install nodejs and vs code

create folder and open it in vs code

in vs code terminal, type npm init

google-express node js and copy the following command to install express using vs terminal:

```
npm install express --save
```

now copy paste hello world example from express website into index.js file of vs code

run the program in vscode and view the output in web browser- localhost:3000

install thunder client extension of vs code and test it using localhost:3000

thunder client is like postman

---

create index.html file in vscode, we will use it in index.js

add path module in index.js

```
const path=require("path")
```

```
app.get('/me', (req, res) => {
    //res.send('Vikas!')
    res.sendFile(path.join(__dirname,'index.html'))
  })
```

---

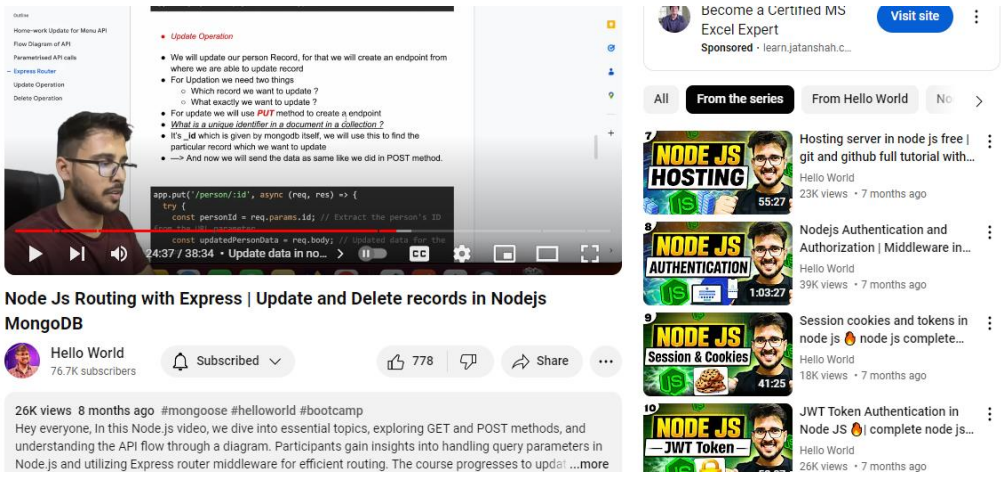
install nodemon using vscode terminal: npm install -g nodemon

install json formatter extension in google chrome for json data

Video Links:

Channel1: Code with Harry

Channel2: Hello World



https://youtu.be/0WnBDaD5flw?si=2-KXnlZF3PzEhpkF

//Important Java Script Code Examples

```
//filter function
const age=[23,45,67,12,34,14]
const result=age.filter(chkage)
function chkage(a)
{
    return a>20
}

console.log(result)

//4 ways to diefine functions in js

//1st way to define a function
function add1(a,b)
{
    return a+b;
}
console.log("add1 function call-" + add1(5,7))

//2nd way to define a function
var add2=function(a,b)
{
    return a+b;
}
console.log("add2 function call-" + add2(15,7))

//3rd way to define a function-call back function
var add3=(a,b) => {return a+b;}
console.log("add3 function call-" + add3(5,27))
//4th way to define a function-call back function
var add4=(a,b) => a+b;
console.log("add4 function call-" + add4(55,7))
```

Concept of call back function: when a function calls a different function

```
function callback(){
    console.log('now adding is successful complete');
}

const add = function(a, b, callback){
    var result = a+b;
    console.log('result: '+result);
    callback();
}

add(3,100989893, callback);
```

```
const add = function(a, b, prince){
    var result = a+b;
    console.log('result: '+result); // main function work complete
    prince();
}


add(2, 3, function(){
    console.log('add completed');
});

add(2, 3, () => console.log('add completed'));
```

- There are many built-in modules in Nodejs that we can use
- https://nodejs.org/api/

- **Learn about the 'fs' module**

It creates a file and writes the message inside

- **Learn about the 'os' module**
- → Learn os.userInfo()
- → Log username

```
var fs=require('fs')
var os=require('os')

var user=os.userInfo;
console.log(user);
console.log(user.username);
fs.appendFile("user.txt", 'Hi'+ user.username, ()=> {
    console.log("User Infomration written succssfully")
});
```

Import and export of files

data.js file:

```
console.log("Data File loaded");

var age=45;
function add(a,b){
    return a+b;
}
module.exports ={
    age,
    add
}
```

importfiles.js file

```
var user=require('./data.js');

var a=user.age;
var f=user.add(a,10)
console.log(f);
```

lodlash npm:

It provides various inbuilt functions for collections, arrays, manipulated objects, and other utility methods that we can use directly instead of writing them from scratch. It makes it easier to iterate over the arrays, strings as well as objects. Its modular methods make the creation of composite functions easier.

Lodash provides a set of functions to manipulate arrays, including methods for sorting, slicing, filtering, and transforming arrays. These methods help in efficiently handling array operations.

```
npm i lodash
```

```
const _ = require("lodash");
```

Examples:

_.isString('Hello') return true

_.isString(47) return false

JSON: JavaScript Object Notation

- Imagine you're sending a message to your friend, and you want to include information like your **name, age, and a list of your favorite hobbies**.
- You can't just send the message as is,
- you need to organize the information in a way that both you and your friend understand.
- JSON is a bit like this organized format for exchanging data between computers.

- JSON is a lightweight
- Structured and organized Data because
- in most contexts, JSON is represented as a string

```
{
  "name": "Alice",
  "age": 25,
  "hobbies": ["reading", "painting", "hiking"]
}
```

- **Inter Conversion JSON to an Object in Node.js:**

```
const objectToConvert = {name: "Alice", age: 25 };
const jsonStringified = JSON.stringify(objectToConvert); // Convert object to JSON stri
console.log(jsonStringified);
```

PROBLEMS OUTPUT COMMENTS TERMINAL zsh

```
prince@Princes-MacBook-Air node_tutorial % node server.js
John
prince@Princes-MacBook-Air node_tutorial %
```

JS server.js > ...

```
const objectToConvert = {
    name: "Alice",
    age: 25
};
const json = JSON.stringify(objectToConvert); // Convert object to JSON string
console.log(json);
```

PROBLEMS OUTPUT COMMENTS TERMINAL zsh

```
prince@Princes-MacBook-Air node_tutorial % node server.js
{"name":"Alice","age":25}
prince@Princes-MacBook-Air node_tutorial %
```

- *Create a server*

- Creating a server in NodeJs via **express** package

- Express.js is a popular framework for building **web applications** and **APIs** using Node.js.
- When you create an Express.js application, you're setting up the **foundation for handling incoming requests** and defining how your application **responds** to them.

- Now we are going to create a server == waiter
- Now the waiter has his own home?

In simple terms, "**localhost**" refers to your **own computer**. After creating a server in NodeJS, you can access your environment in 'localhost'

- Port Number?
- Let's suppose in a building – 100 rooms are there, for someone to reach he must know the room number right?

- *Methods to share data*

- Now, in the world of web development, we need to deal with data
- How data is sent and received between a client (like a web browser) and a server (built with Node.js)

- So there are lots of methods out there to **send or receive data** according to their needs.

- GET
- POST
- PATCH
- DELETE

- **Get**

- Imagine you want to read a book on a library shelf.
- You don't change anything
- you just want to get the information.

Similarly, the GET method is used to request data from the server.

For example, when you enter a website URL in your browser,
your browser sends a GET request to the server to fetch the web page.

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


app.get('/', function (req, res) {
    res.send('Welcome to my hotel... How i ca help you ?')
})

app.get('/chicken', (req, res)=>{
    res.send('sure sir, i would love to serve chicken')
})

app.get('/idli', (req, res)=>{
    res.send('welcome to south india and would love to serve my hotel...')
})

app.listen(3000)
```

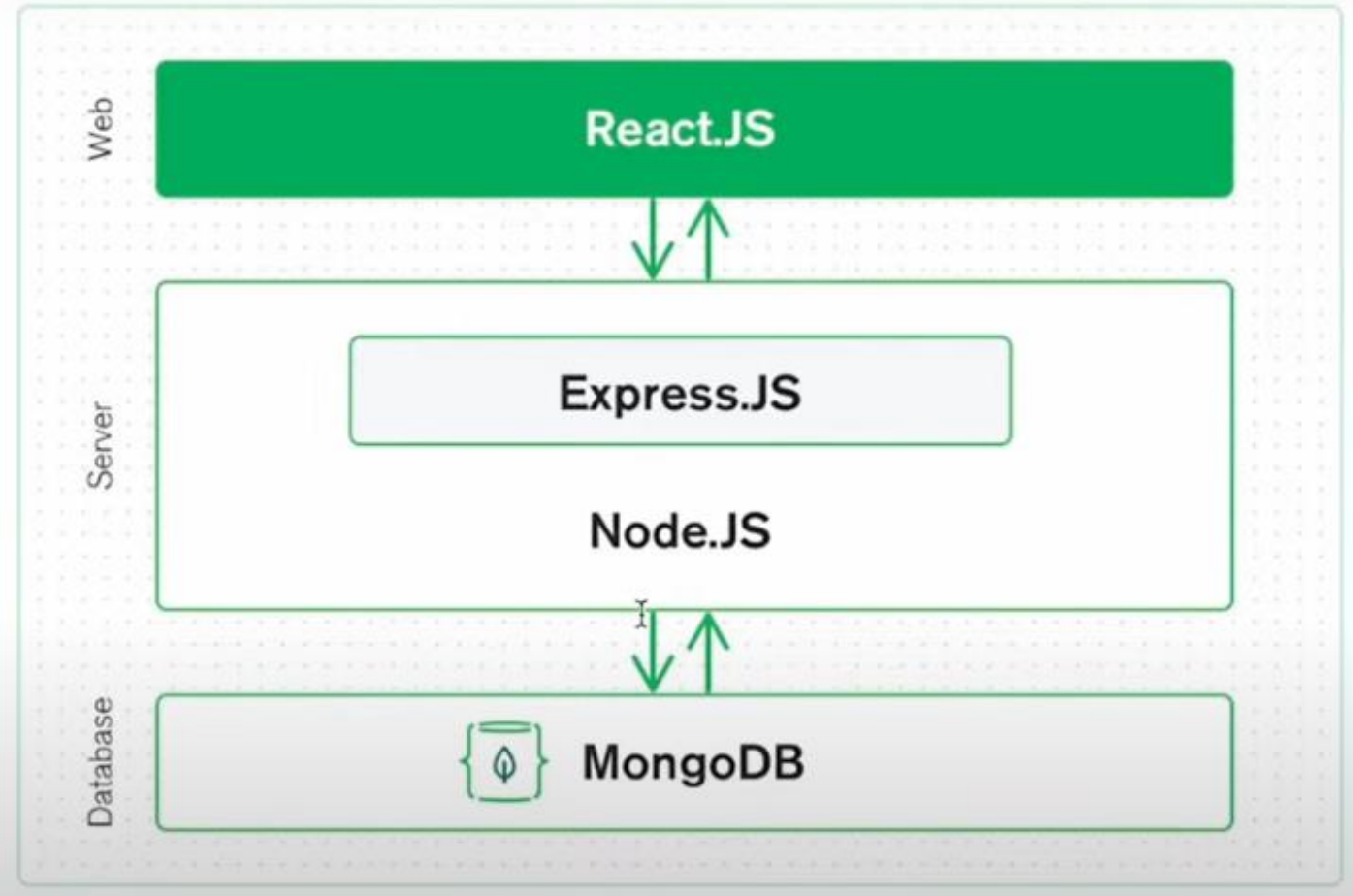
- *Database*

- **Web development = client + server + database**
- Ultimately, Let's suppose we are going to open Restuarant and there is lots of data around it,

  Number of chefs

  Each Person's Detail (like chef, owner, manager, waiter)
    - Name
    - Age
    - Work
    - Mobile number
    - Email
    - Address
    - salary

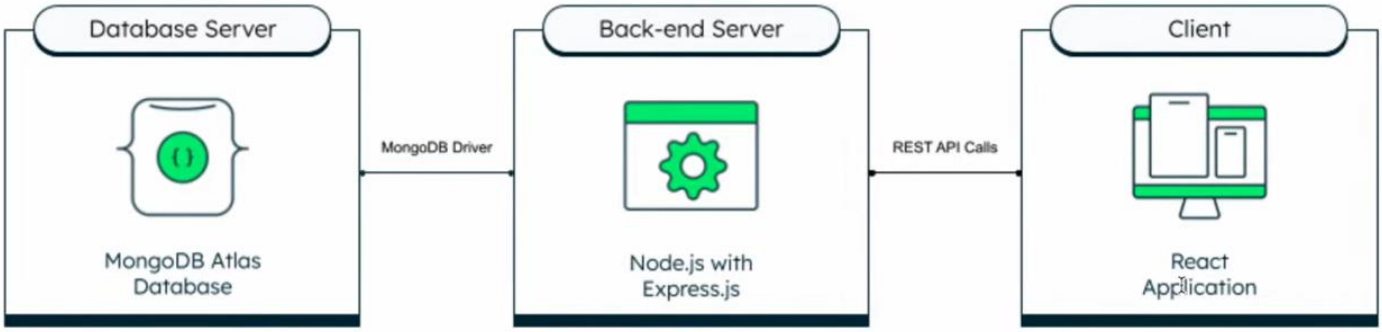
  Menu Details (like, drinks, Snacks, main course)
    - Name of dish
    - Price
    - Taste (like, sweet, sour, spicy)
    - Is_drink (boolean true, false)
    - Ingredients (array of data – ["wheat", "rice", "sugar"]
    - Number of sales (like 76)



- **Node.js Server and Database Server:**

- A database server is a specialized **computer program** or system that manages databases. It stores, retrieves, and manages data efficiently
- The database server stores your application's data. When your Node.js server needs data, it sends requests to the database server, which then retrieves and sends the requested data back to the Node.js server.

- Node.js server is responsible for handling HTTP requests from clients (like web browsers) and returning responses.
- It processes these requests, communicates with the database server, and sends data to clients.



1. Create a Database:

In SQL:

```
CREATE DATABASE mydb;
```

In MongoDB:

```
use mydb;
```

2. Create a Table (Collection in MongoDB):

In SQL:

```
CREATE TABLE users (
  id INT PRIMARY KEY,
  username VARCHAR(50),
  age INT
);
```

In MongoDB:

```
db.createCollection("users");
```

3. Insert Data:

```
INSERT INTO users (id, username, age)
VALUES (1, 'Alice', 25);
```

In MongoDB:

```
db.users.insertOne({ id: 1, username: 'Alice', age: 25 });
```

In SQL:

```
SELECT * FROM users WHERE age > 21;
```

In MongoDB:

```
db.users.find({ age: { $gt: 21} });
```

In SQL:

```
UPDATE users SET age = 22 WHERE username = 'Alice';
```

In MongoDB:

```
db.users.updateOne({ username: 'Alice' }, { $set: { age: 22 }
});
```

6. Delete Data:

In SQL:

```
DELETE FROM users WHERE id = 1;
```

In MongoDB:

```
db.users.deleteOne({ id: 1 });
```

- *MongoDB Compass GUI*

- There are lots of Tools in the market that help to visualize data like mongoDB compass, MongoDB Robo 3T
- mongodb://127.0.0.1:27017

- *Data Desing and Postman*

- Now in order to use the database we have to integrate MongoDB with nodejs
- Now, there should be a form built on ReactJS or HTML or CSS to add chef or person details
- Now currently we don't have a such frontend thing, so we are using **Postman** for this

- *Connect MongoDB with NodeJS*

- Now, To connect MongoDB with NodeJS we need a MongoDB driver (a set of programs)

- A MongoDB driver is essential when connecting Node.js with MongoDB because **it acts as a bridge** between your Node.js application and the MongoDB database.

- MongoDB speaks its own language (protocol) to interact with the database server.
- Node.js communicates in JavaScript.
- The driver translates the **JavaScript code from Node.js** into a **format that MongoDB can understand** and vice versa.
- The driver provides a set of functions and methods that make it easier to perform common database operations from your Node.js code.
- The driver helps you handle errors that might occur during database interactions. It provides error codes, descriptions, and other details to help you troubleshoot issues.

- The most popular driver is the official MongoDB Node.js driver, also known as the `mongodb` package.

```
npm install mongodb
```

- *Mongoose*

- Now but we are going to use Mongoose, rather than `mongodb`

- Mongoose is an ***Object Data Modeling (ODM)*** library for MongoDB and Node.js
- There are lots of reasons we prefer Mongoose rather than a native official driver
- Things are a lot easier here

(Relate Real life Examples with mobiles with earphones)

- Mongoose is like a translator between your Node.js code and MongoDB. It makes working with the database smoother and easier.

- With Mongoose, you can define how your data should look, like making a blueprint for your documents. It's like saying, "In our database, each person's information will have a name, age, and email." This makes sure your data stays organized.
- Mongoose helps you make sure the data you put into the database is correct. It's like having someone check if you've written your email address correctly before sending a message.
- Very easy to query from the database

—> But if you are using `mongodb` Native Driver

- You need to write a lot of detailed instructions to make sure everything works correctly.
- Without Mongoose, your code might get messy and harder to understand.
- Since you need to handle many details yourself, it can take longer to finish your project.

In a nutshell, using Mongoose makes working with MongoDB in Node.js much simpler and smoother. It gives you tools that handle complexities for you, so you can focus on building your application without getting bogged down in technical details.

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

npm install mongoose

- *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.**

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

Define the MongoDB connection URL
const mongoURL = 'mongodb://localhost:27017/hotels' // Replace 'mydatabase' with your database name

// Set up MongoDB connection
mongoose.connect(mongoURL, {
    useNewUrlParser: true,
    useUnifiedTopology: true
})
```

usernewurl and useunified variables are now depreciated

```
const mongoose=require('mongoose');
//Define  mongodb url connection
const mongoURL= 'mongodb://localhost:27017/mydb';
//setup monogdb connection
mongoose.connect(mongoURL);
//get the default connection
const db=mongoose.connection;

//define event listers to database connection
db.on('connected', ()=> {
    console.log("Connected");
});
db.on('error', ()=> {
    console.log("Error");
});
db.on('disconnected', ()=> {
    console.log("Dis-Connected");
});

//export the database connection
module.exports =db;
```

- *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.

```
{
    "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

```
models
  person.js
node_modules
db.js
package-lock.json
package.json
server.js
```

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

// Define the Person schema
const personSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    age:{
        type: Number
    },
```

```
person.js
node_modules
db.js
package-lock.json
package.json
server.js
```

```
    work:{
        type: String,
        enum: ['chef', 'waiter', 'manager'],
        required: true
    },
    mobile:{
        type: String,
        required: true
    },
    email:{
        type: String,
        required: true,
        unique: true
    },
    address:{
        type: String
    },
```

```
server.js
```

```
    salary:{
        type: Number,
        required: true
    }
});

// Create Person model
const Person = mongoose.model('Person', personSchema);
module.exports = Person;
```

HOw to use schema and model (defined above with mangoose) in node js

```
const express = require('express')
const app = express();
const db = require('./db');

const Person = require('./models/Person');

app.get('/', function (req, res) {
    res.send('Welcome to my hotel... How i can help you ?, we have list of menus')
})


app.listen(3000, ()=>{
    console.log('listening on port 3000');
})
```

- *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());
```

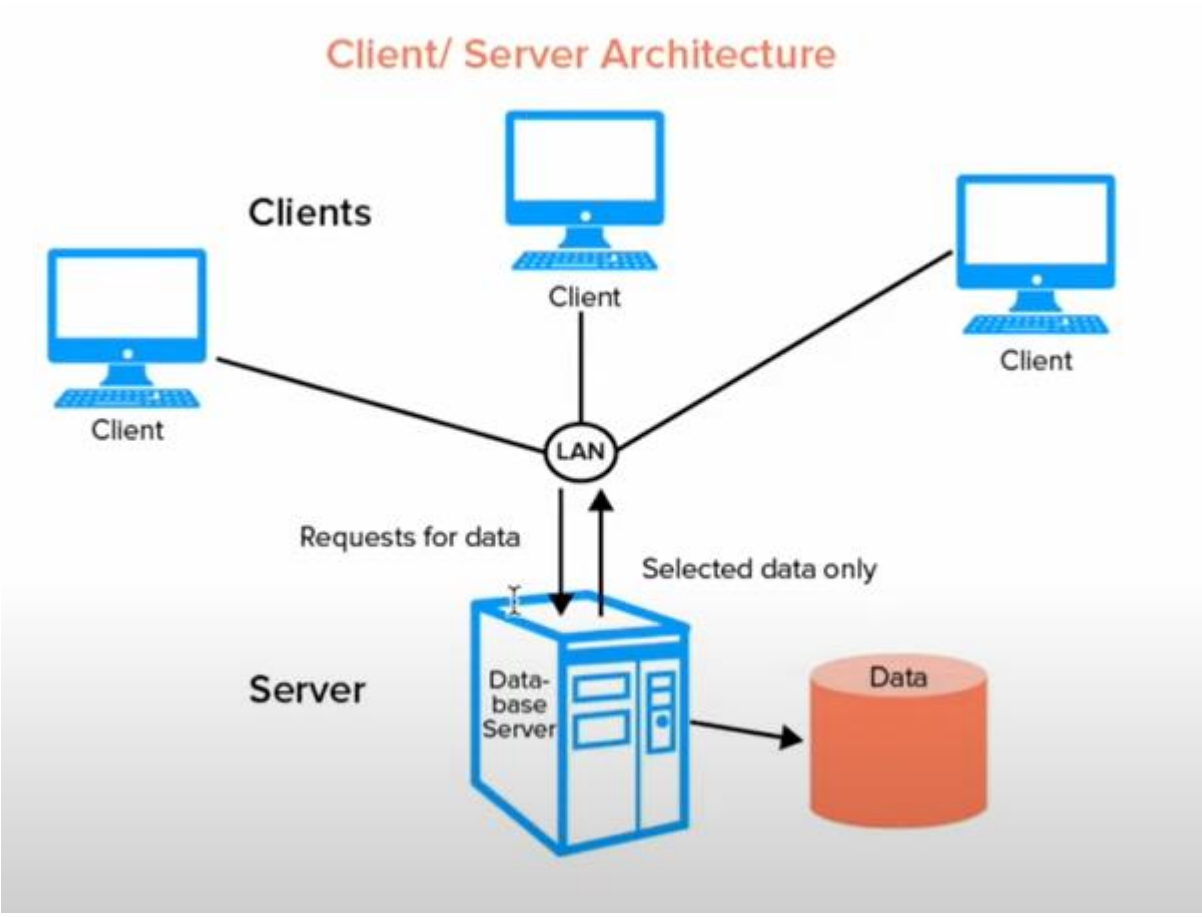
before using we have to install the body-parse:

npm install body-parser

```
const express = require('express')
const app = express();
const db = require('./db');

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

const Person = require('./models/Person');

app.get('/', function (req, res) {
    res.send('Welcome to my hotel... How i can help you ?, we have list of menus')
})


app.listen(3000, ()=>{
    console.log('listening on port 3000');
})
```

- *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);
  }
});
```

it will not work as save does not support call back now. instaed we use try catch system for this purpose

- *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

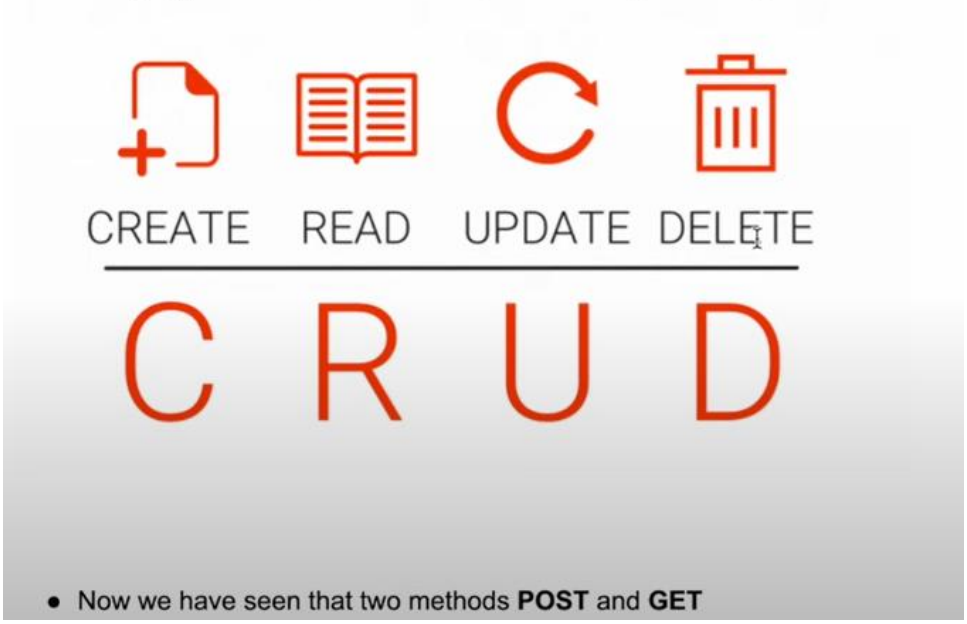
- **Async and await** are features in JavaScript that make it easier to work with asynchronous code, such as network requests, file system operations, or database queries.
- Using try and catch block
- The `try` block contains the code for creating a new `Person` document and saving it to the database 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.

```
server.js > app.post('/person') callback
    res.send('welcome to my hotel... how i can help you ?, we have list of menus')
})

// POST route to add a person
app.post('/person', async (req, res) =>{
    try{
        const data = req.body // Assuming the request body contains the person data

        // Create a new Person document using the Mongoose model
        const newPerson = new Person(data);

        // Save the new person to the database
        const response = await newPerson.save();
        console.log('data saved');
        res.status(200).json(response);
    }
    catch(err){
        console.log(err);
        res.status(500).json({error: 'Internal Server Error'});
    }
})

app.listen(3000, ()=>{
    console.log('listening on port 3000');
})
```

- **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.

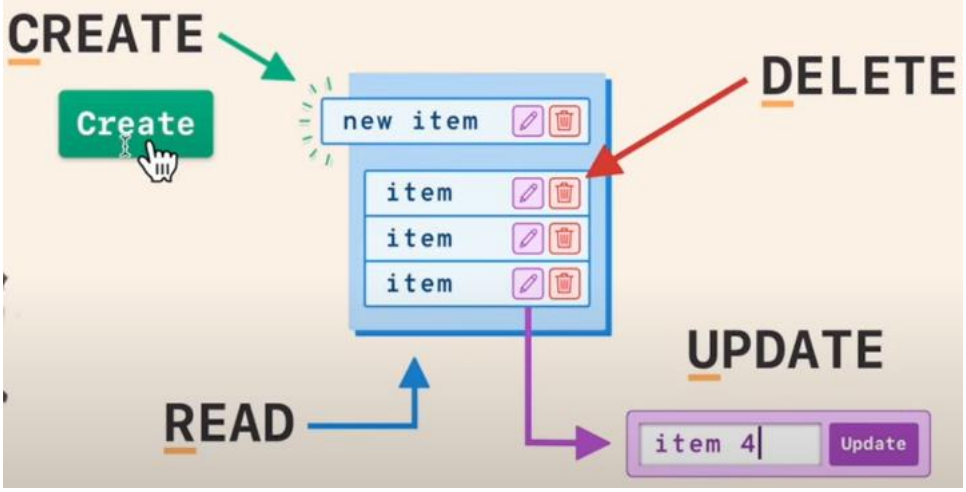
```
const express = require('express');
const path=require("path");
const app = express();
const db=require("./db");
const bodyParser=require('body-parser');
const Student=require('./schema/student');

app.use(bodyParser.json());

const port = 3000;


app.get('/', (req, res) => {
    //res.send('Vikas!')
    res.sendFile(path.join(__dirname,'index.html'))
    //res.json({"Vikas": 47})
  })

app.get('/me', (req, res) => {
    res.send('Hello World from Vikas!')
  })
//saving data into the database
app.post('/stu', async (req,res) => {
  try{
    const data=req.body;
    const newStu=new Student(data);
    const result=await newStu.save();
    console.log('Student data saved');
    res.status(200).json(result);
  }
  catch(err)
  {
    console.log(err);
    res.status(500).json({error:'internal error'});
  }
});
//reading data from database
app.get('/stu', async (req, res) => {
  try{
    const data=await Student.find();
    console.log("Data fetched");
    res.status(200).json(data);
  }
  catch(err)
  {
    console.log(err);
    res.status(500).json({error:'internal error'});
  }
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

- *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



Similarly we can careate more schemas in seprate files

```
models
  Menu.js
  Person.js
node_modules
db.js
package-lock.json
package.json
server.js
```

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

const menuItemSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
    },
    price: {
        type: Number,
        required: true,
    },
    taste: {
        type:
})
```

default values can also be set in the schema definition:

```
models
  Menu.js
  Person.js
node_modules
db.js
package-lock.json
package.json
server.js
```

```
        required: true,
    },
    price: {
        type: Number,
        required: true,
    },
    taste: {
        type: String,
        enum: ['sweet', 'spicy', 'sour'],
        required: true,
    },
    is_drink:{
        type: Boolean,
        default: false
    },
    ingredients:{
        type: [String],
        default: []
    }
})
```

DAY 6

- *Home-work Update for Menu API*

- **Task To create POST /menu and GET /menu**

- We are now creating a POST method to save menu details and it's similar to person details and the same for the GET method

- *Parametrised API calls*

- Now if someone told you to give a list of people who are only waiters
- Then we can create an endpoint like this

- **/person/chef**
- **/person/waiter**
- **/person/manager**

- But this is not the correct method to create as many functions here we can use parametrized endpoints
- It can be dynamically inserted into the URL when making a request to the API.

```
app.get('/person/:workType', async(req, res)=>{
    try{
        const workType = req.params.workType; // // Extract the work type from the URL parameter
        if(workType == 'chef' || workType == 'manager' || workType == 'waiter' ){
            const response = await Person.find({work: workType});
            console.log('response fetched');
            res.status(200).json(response);
        }else{
            res.status(404).json({error: 'Invalid work type'});
        }
    }catch(err){
        console.log(err);
        res.status(500).json({error: 'Internal Server Error'});
    }
})
```

- *Express Router*

- We have a lots of Endpoints in a single file server.js
- This makes bad experience in code readability as well as code handling
- Express Router is a way to modularize and organize your route handling code in an Express.js application.
- So let's create a separate file to manage endpoints /person and /menu
- Express Router is like a traffic cop for your web server
- Express Router helps you organize and manage these pages or endpoints in your web application. It's like creating separate folders for different types of tasks.

- Create a folder `routes` → `personRoutes.js`

```
NODE_TUTORIAL
> models
> node_modules
> routes
  JS personRoutes.js
JS db.js
{} package-lock.json
{} package.json
JS server.js
```

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

// POST route to add a person
router.post('/person', async (req, res) =>{
    try{
        const data = req.body // Assuming the request body contains the person data

        // Create a new Person document using the Mongoose model
        const newPerson = new Person(data);

        // Save the new person to the database
        const response = await newPerson.save();
        console.log('data saved');
        res.status(200).json(response);
    }
    catch(err){
        console.log(err);
        res.status(500).json({error: 'Internal Server Error'});
    }
})
```

```
        console.log('data fetched');
        res.status(200).json(data);
    }catch(err){
        console.log(err);
        res.status(500).json({error: 'Internal Server Error'});
    }
})

router.get('/person/:workType', async(req, res)=>{
    try{
        const workType = req.params.workType; // // Extract the work type from the URL parameter
        if(workType == 'chef' || workType == 'manager' || workType == 'waiter' ){
            const response = await Person.find({work: workType});
            console.log('response fetched');
            res.status(200).json(response);
        }else{
            res.status(404).json({error: 'Invalid work type'});
        }
    }catch(err){
        console.log(err);
        res.status(500).json({error: 'Internal Server Error'});
    }
})
```

remove app.get and app.post methods from index.js and put them in corresponding router files and update the index.js files as shown below:

```
app.get('/', function (req, res) {
    res.send('Welcome to my hotel... How i can help you ?, we have list of menus')
})

// Import the router files
const personRoutes = require('./routes/personRoutes');
const menuItemRoutes = require('./routes/menuItemRoutes');

// Use the routers
app.use('/person', personRoutes);
app.use('/menu', menuItemRoutes);


app.listen(3000, ()=>{
    console.log('listening on port 3000');
})
```

- *Update Operation*

- We will update our person Record, for that we will create an endpoint from where we are able to update record
- For Updation we need two things
  - Which record we want to update ?
  - What exactly we want to update ?
- For update we will use ***PUT*** method to create a endpoint
- *What is a unique identifier in a document in a collection ?*
- It's **_id** which is given by mongodb itself, we will use this to find the particular record which we want to update
- —> And now we will send the data as same like we did in POST method.