DAY 5

- *Database Connection*

- **Connect MongoDB with NodeJS**

- ***CREATE A FILE db.js IN THE ROOT FOLDER***

- The `db.js` file you've created is essentially responsible for establishing a connection between your Node.js application and your MongoDB database **using the Mongoose library**.

- In the Last Lecture, we saw that the mongoose is responsible for connection
- So let's import Mongoose Library

- *Connection Step by Step*

1. **Import Mongoose and Define the MongoDB URL:** In the `db.js` file, you first import the Mongoose library and define the URL to your MongoDB database. This URL typically follows the format `mongodb://<hostname>:<port>/<databaseName>`. In your code, you've set the URL to `'mongodb://localhost:27017/mydatabase'`, where `mydatabase` is the name of your MongoDB database.
2. **Set Up the MongoDB Connection:** Next, you call `mongoose.connect()` to establish a connection to the MongoDB database using the URL and some configuration options (`useNewUrlParser`, `useUnifiedTopology`, etc.). This step initializes the connection process but does not actually connect at this point.
3. **Access the Default Connection Object:** Mongoose maintains a default connection object representing the MongoDB connection. You retrieve this object using `mongoose.connection`, and you've stored it in the variable

`db`. This object is what you'll use to handle events and interact with the database.

4. **Define Event Listeners:** You define event listeners for the database connection using methods like `.on('connected', ...)`, `.on('error', ...)`, and `.on('disconnected', ...)`. These event listeners allow you to react to different states of the database connection.
5. **Start Listening for Events:** The code is set up to listen for events. When you call `mongoose.connect()`, Mongoose starts the connection process. If the connection is successful, the `'connected'` event is triggered, and you log a message indicating that you're connected to MongoDB. If there's an error during the connection process, the `'error'` event is triggered, and you log an error message. Similarly, the `'disconnected'` event can be useful for handling situations where the connection is lost.
6. **Export the Database Connection:** Finally, you export the `db` object, which represents the MongoDB connection, so that you can import and use it in other parts of your Node.js application.

To sum it up, the `db.js` file acts as a central module that manages the connection to your MongoDB database using Mongoose. It sets up the connection, handles connection events, and exports the connection object so that your Express.js server (or other parts of your application) can use it to interact with the database. **When your server runs, it typically requires or imports this `db.js` file to establish the database connection before handling HTTP requests.**

- *What are models or schema?*

- Models are like a blueprint of our database
- It's a representation of a specific collection in MongoDB. Like a Person
- Once you have defined a model, you can create, read, update, and delete documents in the corresponding MongoDB collection.
- Mongoose allows you to define a schema for your documents. A schema is like a blueprint that defines the structure and data types of your documents within a collection.

- Each Person's Detail (like chef, owner, manager, waiter)

```
{
      "name": "Alice",
      "age": 28,
      "work": "Chef",
      "mobile": "123-456-7890",
      "email": "alice@example.com",
      "address": "123 Main St, City",
      "salary": 60000
}
```

https://mongoosejs.com/docs/guide.html

- Parameters:
- Type, required, unique, etc

- *What is body-parser*

- `bodyParser` is a middleware library for Express.js.
- It is used to parse and extract the body of incoming HTTP requests.

- When a client (e.g., a web browser or a mobile app) sends data to a server, it typically includes that data in the body of an HTTP request.
- This data can be in various formats, such as JSON, form data, or URL-encoded data. `bodyParser` helps parse and extract this data from the request so that you can work with it in your Express.js application.
- `bodyParser` processes the request body before it reaches your route handlers, making the parsed data available in the `req.body` for further processing.

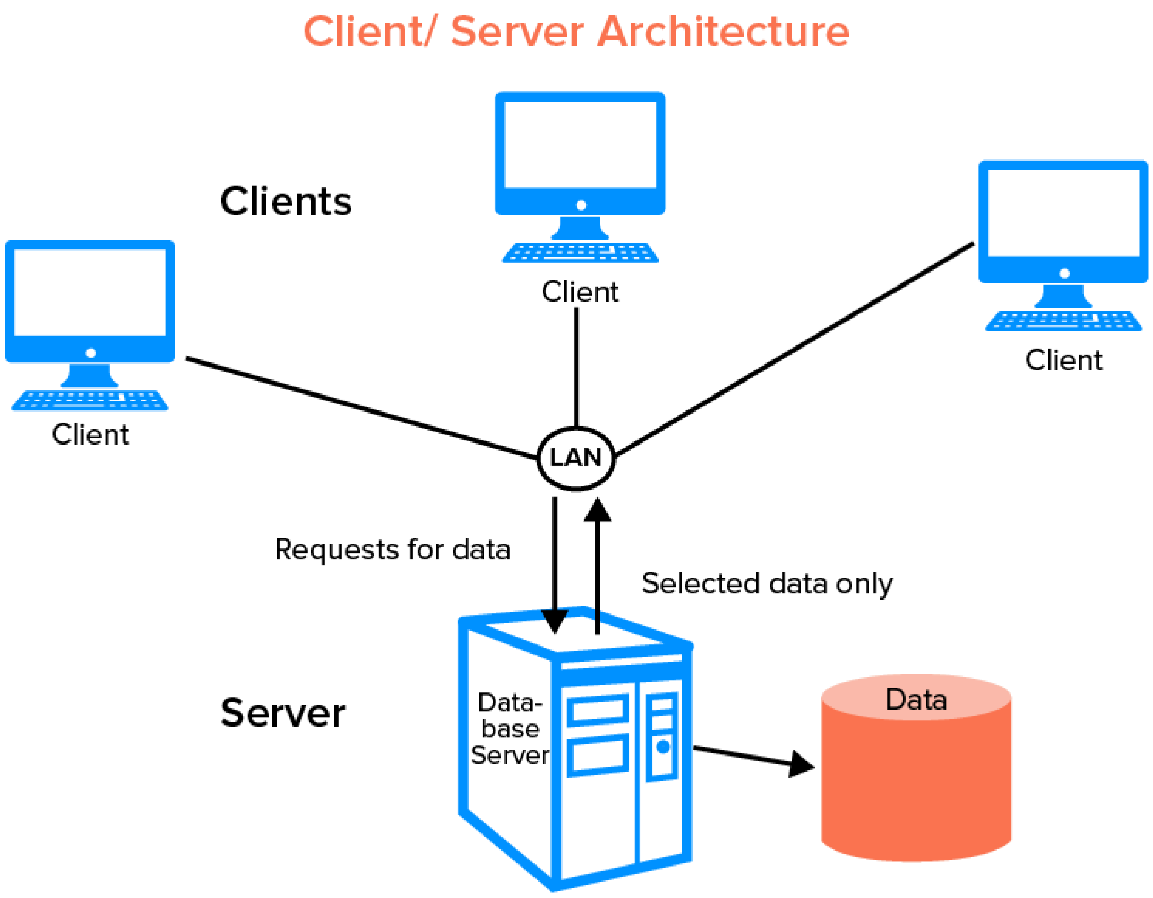
- `bodyParser.json()` automatically parses the JSON data from the request body and converts it into a JavaScript object, which is then stored in the `req.body`

- Express.js uses lots of middleware and to use middleware we use the app.use()

```
const bodyParser = require('body-parser');
app.use(bodyParser.json());
```

- *Send Data from Client to Server*

- we need an Endpoint where the client sends data and data needs to be saved in the database



- we need a method called POST
- Now code the POST method to add the person
- If we send the random values as well Mongoose will not save random values other than predefined schema

```
newPerson.save((error, savedPerson) => {
  if (error) {
    console.error('Error saving person:', error);
    res.status(500).json({ error: 'Internal server error' });
  } else {
    console.log('Data saved');
    res.status(201).json(savedPerson);
  }
});
```

- *Async and Await*

- Nowadays no one uses callback functions like, we used in the POST methods They look quite complex and also do not give us code readability.
- What actually callback does, callback is a function that is executed just after the execution of another main function, it means the callback will wait until its main function is not executed

- **Async and await** are features in JavaScript that make it easier to work with asynchronous code, such as network requests, file system operations, or database queries.
- Using try-and-catch block
- The `try` block contains the code for creating a new `Person` document and saving it to the database using `await newPerson.save()`.
- If an error occurs during any step, it is caught in the `catch` block, and an error response is sent with a 500 Internal Server Error status.

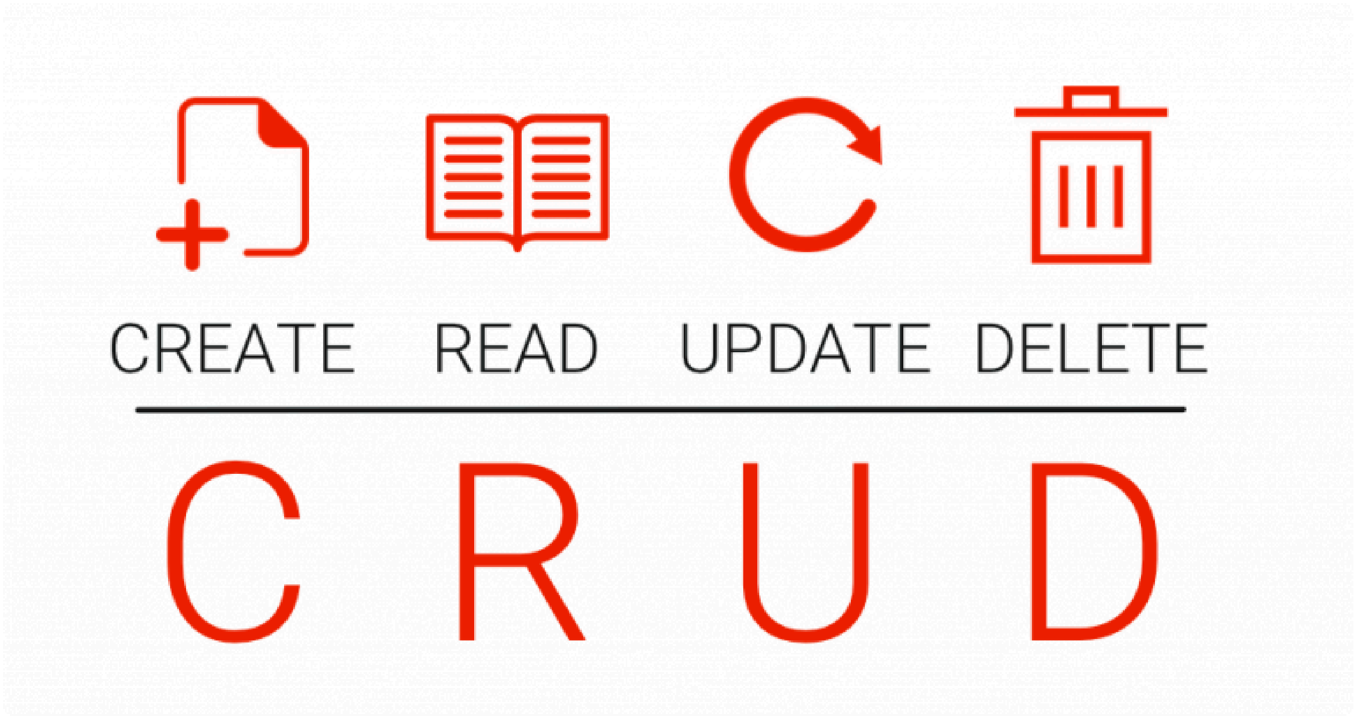
```
app.post('/person', async (req, res) => {
  try {
    const newPersonData = req.body;
    const newPerson = new Person(newPersonData);

    // Save the new person to the database using await
    const savedPerson = await newPerson.save();

    console.log('Saved person to database');
    res.status(201).json(savedPerson);
  } catch (error) {
    console.error('Error saving person:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

- **Async Function (`async`):**
  - An `async` function is a function that is designed to work with asynchronous operations. You declare a function as `async` by placing the `async` keyword before the function declaration.
  - The primary purpose of an `async` function is to allow you to use the `await` keyword inside it, which simplifies working with promises and asynchronous code.
  - Inside an `async` function, you can use `await` to pause the execution of the function until a promise is resolved. This makes the code appear more synchronous and easier to read.
- **Await (`await`):**
  - The `await` keyword is used inside an `async` function to wait for the resolution of a promise. It can only be used within an `async` function.
  - When `await` is used, the function pauses at that line until the promise is resolved or rejected. This allows you to write code that appears sequential, even though it's performing asynchronous tasks.

- If the promise is resolved, the result of the promise is returned. If the promise is rejected, it throws an error that can be caught using `try...catch`.

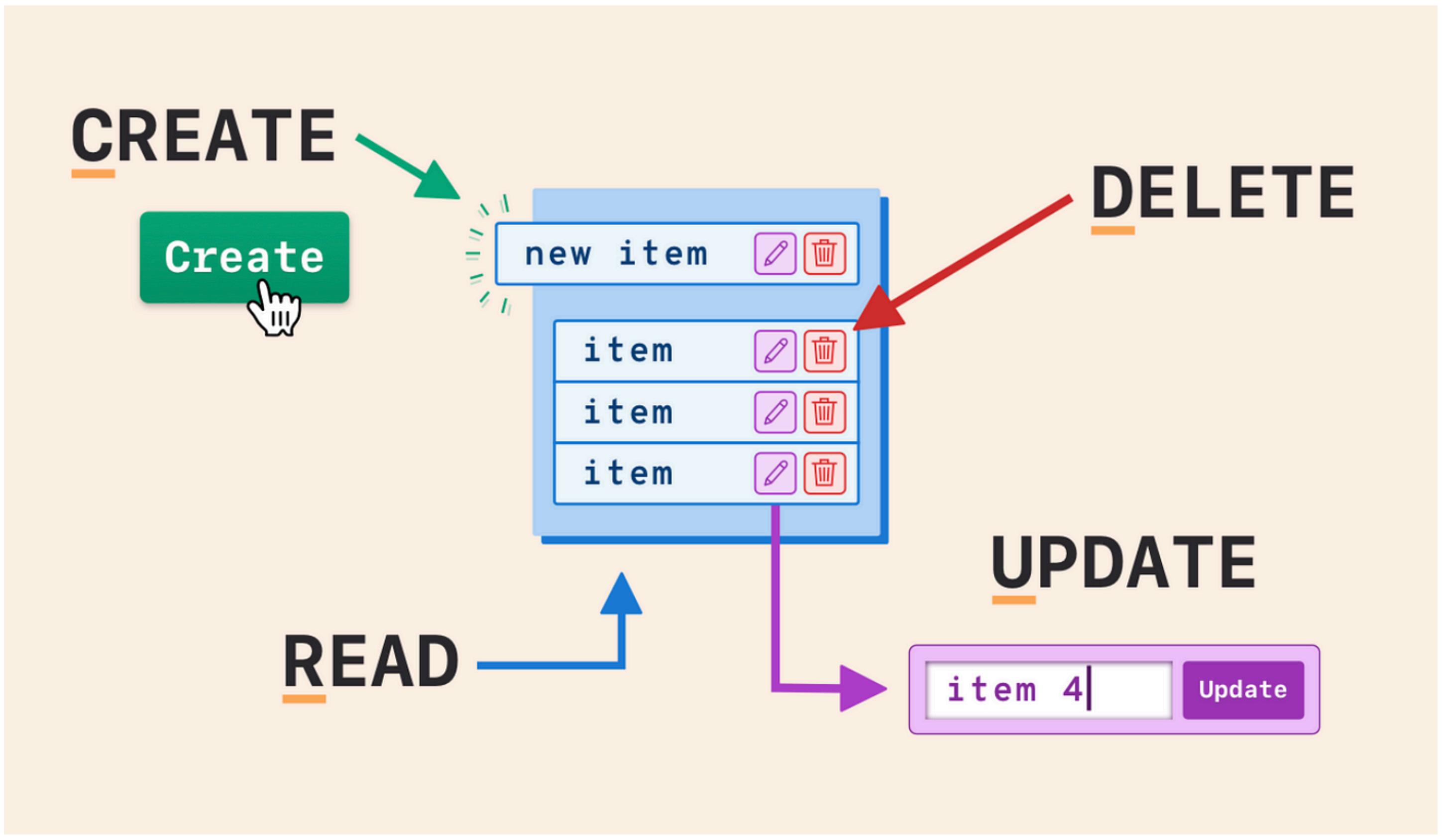
- *CRUD application*

- In any application at the core level, we are always handling the database



- Now we have seen that two methods **POST** and **GET**

		Database Operations		HTTP Methods
C	→	Create	→	POST
R	→	Read	→	GET
U	→	Update	→	PUT/PATCH
D	→	Delete	→	DELETE



- *GET Methods*

- Now Let's suppose the client wants data on all the persons
- So we need an endpoint for that **/person**

```
app.get('/person', async (req, res) => {
  try {
    // Use the Mongoose model to fetch all persons from the
database
    const persons = await Person.find();

    // Send the list of persons as a JSON response
    res.json(persons);
  } catch (error) {
    console.error('Error fetching persons:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

- *Create schema for Menu*

- Now create a model for the menu

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

const menuItemSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  price: {
    type: Number,
    required: true,
  },
  taste: {
    type: String,
    enum: ['Sweet', 'Spicy', 'Sour'],
  },
  is_drink: {
    type: Boolean,
    default: false,
  },
  ingredients: {
    type: [String],
    default: [],
  },
  num_sales: {
    type: Number,
    default: 0,
  }
});

const MenuItem = mongoose.model('MenuItem', menuItemSchema);

module.exports = MenuItem;
```