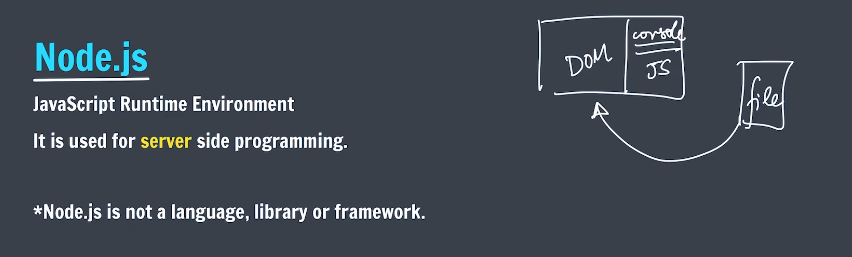
# Backend 1





We’re writing in node, not in browser.

Therefore, it’s not possible to access ‘window’ object in node. There is something similar called ‘global’.

Process in Node



//By default, it has two elements in the array.

//subsequent element will be command-line argument that we

//provide.

console.log(process.argv);

//you can remove the first two elements using slice and use

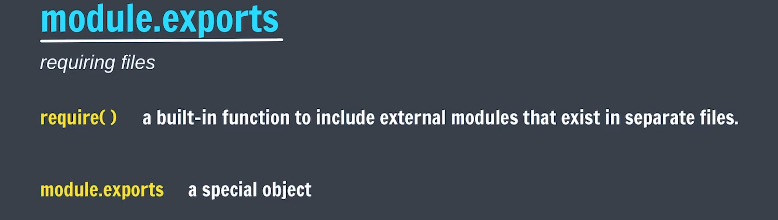
//the remaining



First element id executable path for node.

Second Is our current path of file.

Export Files



*const* sum = (*a*, *b*) *=>* *a* + *b*

*const* mul = (*a*, *b*) *=>* *a* \* *b*

*const* PI = 3.14

//creating an object.

*const* obj = {

    sum: sum,

    mul: mul,

    PI : PI,

}

module.exports = obj;//exporting object.

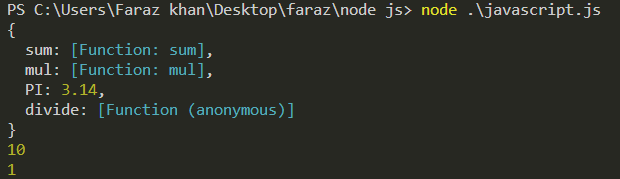
*module*.*exports*.divide = (*a* , *b*)*=>* *a*/*b*//directly exporting function

*const* help = require('./helper')

console.log(help); //object is received

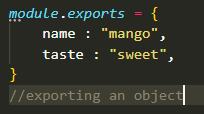
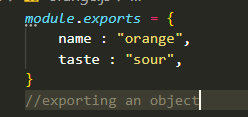
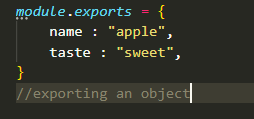
console.log(help.sum(5,5));

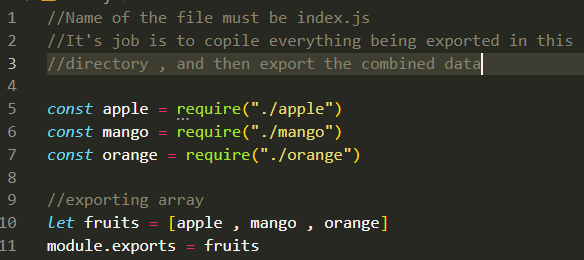
console.log(help.divide(5,5));



It the above example, everything is in same directory.

Export in Directories





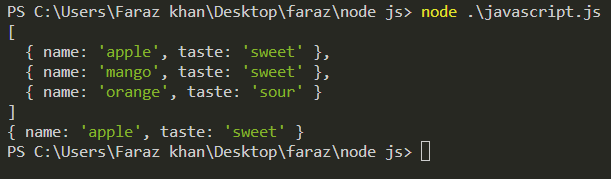
*const* fruits\_info =  require('./fruits')

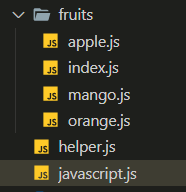
//requiring a directory

//node will search of index.js in 'fruits' directory

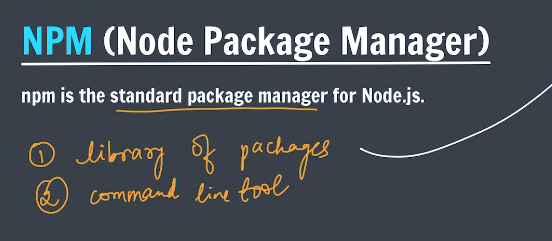
console.log(fruits\_info);

console.log(fruits\_info[0]);





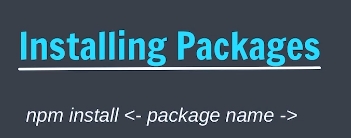
NPM

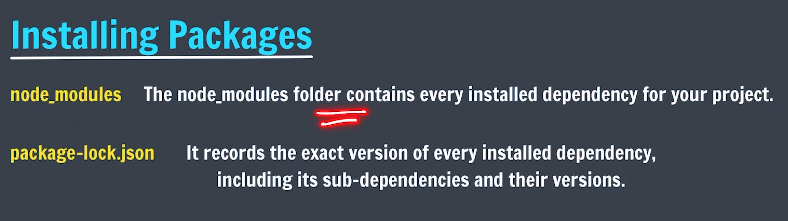


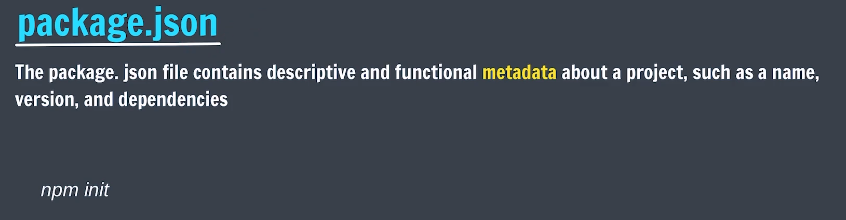
npm is a package manager for the JavaScript programming language maintained by npm, Inc. npm is the default package manager for the JavaScript runtime environment Node.js and is **included** as a recommended feature in the Node.js installer.

Package will be installed in the directory you are currently on. How to use that package is there in the docs.

You have to require those package explicitly.







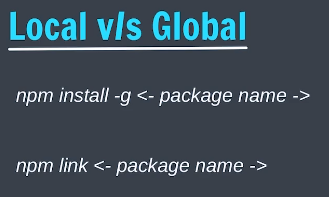
Package.json contains all the important dependencies of your projects, i.e. the things your project relies on. Node modules can be re-downloaded with the help of package.json as it contains all the dependencies.

For example, if you delete the node module folder, you can simply re-install it using “*npm install*”, it will install every dependencies your project needs.

Whenever you need to send your project to someone, you only need package.json, and using it, other dependencies can directly be downloaded from their source.

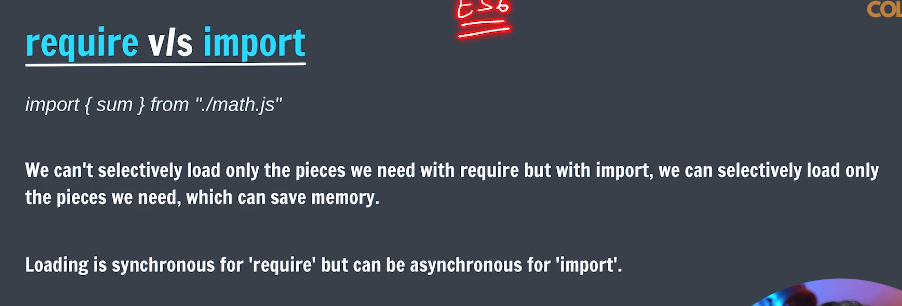
You can create your own package.json for your project using “*npm init*”. “*npm init -y*” creates default.

Local vs Global installation



We will always prefer local installation.

Require vs import



When you want to use import, you have to make “type”:”module” as a key value pair in the package.json file.

export *const* sum = (*a*, *b*) *=>* *a* + *b*

export *const* mul = (*a*, *b*) *=>* *a* \* *b*

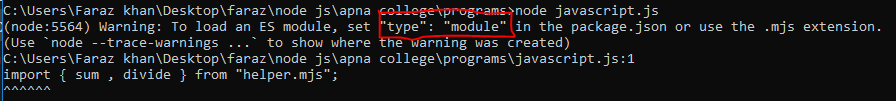
export *const* PI = 3.14

import { sum , PI } from "./helper.js";

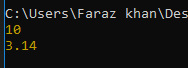
// './' likhna mat bhulna!

console.log(sum(5,5));

console.log(PI);

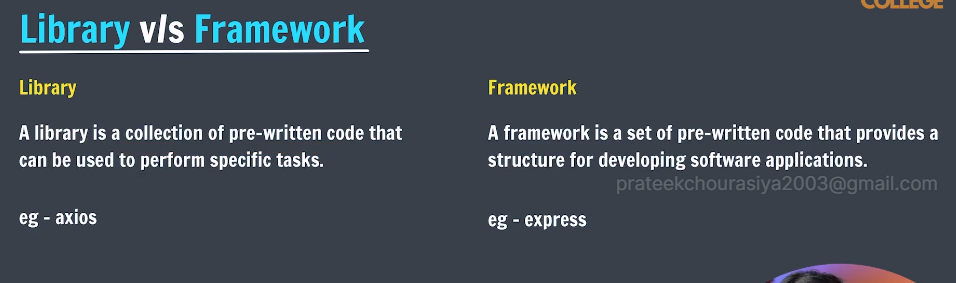


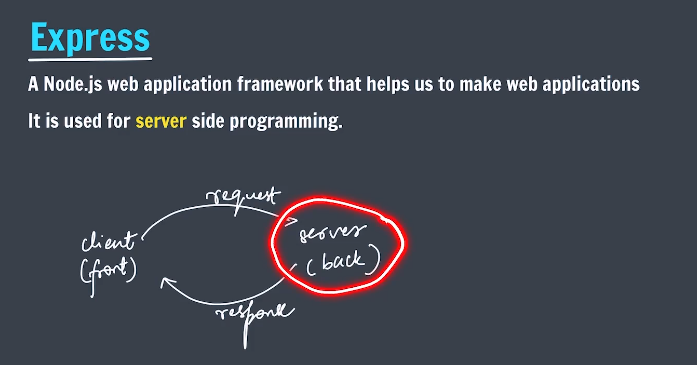
Create a package.json using “npm init” , and make the change.

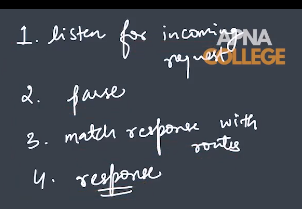


# Backend 2

What is express?







1. Listens for incoming request.
2. Parse.

Getting Started with express



Running Locally

First create a directory named myapp, change to it and run npm init. Then install express as a dependency, as per the [installation guide](https://expressjs.com/en/starter/installing.html).

In the myapp directory, create a file named app.js and copy in the code from the example above.

Then, load http://localhost:3000/ in a browser to see the output.

*const* express = require('express')

*const* app = express()//using app we will make our server

*const* port = 3000

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

*res*.send('Hello World!')

})

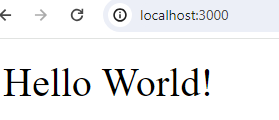
//listen method listens at port and executes callback function.

//It starts the server

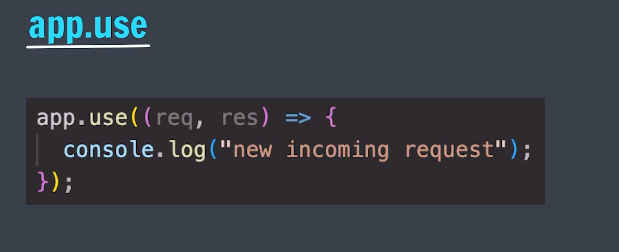
app.listen(port, () *=>* {

  console.log(`Example app listening on port ${port}`)

})

Handling requests



To use hopscotch for request to local host, add chrome extension and set interceptor to browser extension.

*const* express = require('express')

*const* app = express()//using app we will make our server

*const* port = 3000

//listen method listens at port and executes callback function.

//It starts the server. It listen to the port.

app.listen(port, () *=>* {

  console.log(`Example app listening on port ${port}`)

})

//This method will listen for any kind of request

//and then executes the call-back when any request is received

app.use((*req* , *res*)*=>*{

    console.log("request received");

    //this will get printed when a request is received from

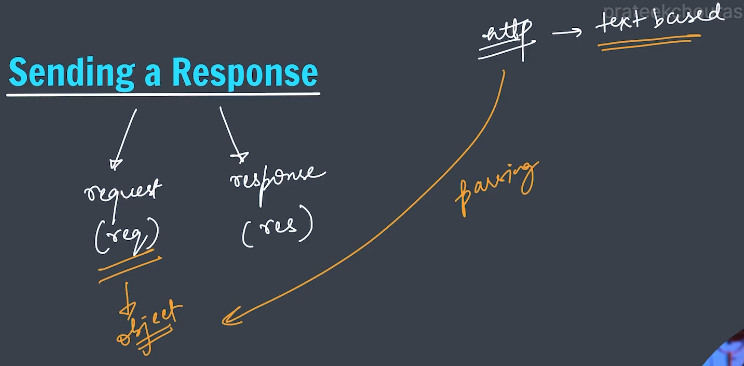
    //browser of hoppscotch

})

The above code will only listen for any request but is not sending any response, therefore hopscotch will load till infinity.

Req (request) and res (response) are objects provided by express by default.

* + Req contains a lot of data.
  + Res will be used to send response.



Https requests are text based, therefore server written in any language can handle them.

Express parses the request into objects which we can use.

Sending a response

Only a single response can be send.

We will use res.send() method

<https://expressjs.com/en/5x/api.html#res.send>

### res.send([body])

Sends the HTTP response.

The body parameter can be a Buffer object, a String, an object, Boolean, or an Array. For example:

res.send(Buffer.from('whoop'))

res.send({ some: 'json' })

res.send('<p>some html</p>')

res.status(404).send('Sorry, we cannot find that!')

res.status(500).send({ error: 'something blew up' })

app.use((*req* , *res*)*=>*{

    console.log("request received");

    //this will get printed when a request is received from

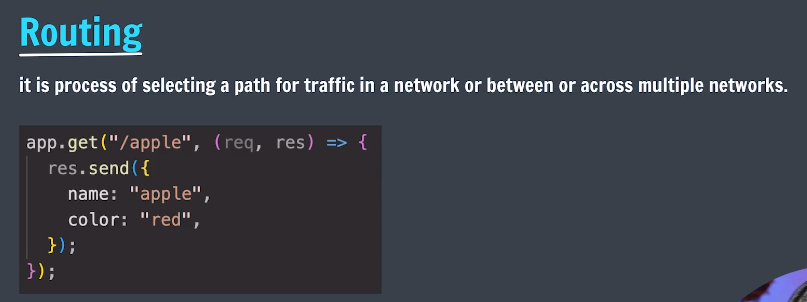
    //browser of hoppscotch

*res*.send("You can send anything")

})

Routing

In big websites, there are many pages , and in the URL, you get to see many paths which can be considered as different routes.



The ‘***app.use()’*** methodlisten to all the routes and all the types of request which we don’t want.

Therefore we will use separate method for all of them.

### app.get(path, callback [, callback ...])

Routes HTTP GET requests to the specified path with the specified callback functions.

Only one of them which matches the path executes.

Similarly for post request.

*const* express = require('express')

*const* app = express()//using app we will make our server

*const* port = 3000

app.listen(port, () *=>* {

  console.log(`Example app listening on port ${port}`)

})

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

*res*.send("This GET request is for root path");

})

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

*res*.send("This GET request is for /home path");

})

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

*res*.send("This GET request is for /search path");

})

app.get("\*" , (*req* , *res*)*=>*{

*res*.send("This GET request is for NONE of the above path")

})

app.post("/" , ()*=>*{

    res.send("You send a post request for /root path")

})

app.post("/home" , ()*=>*{

    res.send("You send a post request for /home path")

})

app.post("\*" , ()*=>*{

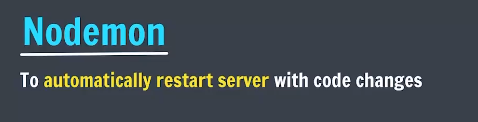
    res.send("You send a post request for none of the above path")

})





Installing



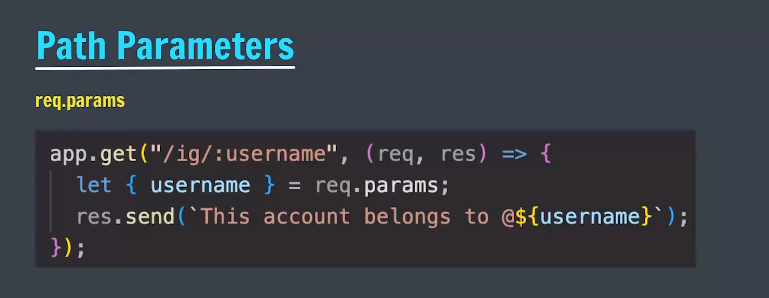
Install globally using npm. (npm I -g nodemon)

Run javascript file using nodemon name.js

Path parameters

Path parameters are variable parts of a URL path. They are typically used to point to a specific resource within a collection, such as a user identified by ID.

**:variable –** Signifies a path parameter in URL



app.listen(port, () *=>* {

  console.log(`Example app listening on port ${port}`)

})

//the value in URL will be assigned in 'username' and 'lastname'

app.get("/:username/:lastname" , (*req* , *res*)*=>*{

    console.log(*req*.params);

    console.log(typeof(*req*.params));//is an object

*let* {username , lastname} = *req*.params

    //destructuring object, name of variables should be

    //same as that in object.

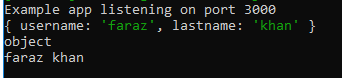
    console.log(`${username} ${lastname}`);

*res*.send(`Welcome ${username} ${lastname}`);

})







Query string

We can also make use of query strings that come in URL.

app.listen(port, () *=>* {

  console.log(`Example app listening on port ${port}`)

})

//the value in URL will be assigned in 'username' and 'lastname'

app.get("/search" , (*req* , *res*)*=>*{

    console.log(*req*.query);//object

*let* {q , lastname} = *req*.query

    //when destructuring an object, name of the keys should be same as

    //variables

    //We know the query parameters in advance that would come which are

    //'q' 'lastname'

*res*.send(*req*.query)//sending object

})



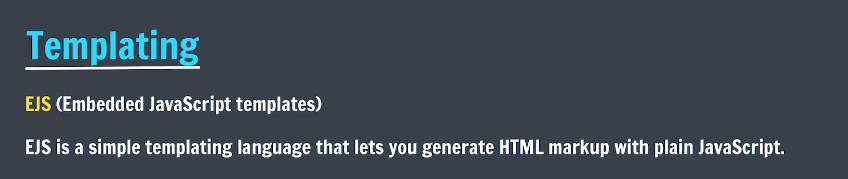




# Backend 3

Templating

***npm install ejs***



EJS, or Embedded JavaScript, is a popular template engine for Node. js and web development. It allows you to generate dynamic HTML content by embedding JavaScript code within your HTML templates

Using ejs

*const* express = require('express')

*const* app = express()

*const* port = 3000;

//app.set() method is used to set many things

app.set("view engine" , "ejs")

//setting view engine to 'ejs'

//view engine will search for 'views' folder for templates

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

*res*.render('home.ejs')

    //rendering home.ejs file

    //express knows that home.ejs is in views folder

})

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

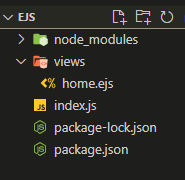
*res*.send("path is /search")

})

app.listen(port , ()*=>*{

    console.log("server started");

})



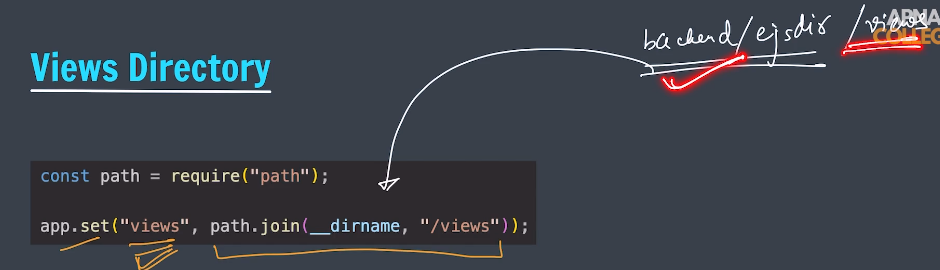
What the above code does Is, when a GET request is made to the server, it renders a ‘*home.ejs*’ which is basically HTML which is dynamic in nature. Therefore, you can render HTML based on the request. You can send different ‘*.ejs*’ files based on the path on which request is made.

‘*.ejs*’ files must be inside the views folder in which is inside the directory ‘*index,js’* is.

The above code will only work correctly if you start your server at the current directory. “*Views*” folder is inside the directory in which we’re starting the server.

If we start our server from some other directory using ‘*index.js*’ absolute path, then views folder will be searched in the directory from where the server was started and will throw error.

If we can somehow set that ‘*views*’ will be present inside the same directory as that of ‘*index.js*’, then it will always work.



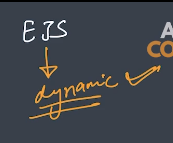
We’re setting path of ‘*views*’ using path.join() method (path package has to be imported). ‘*\_\_dirname*’ is the path of ‘*index.js*’ and we’re appending ‘*/views*’ on that path, therefore express will always find ‘*views*’ in the same directory as that of index.

We’re setting the path of views directory to be a constant.

Name must be **views.**

Interpolation syntax





Ejs has tags which you can use to embed expressions in HTML. <https://ejs.co/#about>

Ejs file:-

    <h2> <%= 1 + 3 %> </h2>

Passing data to ejs

Create an object of anything and send it using the second parameter of res.render( , object).

You can access the object keys just by their name.

*const* express = require('express')

*const* app = express()

*const* port = 3000;

*const* path = require('path')

app.set("views" , path.join(\_\_dirname , "/views"))

app.set("view engine" , "ejs")

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

*res*.render('home.ejs')

})

//request is made to this specific path.

app.get('/some\_path' , (*req* , *res*)*=>*{

    //data will come from database, we're using random

    //number to mimic

*let* variable = Math.floor(Math.random()\*6)+1

    //rendering 'dice.ejs' and sending data as object

    //having key value pair

*res*.render('dice.ejs' , {value : variable,})

“or” *res*.render('dice.ejs' , { variable}) //directly send the variables separated by ‘,’

})

app.listen(port , ()*=>*{

    console.log("server started");

})

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Dice</title>

</head>

<body>

    <!-- 'value' is received from index.js -->

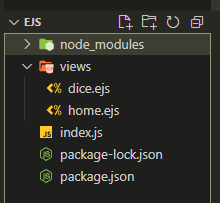
    <h1>Dice value is : <%= value %></h1>

    <img src="<%= value %>">

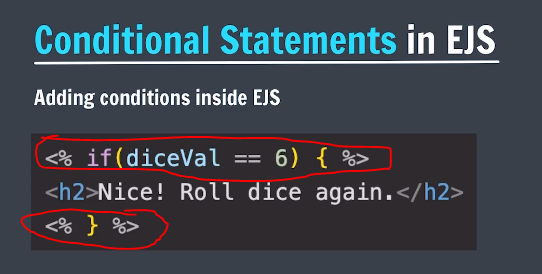
*<!-- you can also use it as such -->*

</body>

</html>

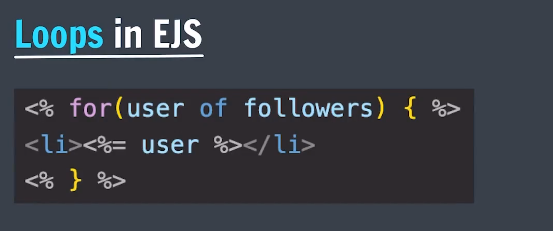


Conditionals Statements in EJS



The tag is different observe! <% %>

Loops in EJS



## Serving static file:



We know from 12 hour backend about static hoisting.

WRT to ejs, if a folder is made static, then inside the .ejs file, it automatically looks for the given file inside the static folders.

Given below is the ejs file which is inside views folder.

We’re accessing style.css file which has a path public/css/insta\_css/style.css. We’ve made public/css as static therefore we don’t have to specify public/css before. Ejs automatically looks for the given path from the static folders.

We can also write the full path like we usually do. That also works.

*//We're making 2 static folders which are inside public.*

*//We can directly reference the static folders from our ejs files.*

app.use(express.static(path.join(\_\_dirname , "public/css")))

app.use(express.static(path.join(\_\_dirname , "public/javascript")))

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

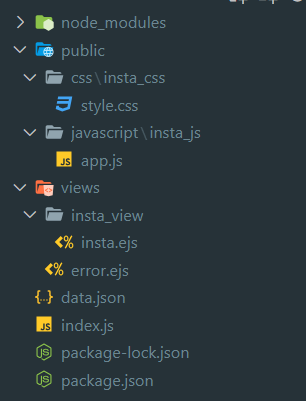
    <title>Insta</title>

    <link rel="stylesheet" href="/insta\_css/style.css">

*<!-- public/css is already known to ejs as we've set it as static folder -->*

*<!-- You can also give the full path if you want -->*

</head>



## Includes

A close-up of a blackboard

Description automatically generated

Used to insert sub templates within a template.

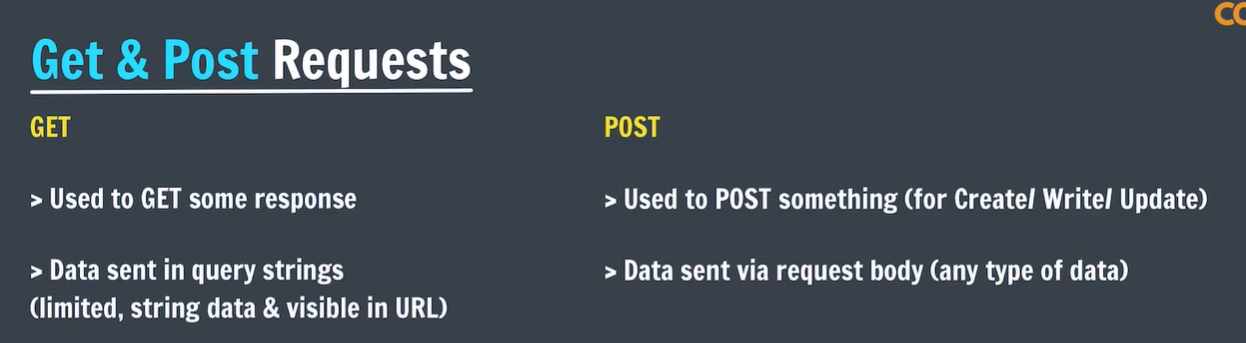
Whaterver code you’ve written in any ejs file will as it is get copy pasted in the location where you’ve included.

## Summary of paths:

1. All the ejs template files should be inside folder named views and you’ve to hardcode the path of the views directory using path.join() with respect to index.js
2. When you render any template from index.js, you can either provide the full path of the template or you can just give the path from views directory onwards. It always searches from views directory.
3. If you want to set css and javascript file for for these ejs, method is to create a public folder and make it static using express.static(). From the ejs template files, you can directly access the files using the full path or provide the path from public onwards as public is static.
4. If you want to include template within a template, use includes and provide the path of the sub template WRT the current template.

Ejs demo : <https://github.com/Faraz-Khan-79996/Backend-apna-college/tree/main/ejs%20demo>

# Miscellaneous



When a data is coming from **GET** request from a plain HTML form, data comes from **query string** with **name** from the form. You can access it using req.query().

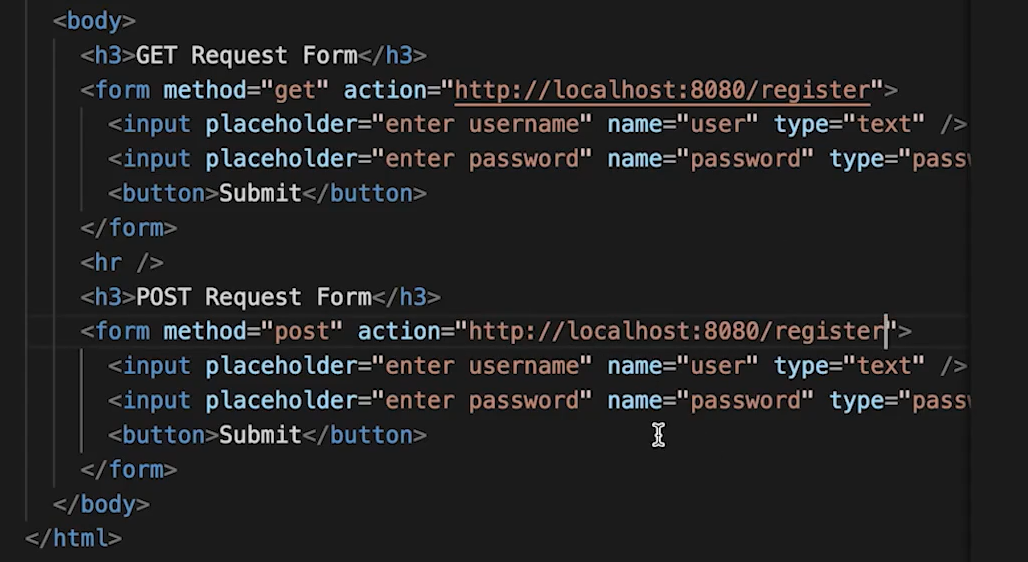
In case of **POST** request, data comes from request body. The data has to be parsed before using it.

When we use **plain HTML** form for POST request, data comes in **urlencoded** form. It can also come in **json** form. You can use the below two middleware to parse them otherwise you won’t be able to use the data from **req.body. req.body** is an object after being parsed.

Use both middleware, if data is json or urlencoded it will get parsed.

A screen shot of a computer

Description automatically generated



Plain HTML form, **method** is type of request and action is the **url**.

## Javascript Oops:



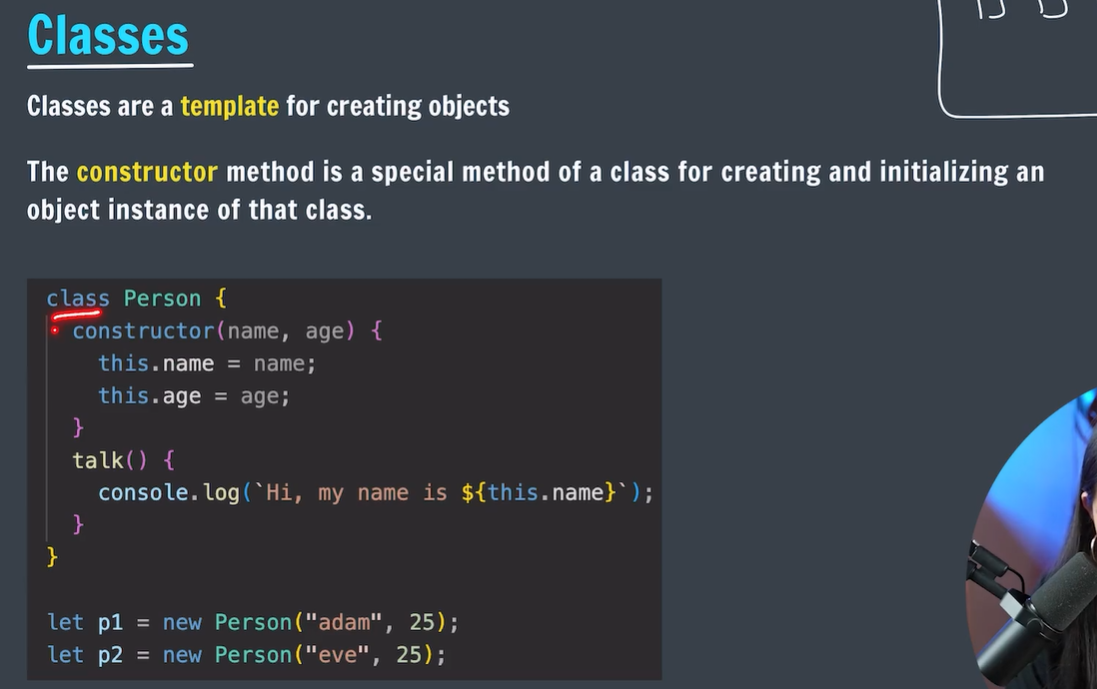
A computer screen with colorful text

Description automatically generated

Making object using factory functions. Inefficient way to do it.

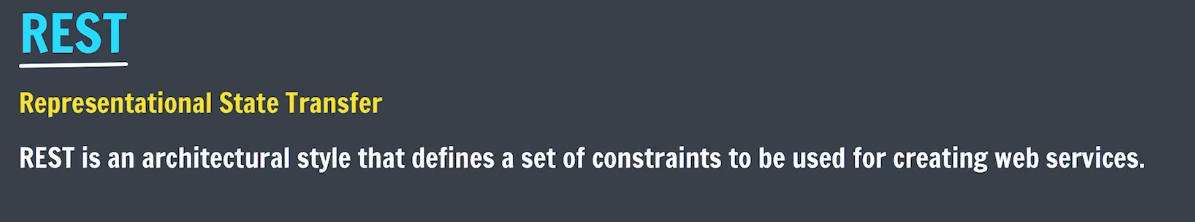
A screenshot of a computer program

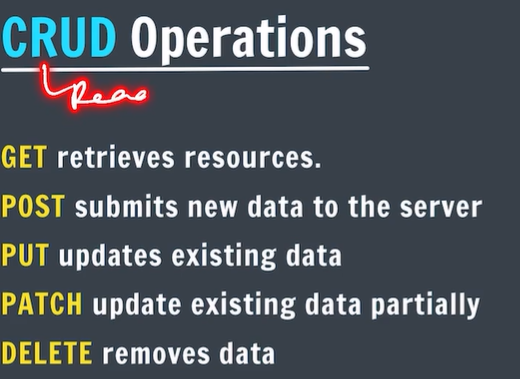
Description automatically generated





# Rest





# MonoDb (part 2)

A screenshot of a computer

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screen shot of a computer

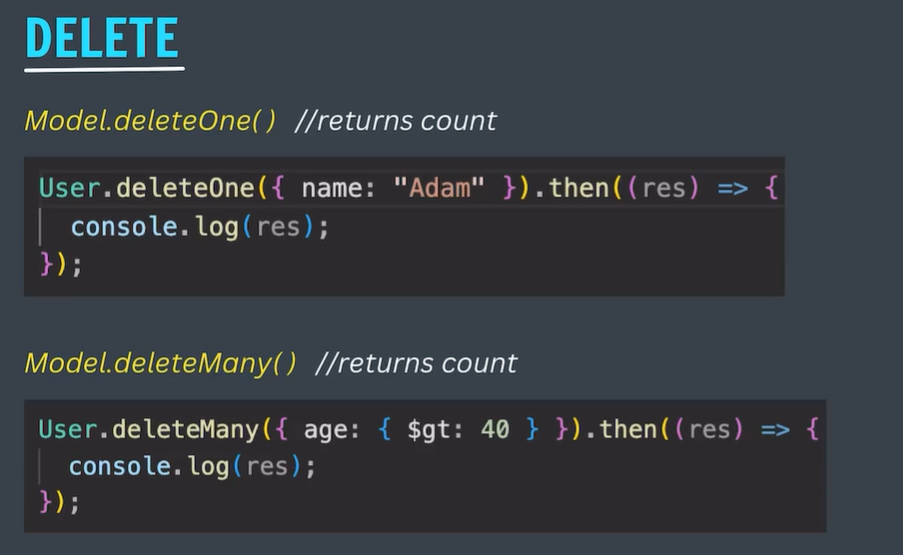
Description automatically generated

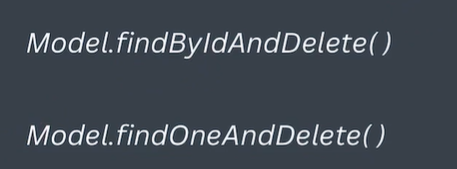
A screenshot of a computer program

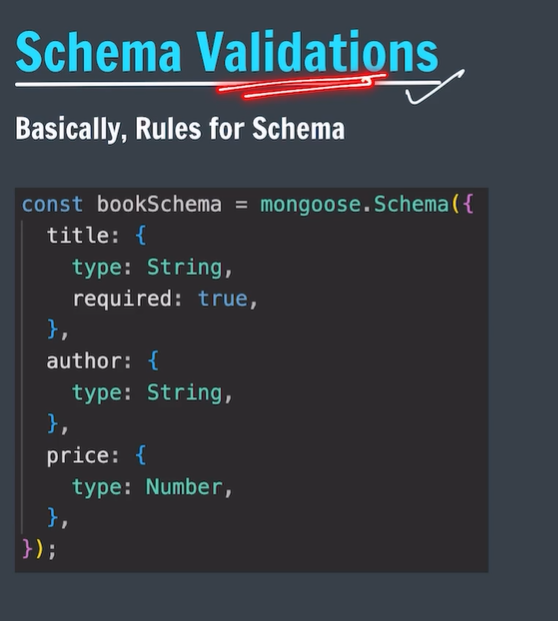
Description automatically generated

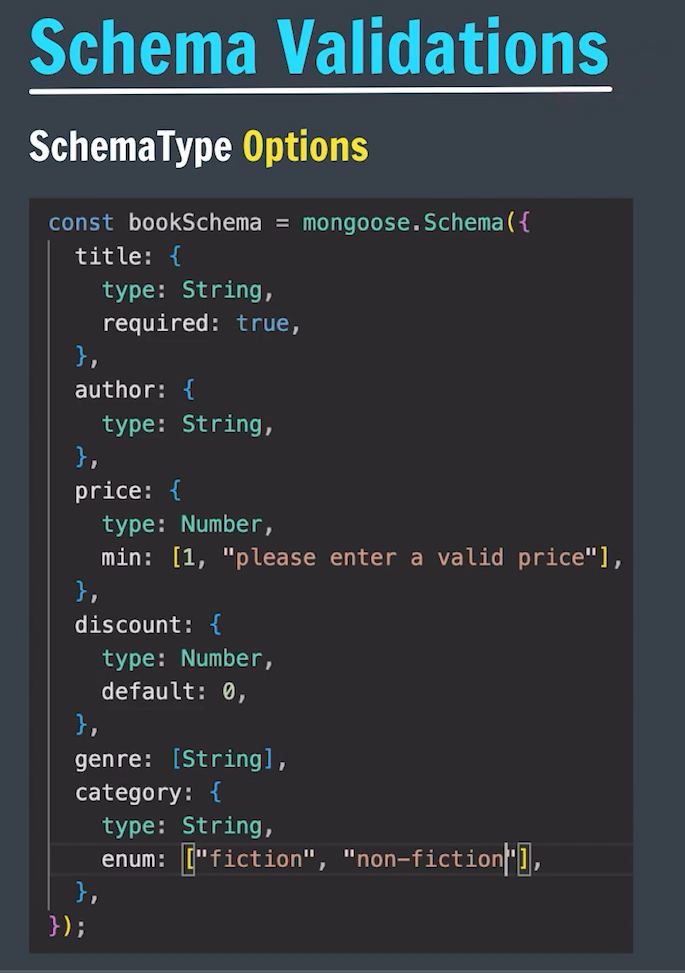
Add a third parameter **{new:true}** as options to get updated document from the database. By default it’s **false.**







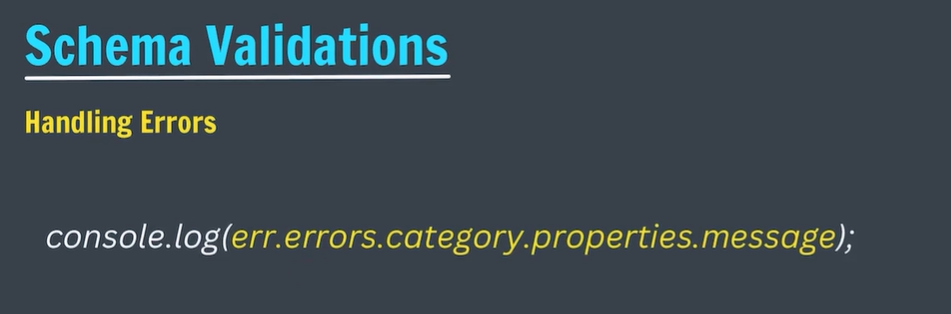






Schema Validations only work at creation of document, NOT at updation.

Inside options object of any update method, add **runValidators:true .** Now validations will be applied at the time of updation also.



Sending request other than GET or POST directly from form.

const methodOverride = require('method-override')

//used to take request other than POST and GET directly from <form>

app.use(methodOverride('\_method'))

// method-override middleware

                <!-- You can't directly send request other than GET or POST directly -->

                <!-- method-override library is used for that. -->

                <!-- apply \_method=DELETE as query string and form method as POST -->

                <form action="/chats/<%= chat.\_id %>?\_method=DELETE" method="post">

                    <button>Delete</button>

                </form>

# Backend 6

* Chaining possible in middleware.
* They have access to request and response object.
* Middleware itself can send a response.
* They can execute any code.
* Make changes to request and response objects.
* End the request response cycle.
* Call the next middleware function.
* You can mount middleware on a specific route.

# Errors

const express = require('express')

const ExpressError = require('./expressError')

//This is our custom error class

const app = express()

const checkToken = (req , res , next)=>{

    let {token} = req.query;

    if(token === "giveaccess"){

        next();

    }

    throw new ExpressError(401 , "ACCESS DENIED : token should be giveaccess")

}

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

    res.send('Your token was : <strong>giveaccess</strong> <br> which is correct!')

})

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

    a = rtytk//creating an error

    //error which is thrown is not custom but by javascript

    //it does not contain 'status'

    //it DOES contain message

    res.send('hello world')

})

//any error which is thrown will be handled by these middlewares.

app.use((err , req , res , next)=>{

    console.log("---------------------------ERROR ---------------------------");

    next(err)

    //next() without parameter will search for next NON error handling middleware or normal middlware.

    //next(err) will search for next error handling middleware.

})

app.use((err , req , res , next)=>{

    console.log('------------------------- Error2 -------------------------');

    next(err)

    //If there're no next error handling middleware, default error handler of express will execute.

})

app.use((err , req , res , next) =>{

    const {status=500 , message } = err;

    //Taking out status and message from 'err' object which was thrown.

    //In case of '/error' route, default error will get thrown by by node,

    //which does not have a 'status', therefore 'status' is 'undefined'.

    //if 'status' is undefined therefor error will occur as express does not

    //undefinded inside res.status() method.

    res.status(status).send(message)

})

app.listen(3000)

class ExpressError extends Error {

    constructor(status , message){

        super()

        this.status = status

        this.message = message

    }

}

module.exports = ExpressError;



//You can also do this in case of async

try {

    // some code

    //errors thrown by third party library functions

} catch (error) {

    next(error)

    //calling our handler.

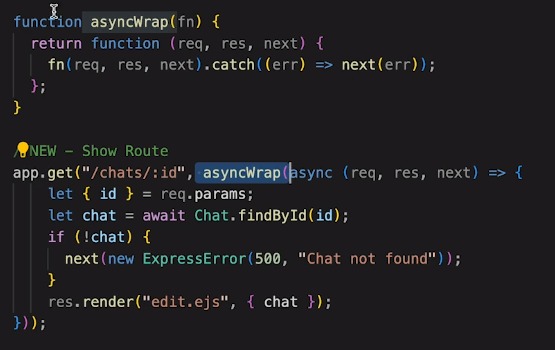
    //If no handler, express default handler executes

}

In case of normal errors, it is thrown and next called on its own, but in case of asynchronous errors,

You have to enclose your error inside ***next()*** in case of asynchronous errors as express does not call next() by default in case of asynchronous errors.

A better way instead of try and catch in case of async function is to use a function named **wrapAsync (**name can be any**),**  as follows:

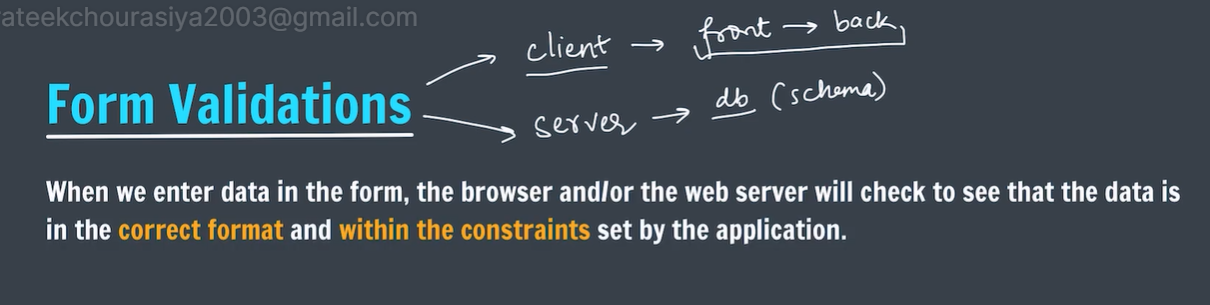


A computer screen with text

Description automatically generated

Mongoose error has name property from which you can identify which kind of error that was.

# Form Validation



Validating the form on frontend. Prevent Bad data from going in request.

Use bootstrap docs.

In the backend, using the knowledge of previous chapter, We :

* We created **wrapAsync** function instead of try catch to handle async errors.
* We created our custom error class names **ExpressError**.
* Created error handling middleware in last and rendered an error.ejs page.
* Created a page not found middleware. **App.all(‘\*’ , (req , res , next))** to generate a page not found error.

Before saving the incoming data to mongodb, we validate that data using a library **joi.**

* Create a schema using joi similar to your mongoose schema.
* Create a middleware which takes your form data.
* Use validate method and pass your data as object.
* If there is error in response, handle error else next()

# Database relationship

Code inside backend-apna college/database relationship

## One to few

const

userSchema = mongoose.Schema({

username

: String,

addresses

:[

{

\_id

: false,//Using false won't create object id for each individual

object of an array

//

as

the sub-object is being considered as another schema.

location

: String,

city

: String,

}

,

]

,

})

;

const

User = mongoose.model('User' , userSchema)

module

.exports = User

adresses is an array of objects where each object has some properties. The subobjects will get treated as another shcema and ObjectId will get generated for them as well if you don't use \_id:false



## One to many

Store the reference of child document inside parent.

in schema, use type : mongoose.Schema.Types.ObjectId and ref : 'name of collection in singular form' like ref :'Order'

At the time of creation and saving the object to a collection, add the whole object at the place of 'ref'. The object will get replaced by it's ObjectId in the database.

At the time of fetching data, use populate to populate the document with the document whole Id is stored.

let

cust

1

= new Customer({

name

:

"faraz the great"

,

})

;

const

order

1

= await Order.findOne({item:

"mobile"

})

const

order

2

= await Order.findOne({item:

"laptop"

})

cust1

.orders.push(order

1

)

cust1

.orders.push(order

2

)

const

res = await cust

1

.save()

const

result=await Customer.findOne({}).populate('orders')

//

Populates

the orders field with the actual documents based

on

the id stored in

orders field.

## One to Squillions

opposite of the above approach. We will store the reference to the parent inside of the child.

# Handling deletion in DB



*//In case of "save"*

*//If you use arrow function, you won't get access to this.*

*//Use normal function if you want to make changes to the data before saving.*

customerSchema.pre("save" , function (*next*) {

*//You can access object about to be saved using this keyword.*

    console.log("Pre-Hool : save : ",this.name);

    next() *//call next as it's a middleware.*

})

customerSchema.pre("findOneAndDelete" , function (*next*) {

*//You can access using this keyword.*

*//console.log(this.name);//it's NOT the object which is about to be deleted.*

    console.log( "PRE-HOOK  " , this.getQuery());*//it's the objectId which is about to be deleted.*

    next() *//call next as it's a middleware.*

})

*//When you delete a customer, you want to delete all the corresponding 'orders'.*

*//'orders' contain ObjectId of of docuemtns of 'orders' collection.*

customerSchema.post("findOneAndDelete" , async(*customer*)=>{

    console.log("POST deleting document")

    const all\_orders = *customer*.orders.length;*//customer.orders  is an array.*

*if*(all\_orders){

        const res = *await* Order.deleteMany({\_id : {$in : *customer*.orders}})

*//we're all the ObjectId's in customer.orders array will get deleted.*

        console.log(res);

    }

*//You don't have next() here as it's post operation*

})