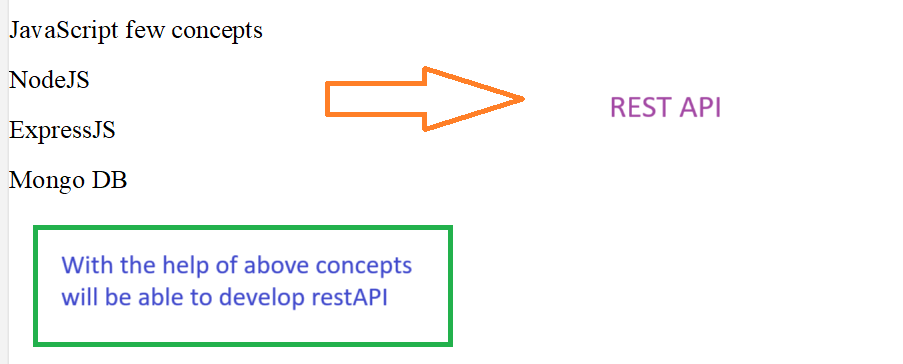
**NODE JS**

BackEnd development with NodeJS



**What is Node.js?**

**Node.js** is an **open-source**, **cross-platform**, **JavaScript runtime environment** that allows you to run JavaScript **outside the browser**, typically on the server.

Node.js uses the V8 engine to execute JavaScript outside the browser. Its non-blocking I/O model and event-driven architecture make it efficient and scalable for building backend services, especially APIs and real-time applications.

1. **JavaScript Runtime Environment**
   * It enables executing JavaScript code **outside the browser**.
   * Built on **Google Chrome’s V8 engine**, which compiles JavaScript to native machine code for high performance.
2. **Open Source**
   * Free to use and actively maintained by the community.
   * You can view, modify, and contribute to the source code — useful for bug fixes and custom features.
3. **Cross-Platform Compatibility**
   * Node.js runs on **Windows, macOS, Linux**, etc.
   * This makes development and deployment more flexible across environments.
4. **Single-Threaded but Asynchronous**
   * Node.js uses an **event-driven, non-blocking I/O** model.
   * Though single-threaded, it can handle thousands of requests concurrently.
5. **Scalable and Fast**
   * Ideal for **I/O-heavy** operations like APIs, file operations, real-time applications (chat, live updates).

**Purpose of Node.js?**

**Backend part of Applications**.=>(can create a lot of Rest API’s).

Create Servers

Scripts

**Backend part of Applications**.

**What is Backend Development?**

**The backend is responsible for managing data, processing business logic, and sending appropriate responses to frontend requests. In Node.js, we use tools like Express, MongoDB, and APIs to manage this.**

* It focuses on:
  + **Functionality of the Application** – What the app actually does behind the scenes.
  + **Business Logic** – The rules and operations that define how data flows and decisions are made in the system.

**Real-Time Examples from the Image**

**Instagram (Social Media):**

* post: Creating a new post
* comment: Adding comments on posts
* like/dislike: Reacting to posts
* follow: Connecting with other users

All of these involve **server-side processing**, storing in databases, and applying logic (e.g., one user can’t follow themselves, etc.)

**Flipkart (E-Commerce):**

* payment: Handling secure transactions
* filters, sort: Business logic for displaying products
* PD (Product Details): Backend fetches product info
* orders: Creating and managing user orders

These are handled in **Node.js or similar backend environments** with databases and APIs.

**For installation:** npm init-- (node setup will take place, package.json will create)

example of package.json

|  |
| --- |
| {  **"name"**: **"backend"**,  **"version"**: **"1.0.0"**,  **"description"**: **"learning nodejs"**,  **"main"**: **"index.js"**,-🡪**renamed as server.js, we can modify as per our wish**  **"scripts"**: {  **"test"**: **"echo** \"**Error: no test specified**\" **&& exit 1"**  **},**  **"repository": {**  **"type": "git",**  **"url": "git+https://github.com/JhanaR/Node.js\_Class.git"🡪github link**  **},**  **"author"**: **"jhana"**,    **"license"**: **"ISC"**,  **"dependencies"**: {-------🡪**npm install express, npm install mongoose**  **"express": "^5.1.0",**  **"mongoose": "^8.16.4"**    }  } |

**Recap of JavaScript:**

**Functions in JavaScript:**

Function is a block in which we can add a set of instructions/statement to do particular task or action.

**Simple**: A function is a set of statements grouped together to perform a specific task.  
It helps in **modularity**, **reusability**, and **clean coding practices**.

**Why Use Functions?**

* To **reuse logic** in multiple places without rewriting code.
* To improve **code readability** and **maintainability**.
* To **encapsulate** logic (especially business logic) within components.

**Named Function, anonymous function and Arrow function etc.**

Difference between **Arrow and Normal** **and Function Expression.**

**Arrow Function**

* Short syntax, good for callbacks.
* Inherits this from the outer scope (lexical this).
* Cannot be used with new or as a constructor.
* Not hoisted.

**const arrow = () => {**

**console.log("This is an arrow function");**

**};**

**Normal Function (Function Declaration)**

* Has its own this context.
* Can be hoisted (called before it's defined).
* Can be used for constructors (with new).

**function namedFunction() {**

**console.log("This is a normal function");**

**}**

**Function Expression**

* A function without a name(anonymous function), assigned to a variable.
* Can be either normal or arrow.
* Not hoisted.

**const anonymousFunction = function () {**

**console.log("This is a function expression");**

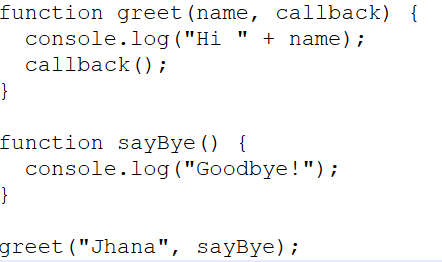
**};**

**Callback functions.**

A **callback** is a function **passed as an argument to another function**, which will be **called (executed)** later inside that function.

**Why Use Callbacks?**

* To **handle asynchronous operations** (like APIs, timers, file reads).
* To execute code **after a task is done**.
* To keep your code **non-blocking** (especially in Node.js).



**What is Callback Hell?**

**Callback Hell** refers to a situation where **multiple nested callbacks** make the code **hard to read, maintain, and debug**.

It usually happens when asynchronous operations depend on one another and are **nested deeply**.

In order to overcome from callback hell we can go with **Promises** or **async/await** to improve readability, reduce nesting, and handle errors better.

**Promises?**

It is a special object in JavaScript that represents the **eventual completion or failure** of an **asynchronous operation** and its resulting value.

It provides a **cleaner alternative to callback hell** and improves **readability and maintainability** of asynchronous code.

Senario:

Output of function1 → passed to function2 with the help of promise

Output of function2 → passed to function3 with the help of promise

We can create promise Using new Promise() constructor

**var promise = new Promise()**

Promise() constructor takes one callback function as a argument with predefined functions--<resolve, reject>

The predefined function will store data in the promise.

var promise = new Promise(function(resolve, reject) { });

**Basic Promise Creation**

**var promise = new Promise(function(resolve, reject) {});**

**console.log(promise); // Prints: Promise {<pending>}**

A Promise is a placeholder for a future value.

At this point, it’s in pending state because:

Neither resolve() nor reject() was called.

**Self-referencing Inside a Promise**

**var promise1 = new Promise(function(resolve, reject) {**

**console.log('promise1', promise1);**

// ReferenceError or undefined--promise1 is still being declared and has not been initialized

**resolve(1234);**

**});**

**If you store data inside the promise using resolve() then that is called success data.  
If you store data inside the promise using reject() then that is called failure data.**

**Multiple resolve() Calls**

**var promise1 = new Promise(function(resolve, reject) {**

**resolve(1234); // This works ✅**

**resolve("pass"); // ❌ Ignored**

**resolve({ name: "jhana", city: "chennai" }); // ❌ Ignored**

**});**

A Promise can be settled only once — either resolved or rejected.

**How to access data from Promises?**

1.Using .then() and .catch().

2.async and await with try catch block

**Using .then() and .catch()**

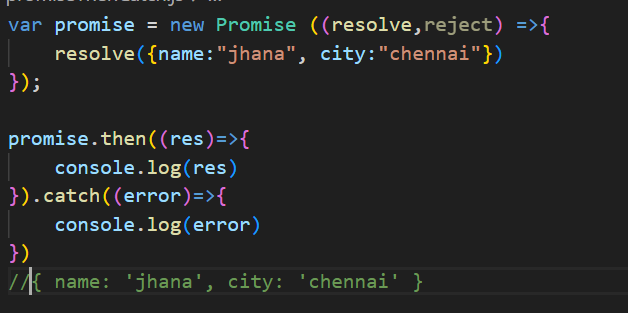
**promiseObj**

**.then(function(successdata){})**

**.catch(function(errordata){})**

.then(function(successdata){}): runs if the promise resolves successfully; successdata is the value passed to resolve().

.catch(function(errordata){}): runs if the promise is rejected; errorData is the value passed to reject().



**async and await with try catch block**

async/await is syntactic sugar over Promises. It allows us to write asynchronous code that looks and behaves more like synchronous code. This improves code readability, avoids .then() chaining, and handles errors using try/catch blocks — making async flows easier to manage, especially in Express routes or data-fetching scenarios.

**Event loop, single-threaded model**

**CommonJS vs ES6 modules**

**Built-in modules: fs, path, http, events, etc.**

**Package.json**

**Local vs global modules**

**Creating a Web Server (using http module)**

In js file.

* Have to import express const express = require('express');

We can import as import import *express* from **"express"**; **If used “type”:module**



* Instance of express need to store in variable. **const** *app* =  express();
* Define a route in order to access API usig url.

app.get('/', (req, res)=>{

**res.send**(“hello World”)

})

const port = 3000;

app.listen(port, () => {

console.log(“server is listening”,+ port);

})

In terminal run : node server.js ------**(node js file name)**

**(npx kill port === to kill port)**

**//CREATING A TODO LIST**

**Let todos =[];**

***app*.post('/todos',(*req*,*res*)=>{**

**const {title, description} = *res*.*body*;---(where the output from API from todos url)**

**const *newTodo* ={// newTodo—storing of API values from todos**

**id : 0,**

***title*,**

***description***

**};**

***todos*.push(*newTodo*);pushing values to todos from newTodo**

***console*.log(*todos*);//logging it**

***res*.status(201).json(*newTodo*);//sending newTodo values JSON format**

**})**

**EXPRESS JS**

**Express.js Framework**

**Middleware**

**Routing**

**Request & Response objects**

**Error handling**

**REST API Development**

**CRUD operations**

**Route parameters vs query parameters**

**Middleware Types**

**Application-level**

**Router-level**

**Error-handling**

**Third-party**

**Asynchronous Programming**

**fs.readFile, setTimeout, Promise, async/await**

**Environment Variables**

**Using .env with dotenv**

**Error Handling**

**Try/catch, next(), error middleware**

It is framework for NODE JS

npm install express.-🡪 **package-lock.json will create**

In package.json-🡪 **under dependencies express will be available.**

import *express* from **"express"**; **so that we can use express JS in node.JS**

**const** *app* =  express();-🡪**express() module calling as function. (Instance of express)**

(OR)

const express = require('express');

const app = express();

**MONGO DB**

**MongoDB Integration**

**Using mongoose with Expres**

**No sql** is a type of data base designed to store, manage and retrieve data without structural format.

It allows for flexible schemas, scalability, and high performance, making it suitable for large-scale applications and real time applications.

Handle key-pair values, graphs and wide columns.

**Mongo DB** is a no sql DB that uses a flexible, document-oriented data model to store data in JSON like BSON format.

It allows dynamic schemas, large volume of data across distributed system.

**JSON**: JavaScript object notation is a light weight data interchange.

It is a simple format to represent structured data object, array and key-pair value.

**BSON**: Binary encoded serialization format that extends JSON like adding data type likes integer, and dates making it more efficient for storing and retrieval.

It is primary data format used by mongo DB for storing documents and making procedure calls.

**INTEGRATION OF MONGODB**

npm install mongoose

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

*mongoose*.connect(**'mongodb://localhost:27017/mern-app'**)

.then(()**=>**{

*console*.log(**'DB connected'**)

})

.catch((*err*)**=>**{

*console*.log(*err*);

})

**Connecting to MongoDB Server**

Step 1: Open command prompt(cmd) in your system

Step 2: Type 'mongosh' then hit enter key

**Show database**

show dbs: It will show all the available databases

**Command to connect database**

#Syntax: use [database name]: It will pick you which data base need to work on

|  |  |
| --- | --- |
| How to delete collection:   |  | | --- | | db.collectionname.drop() ex: db.orders.drop() | |

**CRUD Operation**

1. Create
   * insertOne(data,options)
   * insertMany(data,options)

**Creating Collection & Document**

#Syntax: db.collectionName.insertOne({document})

#Example db.products.insertOne({name:'Mac Book Pro',price:120000,brand:'Apple'})

Under the database need to create a collection in that we have to inserted one document.

Example: db.products.insertMany([{name:'Iphone14',price:75000,brand:'Apple'},{name:'Samsung Galaxy A50',price:54000,brand:'Samsung'}])

Under the database need to create a collection in that we have to insert more than one document.

1. Read
   * find(filter,options)
   * findOne(filter,options) #first matching document

**To get all the data from the collection**

#Syntax: db.collectionName.find(): It will show us the document available in that particular collection(1st set)

db.customers.find().toArray():All data

db.customers.find().forEach((customersData)=> {printjson(customersData)})

db.products.find().forEach((*customersData*)=> {*printjson*(*customersData*)})

db.products.find({},{name:1,price:1}) :it will list only name and price along with id

db.products.find({},{\_id:0,name:1,price:1}) :it will list only name and price.

#Syntax: db.collectionName.find({particular document}): It will show us current document available in that particular db

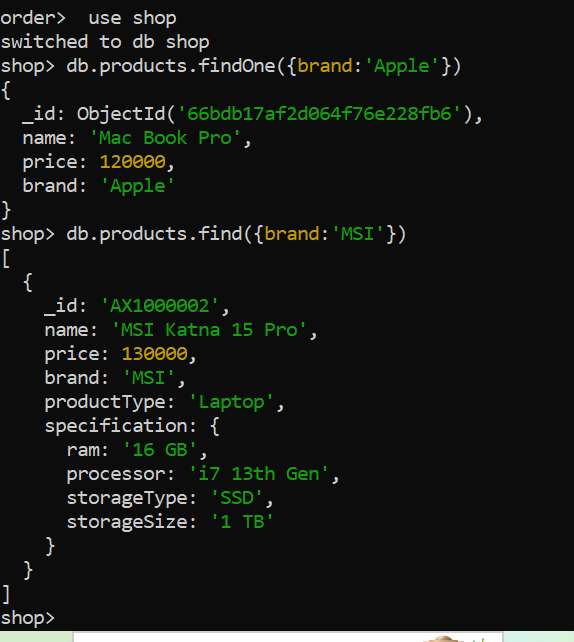
#Syntax: db.collectionName.findOne({particular document}): It will show us first one of the current document available in that particular db

Example:

db.products.find({brand:'Apple'}) :

Example:

db.products.findOne({brand:'Apple'})



1. Update
   * updateOne(filter,data,options)
   * updateMany(filter,data,options)
   * replaceOne(filter,data,options)

**Update document in collection**

#Syntax: Db.collectionName.updateOne({},{}): filter and what to set

**Example:**

db.products.updateOne({name:'Mac Book Pro'},{$set:{productType:'Laptop'}})

**Example:**

db.products.updateOne({name:'Samsung Galaxy s24 Ultra'},{$set:{price:93000}})

**Example:**

db.products.updateOne({name:'Bravia 43 inch 4k'},{$set:{price:45000}})

**Example:**

db.products.updateOne({name:'Iphone 15'},{$set:{brand:'Apple',productType:'Mobile'}})

**Example:**

db.products.updateMany({\_id:{$in:[ObjectId('66695c4aa52b865509cdcdf8'),ObjectId('66695c8ba52b865509cdcdfd')]}},{$set:{productType:"Mobile"}})

**Example update many**

db.products.updateMany({\_id:{$in:[ObjectId('66695c4aa52b865509cdcdf8'),ObjectId('66695c8ba52b865509cdcdfd')]}},{$set:{productType:"Mobile"}})

**Replacing of entire document:**

db.products.replaceOne({\_id:ObjectId('66695c8ba52b865509cdcdfc')},{name:"Iphone 14 Pro Max",price:115000,brand:'Apple',productType:'Mobile'})

1. Delete
   * deleteOne(filter,options)
   * deleteMany(filter,options)

## Delete document from collection

# the following command will delete the 1st document with matched filter

db.products.deleteOne({brand:'Apple'})

# the following command will delete document many at same time

db.products.deleteMany({productType:'Laptop'}) documents matches the filter will delete from collections

db.products.deleteMany({})--- all documents will delete from collections

**deleting collection:**

db.products.drop()---collection will delete

**Deleting Database:**

db.dropDatabase()