot be stored in document databases. It means that relational data is stored differently. A better way to refer to it is as a non-tabular database.
- MongoDB is a NoSQL, document-oriented database. Instead of tables and rows (like SQL), it stores data in JSON-like documents called BSON.

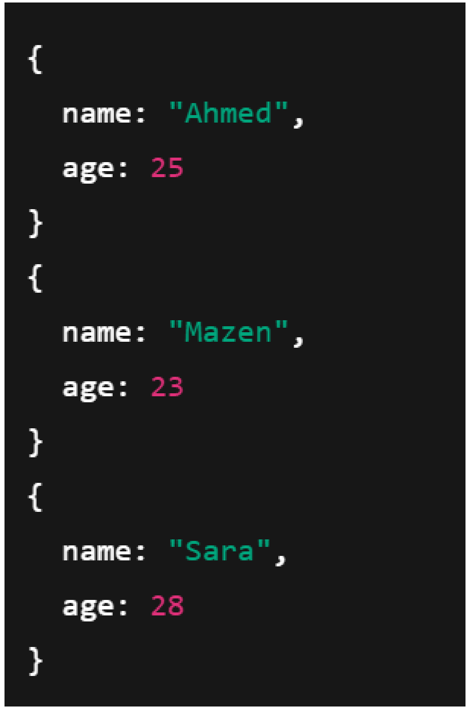
How MongoDB stores data

- Records in a MongoDB database are called documents, and the field values may include numbers, strings, booleans, arrays, or even nested documents. The basic data unit in MongoDB
- Looks like JSON
- The following shows an example of a document:

```
{
  title: "Post Title 1",
  body: "Body of post.",
  category: "News",
  likes: 1,
  tags: ["news", "events"],
  date: Date()
}
```

MongoDB Collections

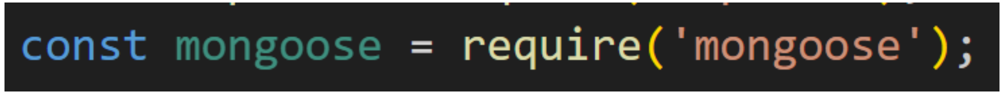
- A collection is where MongoDB stores multiple documents of the same type.
- For example:
  - A “users” collection → stores user documents
  - A “posts” collection → stores post documents
  - A “products” collection → stores product documents
- Each collection contains many documents arranged logically.



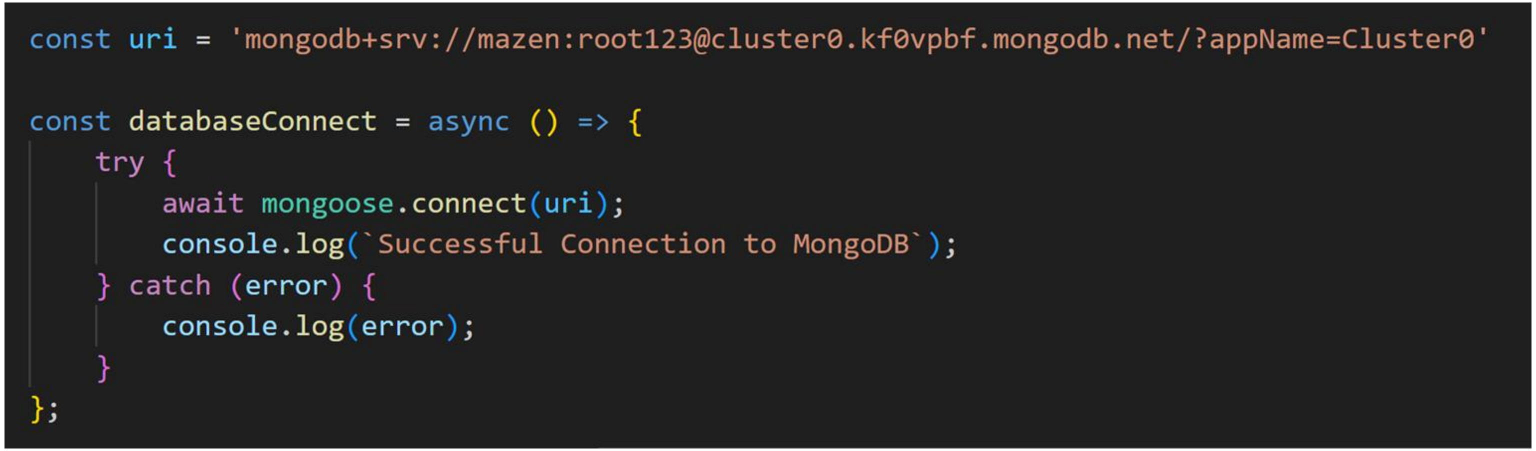
```
{
  name: "Ahmed",
  age: 25
}
{
  name: "Mazen",
  age: 23
}
{
  name: "Sara",
  age: 28
}
```

How to use MongoDB

- After setting up the database in mongodb cloud platform and copying and pasting the url, we need to install mongoose package to be able to connect and interact to our mongodb database using:
    - Npm install mongoose



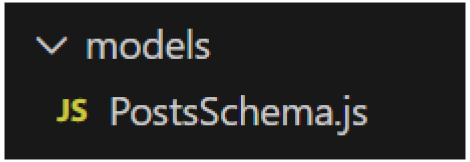
```
const mongoose = require('mongoose');
```

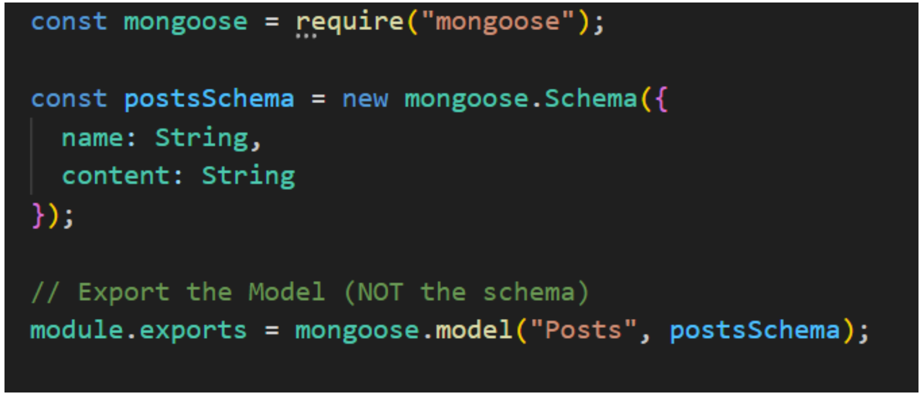


```
const uri = 'mongodb+srv://mazen:root123@cluster0.kf0vpbf.mongodb.net/?appName=Cluster0'

const databaseConnect = async () => {
    try {
        await mongoose.connect(uri);
        console.log(`Successful Connection to MongoDB`);
    } catch (error) {
        console.log(error);
    }
};
```

Creating a schema

- A schema is a template that describes what your data should look like.
- It defines:
  - What fields exist
  - What types those fields must be
  - Which fields are required
  - Default values
  - Validation rules
- Think of it like a form template every document must follow.

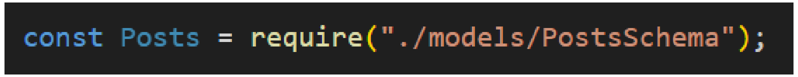




```
const mongoose = require("mongoose");

const postsSchema = new mongoose.Schema({
  name: String,
  content: String
});

// Export the Model (NOT the schema)
module.exports = mongoose.model("Posts", postsSchema);
```

Using the Schema in app

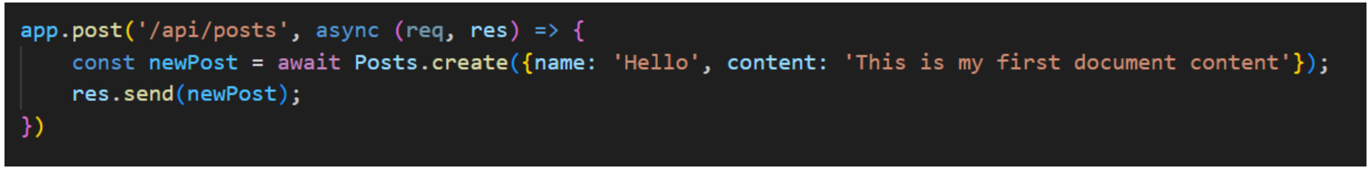
- Now, what we need to do is import the model we created from the schema.
- A model is created from a schema.
- A model is what actually:
  - Communicates with the database
  - Reads data
  - Writes data
  - Deletes data
  - Updates data
- A model represents a MongoDB collection.



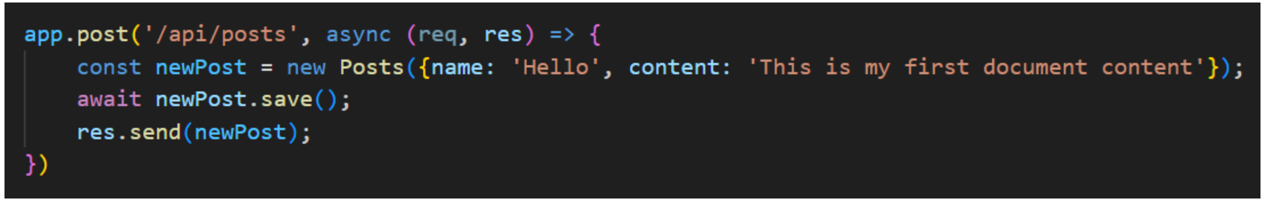
```
const Posts = require("./models/PostsSchema");
```

CREATE Operation

- Now, let’s see how we will use the model to perform crud operations
- CREATE
    - This is done using the Model.create(data) function. This function creates and inserts a new document.
    - You can also use the new Model(data) + .save() functions.
- The following example shows you how to do it



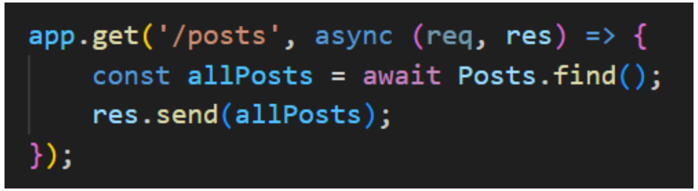
```
app.post('/api/posts', async (req, res) => {
    const newPost = await Posts.create({name: 'Hello', content: 'This is my first document content'});
    res.send(newPost);
})
```



```
app.post('/api/posts', async (req, res) => {
    const newPost = new Posts({name: 'Hello', content: 'This is my first document content'});
    await newPost.save();
    res.send(newPost);
})
```

READ Operation

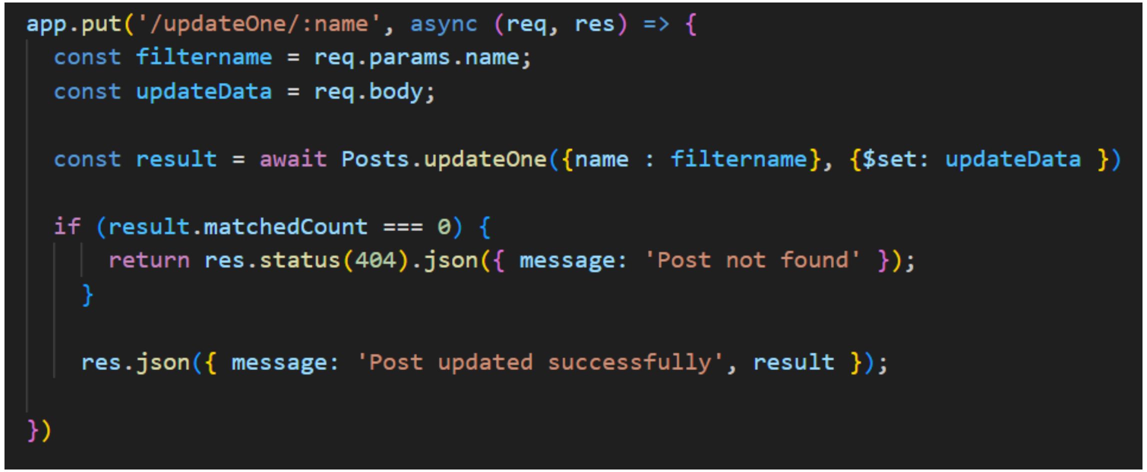
- READ
  - This is done using the Post.find(). This function gets all the documents.
  - You can also use the new Post.find(filter) function to find a specific document.
- The following example shows you how to do it



```
app.get('/posts', async (req, res) => {
    const allPosts = await Posts.find();
    res.send(allPosts);
});
```

UPDATE Operation

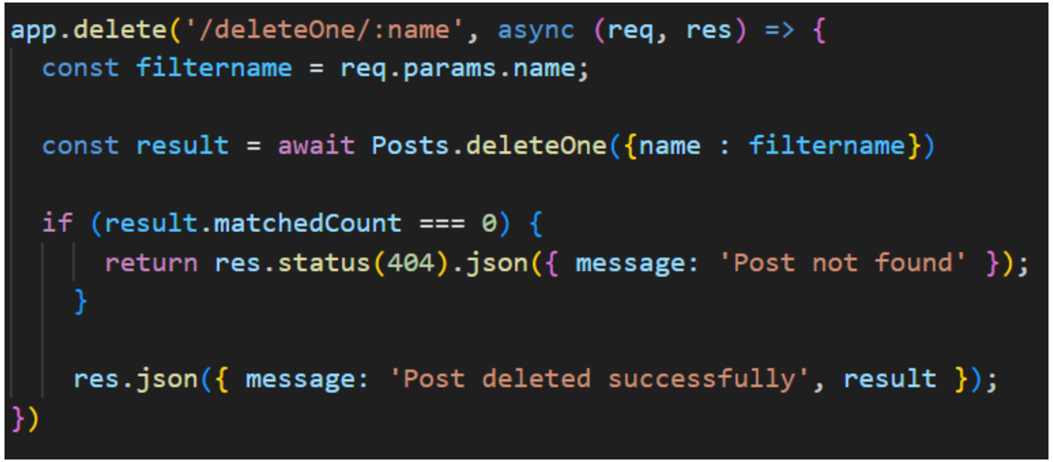
- UPDATE
  - This is done using the Model.updateOne(filter, update). This function updates the first document that matches the filter, but does NOT return the updated document.
  - You can also use the new Model.updateMany(filter, update) function Updates ALL documents matching the filter but does NOT return updated documents.
- The following example shows you how to do it



```
app.put('/updateOne/:name', async (req, res) => {
  const filtername = req.params.name;
  const updateData = req.body;

  const result = await Posts.updateOne({name : filtername}, {$set: updateData })

  if (result.matchedCount === 0) {
    return res.status(404).json({ message: 'Post not found' });
  }

  res.json({ message: 'Post updated successfully', result });

})
```

DELETE Operation

- DELETE
  - This is done using the Model.deleteOne(). This function Deletes the first matching document but does NOT return the deleted document.
  - You can also use the new Model.deleteMany() function to delete all documents matching the filter, but does NOT return deleted docs.
- The following example shows you how to do it



```
app.delete('/deleteOne/:name', async (req, res) => {
  const filtername = req.params.name;

  const result = await Posts.deleteOne({name : filtername})

  if (result.matchedCount === 0) {
    return res.status(404).json({ message: 'Post not found' });
  }

  res.json({ message: 'Post deleted successfully', result });
})
```

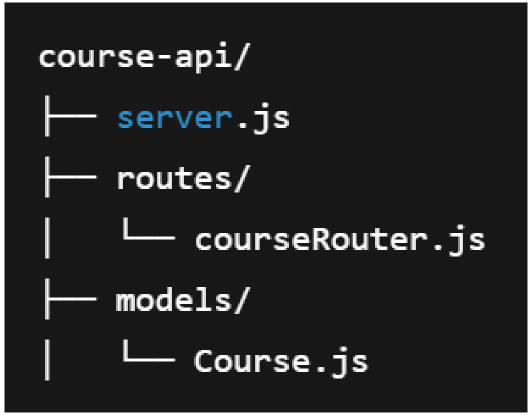
Lab Task

Lab Task (Building a Course Management API (Udemy-Style) Using Express.js, MongoDB & Routers

- Create a Database Schema called Courses
- Your task is to build the backend for managing courses.
    - Each course should include:
    - Title
    - Description
    - Instructor name
    - Price
    - Category (e.g., Web Development, Design, Marketing)
    - Number of enrolled students (default = 0)

Lab Task (Building a Course Management API (Udemy-Style) Using Express.js, MongoDB & Routers

- Students must:
  - Build an Express server.
  - Use Mongoose to interact with MongoDB.
  - Separate route logic using Express Router.
  - Implement CRUD operations for courses.
  - Test all endpoints with sample data.



```
course-api/
├── server.js
├── routes/
│   └── courseRouter.js
├── models/
│   └── Course.js
```