1 - Introduction & Setup

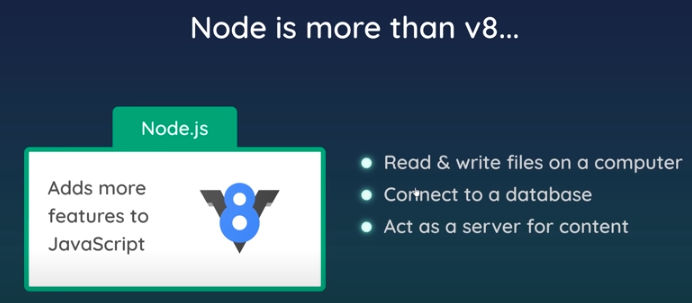
* Course files : <https://github.com/iamshaunjp/node-crash-course>

**What is NodeJs?**

* Node allow as to run JavaScript on the server side.
* V8 engine is written in C++ by Google, and it compiles the JavaScript into machine code at runtime(as a service). However, you can not run JavaScript outside the browser hence V8 is not available outside the browser. Then that is when “Node” come in to the play
* NodeJS is a program written in “C++” that wraps the V8 engine. i.e the v8 engine that is available in browser also lives in NodeJs too.
* Hence Node is written in C++ it can run directly on our computer.



* Node is just more than an v8 engine, unlike JS used in browser for interactivity only. And hence we use JavaScript outside a browser in Node, we lose access to DOM and we can’t interact html element anymore.



**Role of NodeJS**

* Runs JavaScript at the backend or server-side, handling request coming from browser.

**Installation**

* Check if you already have node on your machine: # node –version //if you see version number you have node installed



* If you don’t ger version number, it means you don’t have node installed on your machine
* Goto [https://nodejs.org/en/download/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbEM3MnVuNFVfbkh3VGExa1FITEc0TEtMc0UwUXxBQ3Jtc0trOUNRbGowVHZmVTVHU3VVU3dmS3B1ZzB4bUhyM0JVb0xtcVM2bk9USmY1amdKMVktVTNyTFFkdXJJYk1FX3RJTGdQSGxvdTNnQXhFSTQ5ZkdsdWZxV0FEOWR0MGlvV0IxWUIxWWZGaWtWbkNTbnhIRQ&q=https%3A%2F%2Fnodejs.org%2Fen%2Fdownload%2F&v=zb3Qk8SG5Ms) download node and install latest LTS version and run “*node –v*” again to make sure you have installed node.
* You can run Javascript code directly in the terminal :
  + #node > will create a repl environment that allows you to run JS code in the terminal
  + #> 5+5 or #> const name=”John Doe” , etc…

**Setup a project**

* Switch into your directory you want to create the project then create project folder:
  + # cd Repo/Node/



* + Then switch into the project folder and open it with your text editor



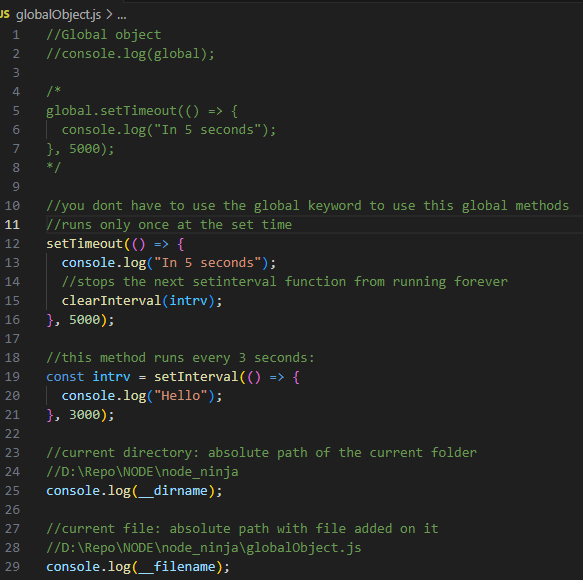
* Running Files with nodeJs:
  + To demo this create a file called “test.js” will console.log ;
  + To run this file, open the terminal and make sure you are in the current directory where the file exists and run the following command



# 2 - Node.js Basics

**The Global object**

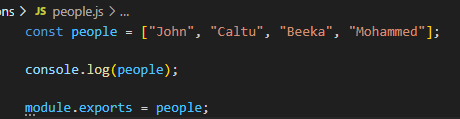
* Are methods and objects that are available for us by default in the node. Like “window” object is available for us in the browser when we Manipulate DOM elements.
* examples:



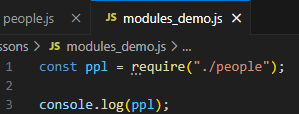
Note, global objects In the browser like window & (document.querySelector()) are not available in the node.

**Module and require**

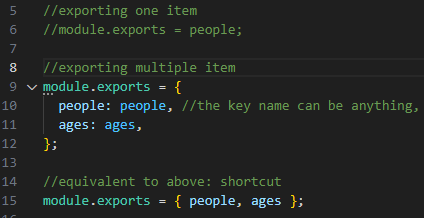
* Allow as to split our code into different files and export & import to/from those files, which keeps our code modular and reusable, much easier to maintain.
* For example, if we have one module that contain data about a people called “people.js” and want to import and use in another file called “modules\_demo.js”.
  + We first add “*module.exports=<filename>*”; in the file we want to export



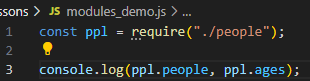
* + And we import it in the file we want to reuse it using “*require() method*” which make the content of the imported module available here for us to use.



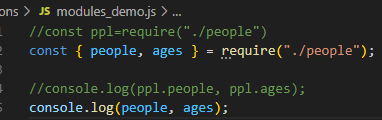
* + What if you want to export multiple things from module at one time



You can import and use them as follow :



But the corresponding importing method would be destructuring while importing to extract data from the object we have imported for use.



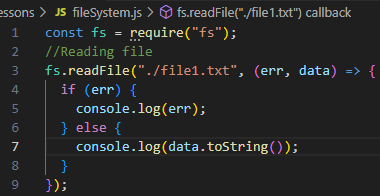
* Node comes with core modules by default which we can import via “require()” and use them in our applications. For examples: filesystem, http, url, os, etc.…

**File System**

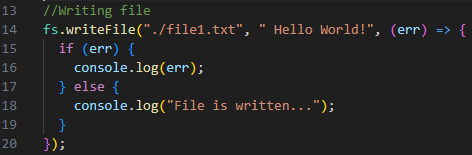
* Allow as to read, create, modify or delete files.
  + To use filesystem module, you need to import them to your NodeJs file.



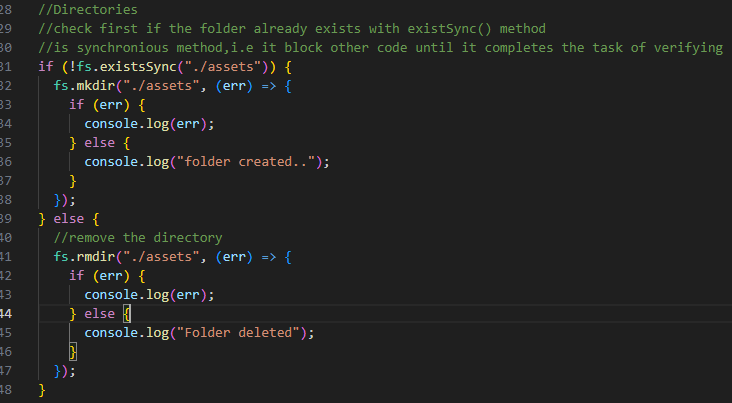
* **Read file**:
  + We use readFile() method on the imported filesystem object. readFile() takes two argument the 1st is relative path to the file we are going to read and the 2nd is a asynchronous(takes some time to complete) callback function that fires when file reading is complete. And the callback function in turn takes two arguments: one the error and 2nd the data we read from the file.



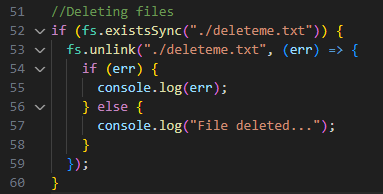
* **Write file**:
  + We use writeFile() method of filesystem module which takes three argument; the 1st the relative path to file we want to write to, 2nd the content of file we want to write and the third the callback function that runs when the action of writing complete.
  + Note the callback function is need, because the witeFile() or readFile() are ***asynchronous*** functions which takes some time to complete.
  + Note, *writeFile()* replaces the original content of the file with new content provide, and if the specified file ***doesn’t*** exist; it will create new file & adds the content in the 2nd argument to it.



* **Directories**:
  + mkdir() creates specified directory at specified location
  + Note, use existSync() method to verify if the directory you want to create already exist before creating one, because if you try to recreate the existing folder you will run into error. The existSync() method is a synchronous method, which means it block other code from running until it finishes its own task(which is checking if the folder specified already exist).
  + Similarly you can use the “ rmdir() “ method to delete a directory

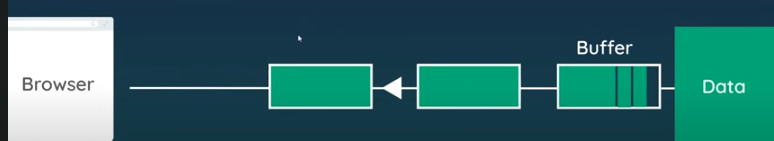


* Delete file:
  + We delete file using the “unlick()” method of filesystem module,

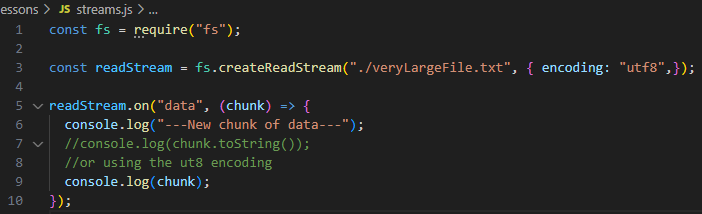


**Streams and Buffers**

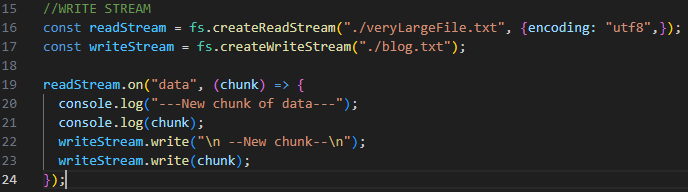
* Used to read and write to large data from/to files.
* **Streams**:
  + Allow us to start using the data, before it has finish loading(before it has been fully read). i.e. Hence reading large file takes time; instead of waiting until the file is fully read, we can fetch small batch of data called “buffers” at a time and we start working on the arrived batch immediately until the next batch arrives.
  + For example, when you watch movie on Netflix or YouTube; a little bit of data is sent to the browser, so that you can start watching immediately without having to wait the for the whole video to load.



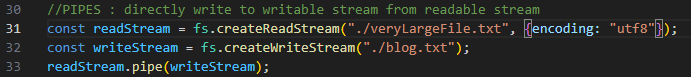
* + Lets see this in practice
  + **Read Streams**:
    - We use the “createReadStream( ) “ method of the files system to create read stream which takes two argument the first specifies from where we read & and the 2nd being encoding that converts the bytes to readable strings of character.
    - then we start reading from that stream when an event is triggered by event listener on arrival of every new chunk of data(buffer) which fires a callback function that allow us access that chunk of data.
    - Note, When you get buffers the value is bytes and you need to convert it to readable format via “toString()” method or by passing ***encoding*** object(which converts the bytes into ‘utf8’ character while fetching/reading data) as second argument when creating read stream.



* + **Write Stream**:
    - We use “createWriteStream()” method of filesystem to create a write stream which allow us to read from file using “ceateReadStream()” and then write to file using “write()”.
    - “createWriteStream()” takes two argument which specifies the file we write into and

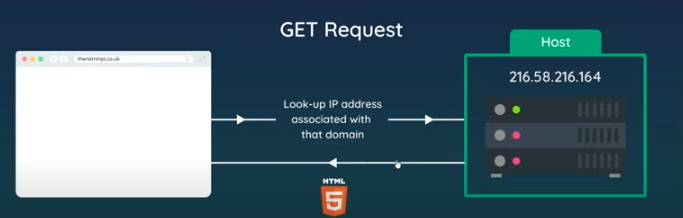


* + **Pipes**:
    - Allows you to directly write to writeable stream from readable stream. Shorthand notation that shortens the process of writing to file directly from readable stream to writeable stream.
    - i.e it open read open readStream reading data every time it gets a chunk of data then pipes that into writesteam.
    - It simple does what “write stream” do in shorter steps.



# 3 - Clients & Servers

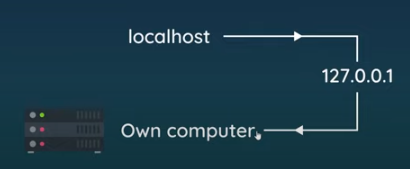
* Servers listens to requests coming from clients (usually browsers) and responds with particular data or file corresponding to request. usually, servers return a webpage.
* **IP Address and Domains**:
  + Every computer connected to internet have unique IP address to identify it.
  + So to access particular website hosted on a server, we need to know the IP address of the host server and type that IP address on the browser to connected to the server.
  + However, IP addresses are series of numbers & are difficult to remember, so we use “domain name” which masks the IP addresses with a memorable name. then when we type a domain name into the browser, the browser will look the IP address associated with that domain and use that IP address to connect to the server on the host computer where your website is deployed.
  + **Get Request**: The type of request where you type something in the browser and hit enter is called “GET” request, it is made every time we go to different webpage either by “link” or typing into browser.



* + **POST Request**: type of request used to send data to a server from web form or something.
  + there are few more types of requests you can make.
  + This communication between client and server is made possible via “HTTP “(hyper text transfer protocol”, it is ***set of instructions which dictates how communication occurs***.

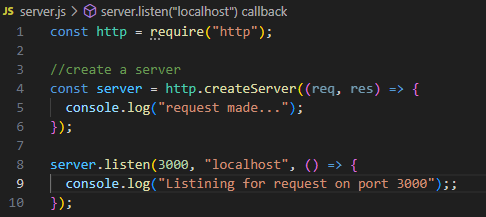
**Creating Server:**

* + With Node we can write code that creates a server and listens for requests coming from browser and respond to it with appropriate data and decides what to send back to the browser.
  + Unlike other languages like PHP where we don’t have to create the server manually, (they use other tools that manage that for us like “apache”); In Node we manually create our own server witch lives on the backend of our application.
  + Let’s create Server using NodeJS:
    - Create a JavaScript file and name it whatever you like(here we call it “server.js”)
    - Import/require a node core module that imports the “HTTP” object, which allow us to create a server using its method called “createServer()” method, which takes a callback function which runs every time the request comes in to our server.
    - The “createServer() “ method provide as two objects: the request and response objects.
      * The request object is fully loaded with information about the request, such as ‘URL’, type of request GET/POST etc
      * the response object allows us to send response to the user
    - For the server to work you need to invoke “Listen()” method on the “createServer()” method which specifies the host name and on what port the server listens for requests.
    - The “listen()” method takes three arguments,
      * port number: on which it the server listens
      * host Name: //default value is ‘localhost’
      * and a callback function that fires when server starts listening
  + **Localhost and Port Numbers:**
    - **Localhost** is like domain name on the web, which takes as to specific IP address called a **loopback IP address(127.0.0.1)** and point back directly to your own computer. That means when we are connecting to localhost domain in the browser; the browser is actually connecting back to our own computer which is then acting as a host for our website.
    - So the hostname “localhost” means listen for requests coming to our own computer. And this is how we use our computer as a host during development.



* + - **Port Number** are like “doors” into a computer through which internet communication can be made to different programs. i.e. they represent specific channel, gateway or port on a computer where a software or server communicate through.
    - All software’s on your computers that are connected to the internet, receives and sends data (such as skype, mail clients…) on different port numbers to keep information separate from one another.
    - So our server also needs a port number to communicate through. And this port number should be unique(it is not used by another program; so it cannot clash with other program on your computer)
    - For example, let us assume our app is running on port “3000” on the localhost domain, to access this app on the browser we use the following syntax: *localhost:3000*

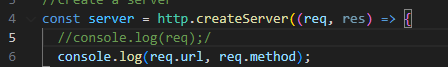
The server we created looks like this sofar



* + ***Note you can not see the “console.log” message on the browser console, you only see it on terminal; because the code is running on the server not on the browser.***

# 4 - Requests & Responses

* The request object(the first argument callback function in the createServer() definition) contains all the information about the request made to the server such as URL, request method, any thing you think about request can be found from this object.

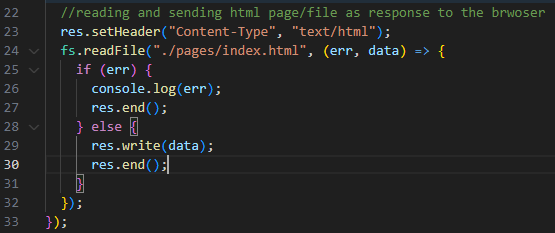


* Response object (2nd argument) used to send response to the browser.
  + Formulating response: when you create response back to the user:

1. first define the header which specify what kind of response you are sending back
2. then send the response and
3. finally end the response: sends all the response back( signal the response to be sent )
   * ***Response header***: gives the browser information what kind of response is coming back to it. For example, what kind of data we are sending back( text/html/json etc)
     + Headers also allow us to set “cookies”

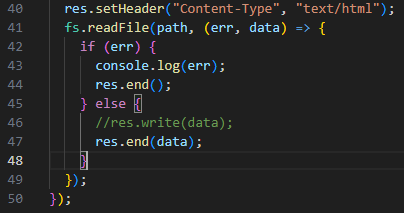
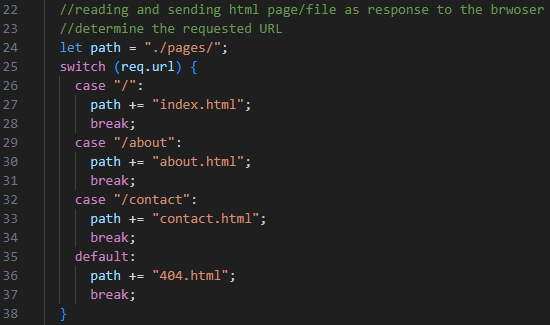


* + Retuning HTML pages:
    - To return html page you use the core filesystem module



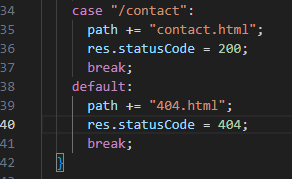
**Routing**

* A way to return different response for different requests. i.e. Figure the URL of the request and then depending on that URL send back different HTML file or resource.



**Status Code**

* Is part of the response & describe the type of response sent to the user/browser.
* i.e. it describe how successful is the response
* Some of common status codes:
  + 200- Ok
  + 301-Resourse moved
  + 404- Not found
  + 500-internal server error
* Status code ranges:
  + 100 Range: - Information response
  + 200 Range: -success codes
  + 300 Range: - codes for redirect
  + 400 Range -user or client error codes
  + 500 Range- server error code
* Example



**Redirects:**

* + Allow us to manage renamed route. for example, if the you change ’about’ page to ‘about-me’ where other webs have already referred to the original ‘about’ page, they will get error 404 because it is renamed to ‘about-me’ so we use redirects to manage such errors.



# 5 - NPM

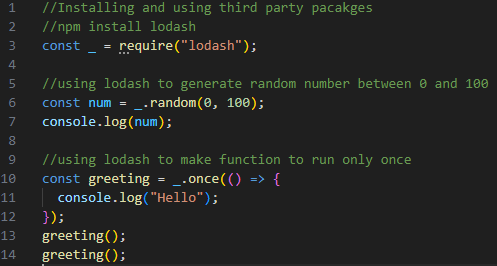
* NPM(Node package manager) is automatically installed when you install node on your computer .
* it is a tool used to install, update or remove node packages to our computer/or our projects to use.
* **Installing packages globally**:
  + For example, lets install “nodemon” package which allow us to create ***live reload server*** during development which automatically restart the server when changes are made to the file to reflect those changes for us. i.e. nodemon watches change in our file and automatically restart.



* Installing packages locally to specific project:
  + “ package.json “ is a json file that keeps track of any packages we install locally to our project and other project details( like project specific scripts).
  + If you want to use third party packages you should create a ‘package.json’ file in your project by running ‘npm init” which prompts series of question about our project( such as project name, version, description, entry point, test commands, etc.) or provides a default value in brackets if you don’t provide any and initializes/Creates package.json file within the project for us.
  + Along the pacakage.json file you will get the “ package.lock.json” which keeps track of different dependency versions we have installed in our project.
  + so we install third party packages/libraries/utilities/tools which help us implement extra features into our project we follow the following syntax:



* + - this will install & put it in “node\_modules” folder & register the installed package/dependencies in the “package.json”
    - then to use the installed package, you first import/require it to the file & then implement it



* + - Note, all the different files, folder and dependencies needed for that package is kept in “node\_module” folder.

**Dependencies**

* Npm allow us to share project code by only uploading your code and leaving out the decencies in the “node\_mdule” (for example to github) and someone who download your code can re-install all the dependencies you use by only running “npm install”, because node will look the dependencies needed by the project which are listed in the package.json file and recover and re-install them for you.

# 6 - Express Apps

* Is a node framework that helps us to easily create node website by managing routing requests, server-side logic and responses.
* It simplify things you do with node core modules and allows us to write clean code.
* Using express:
  + First install express to your project



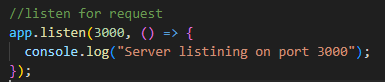
* + **Creating Express app**:
    - Create a JS file called “app.js”
    - Import /require express

//

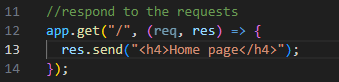
* + - Setup express app, by invoking the returned function when we import express on the app variable to create instance of express app



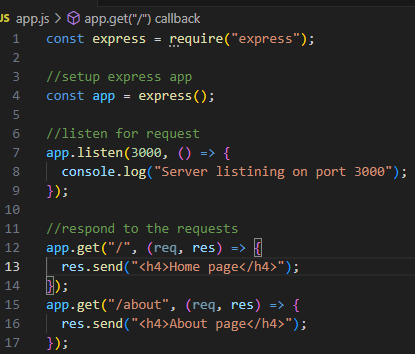
* + - Start listening for request



* + - Respond to those requests:

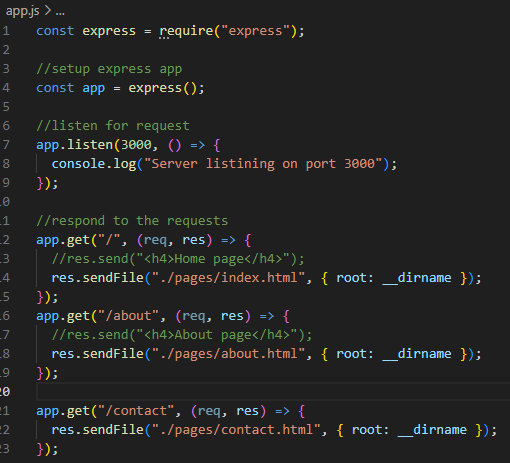


* + - “ res.send()” method of express have similar task like node “res.write()/res.end()” but have more functionalities. It first infers the type of content we try to send and automatically ser content header, also infers the status code which minimizes what we have been doing with vanilla node modules. i.e. it determines the content type, status code and send response at same time.



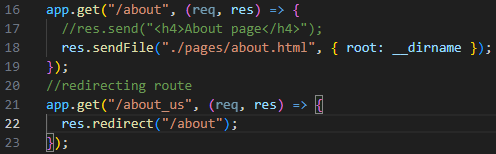
**Routing and HTML**

* We use sendFile() method to send an HTML file as a response to request; which takes two arguments the first being the path of the file and the second an object which specifies root path of the project(what is the file being sent is relative to; otherwise it will take the absolute path from the computer root(C:/) instead of the project root)

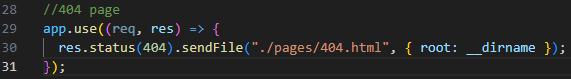


**Redirects and 404**

* In express to redirect a request to route we have changed/rename, we create a separate route for new route and we set a callback function which fires when request for new route comes and redirect to the old route.

//line 21-22

* To respond to unavailable page(404) we use a middleware that fires for each incoming request when other routes are exhausted and no match is found for request URL( so it is usually defined at the end of all routes, if it is defined first the server does not look for other routes past that point/after )



# 7 - View Engines

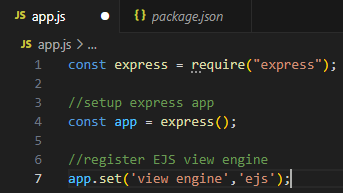
* Allow us to inject dynamic data in to an html file.

**EJS**

* Is a view engine or template engine that you can use in the express app to dynamically populate html file with data and allows you to integrate JS codes to the html template.(similar to react)
* Steps:
  + Install EJS to your project via npm



* + Register the view engine to the express app, with set() method.

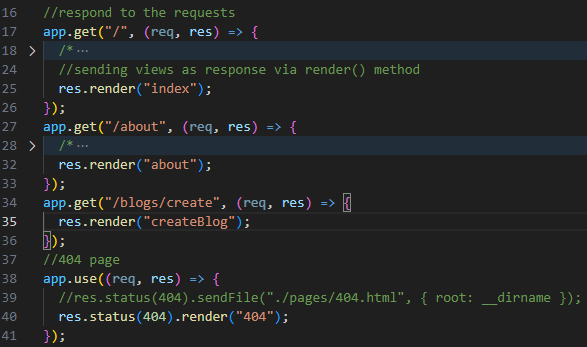


**Set()** method lets us configure application settings

* + Specify where the views are stored(by default express will find them in ‘views’ folder unless you specified otherwise as follows)



* + Create your views with “.ejs” extension in the ‘views’ folder( example: index.ejs),
    - an “ejs” file is similar to html file except they allow you to dynamically inject the content with data you want
  + Render the views corresponding to request as response with render() method for each route .
  + Note when rendering views you don’t have to specify the ‘.ejs’ extension of the file, hence you have registered the type of view engine you use to express app at the beginning.



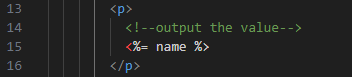
**Passing data into views**

* + We use ejs tags(looks a bit like php tags) which takes a JavaScript code to pass data to views..

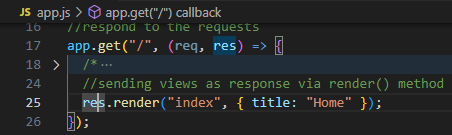


Note when you use only the opening “<%” and closing “%>” to define your code that code only runs on the server and you do not see the result on the scree/browser.

* + To see the resulting code of JS in the browser add the equal sign in next to the opening tag as follow



* + To pass data from backside and output it in browser we follow the following steps:
    - For example; to pass the title of the page, we use the data as a second parameter to the response. render() method as an object which is sent to the ejs file(the 1st argument).

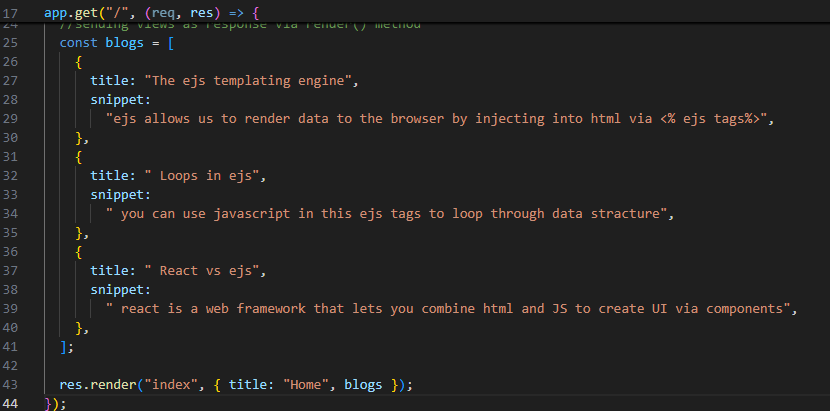
//app.js

This allow us to access this data (the title of the page) in corresponding ejs file(index.ejs).

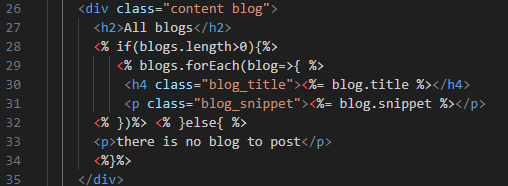
* + - So to output this data,simply reference the data we just defined in ejs template tag:

//views/index.ejs

* + - We can also loop through array or object and render the data to the browser. To do this first define the data structure and pass it to the object(2nd argument of the render()

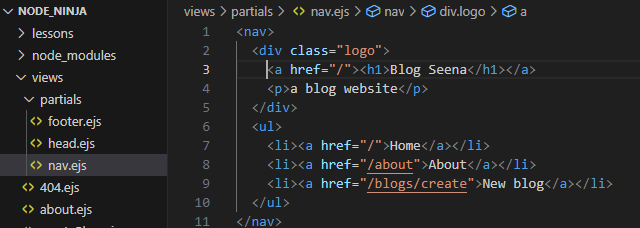


Then go to the ejs file and write the JS code that loop and display each values



Note, how we are switching between ejs tags, to write JS code which runs on the server and to display the resulting value. (we even use give “}” its own ejs tag , b/c ejs need to know that is a JavaScript code)

* **EJS Templates**
  + EJS templates are processed through the EJS view engine on the server.
  + i.e. our view files live on the server and when we want to render them to the browser that view file is passed into EJS view engine to be processed. Then the view engine looks for any kind of dynamic content, variables, loops, conditionals, etc. and calculates the resulting HTML code and then generates a valid html page based on template we created; and then renders that html page with resulting data inside it to the browser. This process is called server-side-rendering.
* **Partials:**
  + Allow us to create a reusable segment of code(for example, navbar, header, footer…) that can be used by multiple views/pages, and because of this shared feature you don’t have to change all the code in every page, you just have to change at one place and those changes are propagated to wherever that “partials” is used.
  + Steps:
    - Create a “partials” folder inside “views” folder and in this folder create your partials template

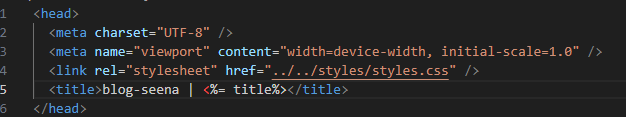
//views/partials/nav.ejs

* + - Then to use this partial template in your view/page; use the “include()” method which takes the relative path of the partial template and apply that partial at place you want to be used.
    - Let’s reuse this nav in “index.ejs” view. This is done by writing the following code in section where we want to place our navbar



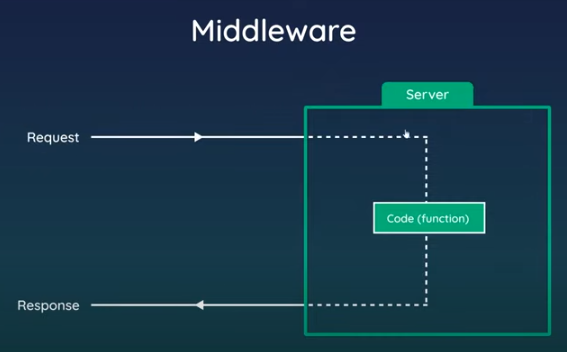
Note we used minus “-“ sign inside the ejs tag instead of equal(=) which we have been using so far to output values to browser. This is because equal(=) sign escapes special characters but we need string values in include() instead of raw html for the path/ URL of the partial.

* + - This allow us to update one file(the partial) instead of manually going through all the file and change each file you update the partial and all the views using the partial get automatically updated.
* **Adding CSS**
  + Create and add style tag in “head.ejs” partial

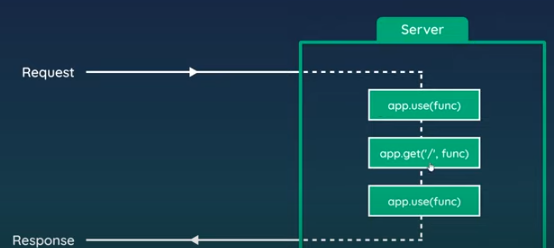


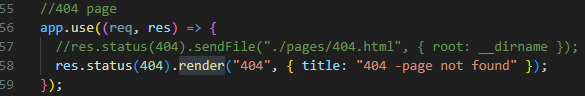
# 8 - Middleware

* Are any code that runs (on the server) between getting a request and sending response.



* We use a “use()” method to run middleware code.
* The functions that run on request handlers(like GET) are essentially middleware, except they run only in response to the routes targeting them(for eg, get runs only in response to get request to fire function and send response), however use() methods run for every type of request to all routes including post request.



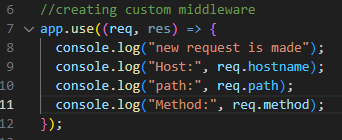
* Middleware run from top to bottom until we exit the process or explicitly send the response to the browser. Once the response is sent, middleware that comes after the point response sent are skipped/not executed. Therefor **order of middleware is important** to specify how it runs.
* Middleware Examples:
  + Logger middleware: to log details of every request
  + Authentication check middleware for protected routes
  + Middleware to parse JSON data sent from post requests
  + Return 404 pages
  + …
* Creating custom middleware:
  + For example we have created middleware for 404 page, and it runs as long as there is no response is sent back(i.e. where there is no matching route found and sent back to browser).

//app.js

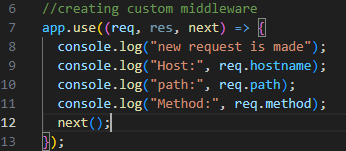
Note, middle runs top to bottom looking for matches, until it reaches the last 404 code and runs and send response(in this case 404 page), if there is no match is found while it sniff through the stack of codes top to bottom. Also note this code should be at the bottom, however if you place it at the top, all codes after it will be skipped even if the match exist for the requested route, because this middleware will automatically send the ‘404’ page when code reaches it.

i.e. it acts as catch-all/default clause which run whenever there is no match found

* + Now lets create a middleware that logs some details out to the console for every request
    - Lets define it at the top so that it can run for every incoming request



* + - When you make request this middleware will run & log the information, but the browser hangs and doesn’t know what to do next so we have to explicitly tell it to move on to next function/middleware(so to handle this we use the next() method.)
* **Using next()**
  + Pass it as third argument in the middleware callback function it and invoke it at the end of the function; so that the next middleware can start running immediately when the current finishes.

 //notice function definition and line #12

* Note, once response is sent to browser, middleware’s coming after it do not run unless other event/ request is made.
* **Third party middleware:**
  + Are a ready made middleware that we can implement them in our project.
  + To demo lets use “morgan” middle ware which logs information like our custom middleware we created earlier.
    - First install it in your project



* + - Require it in your app



* + - Use it in your app; notice we don’t pass callback function like when we create custom middleware, because its code is already coded for us, we just simply invoke it.



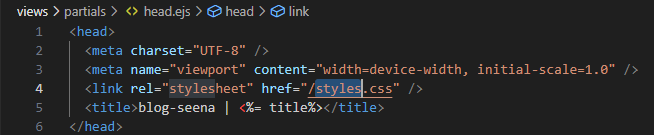
Now run the server & refresh the browser and check your terminal for the result

**Static Files**

* + hence the server automatically protects our files from user we can not directly access the static files (eg. Css, images) from browser. So to allow browser access this files we have to specify should be allowed to be accessed using use middleware shipped with express known as “static middleware” to serve static files like css and images. i.e. specify what files should be public
  + this static() method specifies where the public files are located(for example “public” folder) and anything inside that folder will be made available as static file for user.



* + then create the static files and link them to your app. For example, create css file, place them in “public” folder which should be at root of the project. Then link the css file to the “header” partials. This should apply the styling to your pages/views. And you don’t have to provide the relative path of the CSS, b/c the server will automatically look for “public” folder in the root and get the style file, hence you have specified the “public” folder using the static() middleware.



//views/partials/head.ejs partial

Note even though the “header” partial is to deep in file structure we didn’t specify the path as (../.../public/styles.css)

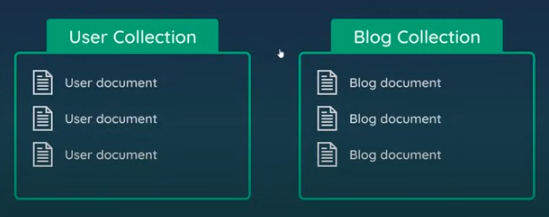
# 9 - MongoDB

* there are two types of databases the SQL and NoSQL based on how they structure data.
* SQL databases use tables, rows and columns to store records of data, while NoSQL databases make use of collections and documents.

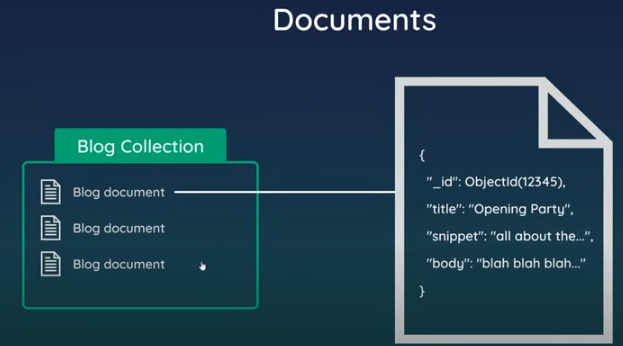


**How Does NoSQL database work?**

* They split up into collections, they are a bit like tables and each collection is used to store particular data type of data. For example, user collection stores user documents.



* Note each collection only stores one type of documents(either blog, user or author)



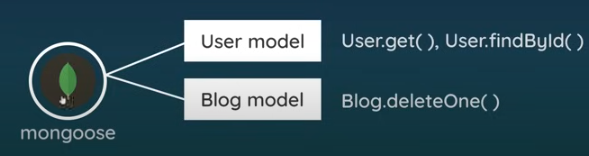
* Document is a bit like a record in the SQL database; each one represents a single item of data and have a format similar to “JSON” which stores series of “key-value” pair; and auto generated unique Id.

**MongoDB Setup and Atlas**

* You can install MongoDB locally or use cloud database which is already hosted for us.
* **Atlas**:
  + You can sign up for free and start using immediately at [**https://www.mongodb.com/cloud/atlas/**](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbV9OajRKX05udkNESzdpQllyS1ppWTVaTkFnZ3xBQ3Jtc0ttcm1oX1gzTXZBMXk1b1N4OVUwQUJVZ0tPVG9BTGdrQ0s1Y3JNQ0NDVzJ3NFBrYkpnMkpWX0pJU0w3YTFzakhGelNaVS1wRmwzajB4LUw3S0ZJTm1HaE9IbHdBM0wxdjBxZmlqaDh2cW9VQjVBZTJxUQ&q=https%3A%2F%2Fwww.mongodb.com%2Fcloud%2Fatlas%2F&v=bxsemcrY4gQ)
  + **Steps:**
    - Build Cluster:
      * When you first create atlas account and sign in you need to create cluster before start using. So choose the option for cluster (free or paid) in this case we use free option >> choose a provider and region(choose default ) >> give the cluster the name(or use default=cluster0) >>
      * IF you see “add current IP address” button on the home page click it to add the IP of your machine
    - Create Database and collection:
      * Under your cluster(cluster) go to collections >> + Create a database (btn) >> give the database a name (“BlogDB”) >> give the collection a name(“Blog”) > if you want to have preference choose one(here we skipped) and then click “Create”
      * Note collection is where we store all the documents of certain type.
    - Create a user that access the database
      * Go to “database access” and create new database user by clicking “add new database user” button >> then choose authentication method (password) >>then provide username(“seenaBlog”), password(“admin”) and description >> give the user Role(read and write access to any database right) >> Add user
      * Note we use this username and password to connect to the database from our code.
    - Set a connection to the database:
      * Go to your cluster > > connect >>Drivers ( you will get a connection string copy it (mongodb+srv://<db\_username>:<db\_password>@cluster0.wokglty.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0) and store it on variable in the your projects “app.js” or environment variable.
      * And replace the *<db\_username>* with your username(“seenaBlog”) and <db\_password> with your password(“admin”) you have created earlier.
    - Connect to the Database:
      * You can connect using plain MongoDB API package & make queries to the database, but we use a “Moongose” to connect and interact with DB which is easier to use.

**Mongoose, Models and Schemas**

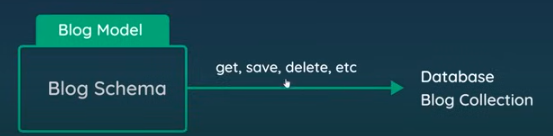
* Mongoose is an ODM(Object Document Mapping) library. That means it wraps the standard MongoDB API and provides us with much easier way to connect to and communicate with database. It does this by allowing us to create simple data models which have database queries methods to create, get, delete and update database documents.



* Mongoose does all the heavy lifting for us; it queries the correct DB collections for us based on the name of the model we use and performs the action required and returns a response.
* How to use Mongoose to connect to the database:
  + We create schemas and models first
  + Schema defines the structure of the data type/ document stored in database collection.
    - i.e. it describe property and property types



* + Model allow us to communicate with database collections, hence a model to represent the data resource. For example, if we create a blog model based on blog schema, then blog model will have both static and instance methods we can use to save, update, delete or read data from the blog collection.



**Steps to connect Mongoose to the database**:

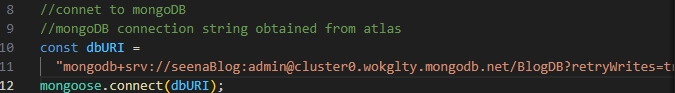
* Step 1: Installation and setup
* Install mongoose (b/c it is third party package i.e. it does not come with MongoDB)



* Require it in you “app.js”



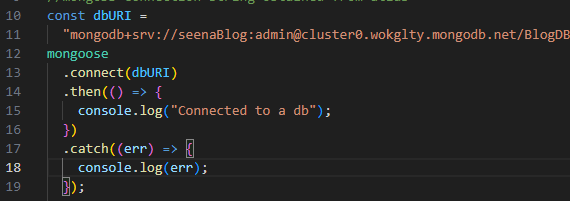
* Then use this imported mongoose object’s connect() method to connect to the database.
  + - The connect() method takes DB connection string

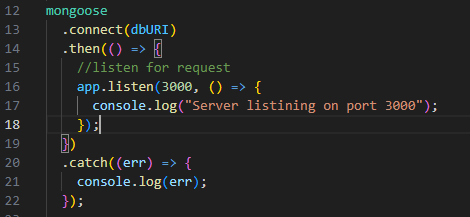


* + - Note change the database name in the connection string from ‘test’ to your <your\_db\_name> which is “BlogDB” in this example, not only the username and password



* + - Hence the connect() method is an asynchronous task witch takes time complete, it returns a promise that we can attach the then() method to it; which takes a callback function that fires when the connect() method completes and return a result after connecting to db.



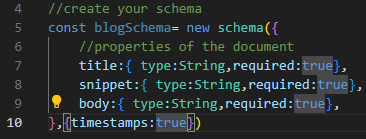
* make the server listen to the request after connection is established by placing the listen() method of the express in the then() clause of connect() method as follow.
* **Step2:** Create schema and model for our blog data
* Create folder for your schemas and models called “models”.
* Inside the “models” folder create a file called “blog.js” which defines the schema and models.
  + Require mongoose



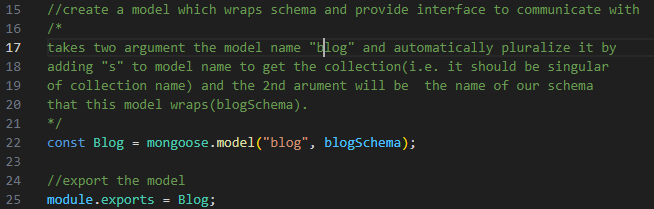
* + Get schema constructor function used to create new schema) from mongoose object



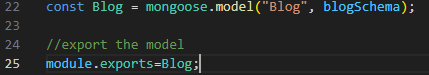
* + Note, schema defines the structure of you data/document we store inside collection, and it is the thing model wraps around.
* Create your schema which takes two arguments the 1st is an object which describe structure of the documents we want to store in our blogs collection and the 2nd object option that automatically generates timestamp property for us on the document creation.



* Create model base on this schema we just created.
* Note, schema defines the structure/properties of the document while the model surrounds the schema and provide us with an interface by which we can communicate with a database collection for that document type,
* To create a model you invoke the model() method of mongoose which takes the name of the model as first argument (“Blogs”) and the second argument the schema that this model will based on.



* Finally export the model so that we can use it in our project

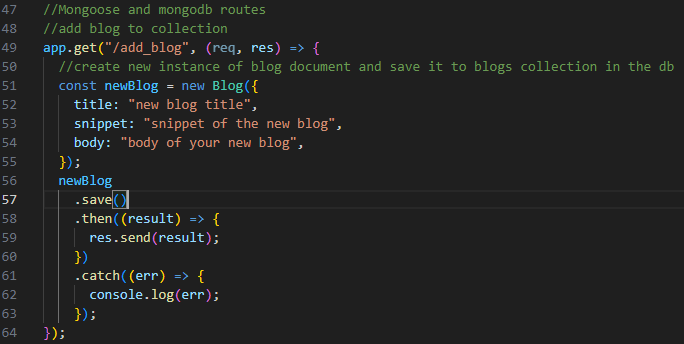


**Getting and saving data**

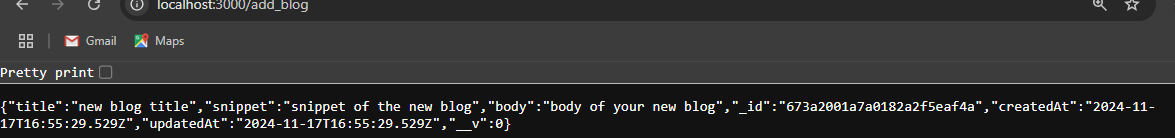
* Now we have created the schema and model, lets interact with the db. So go to “app.js” and create routes to interact with the db.
* First Import the model to “app.js” and create routes for each type of request; and in this request create a new instance of the document & then save it to the collection to the db.



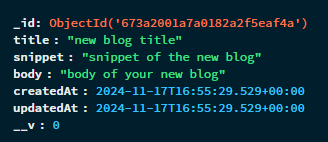
* Then call a save() method on it to save it to the database. It is asynchronous task which takes time and when it does it returns a promise on which we can attach a then method to perform desired action.



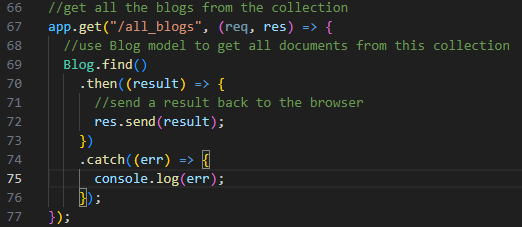
Go to the browser(<http://localhost:3000/add_blog> ) and check the result, if succeeded you will get a Json object that saved to the MongoDB that includes your blog data, auto generated id and timestamp of your record.



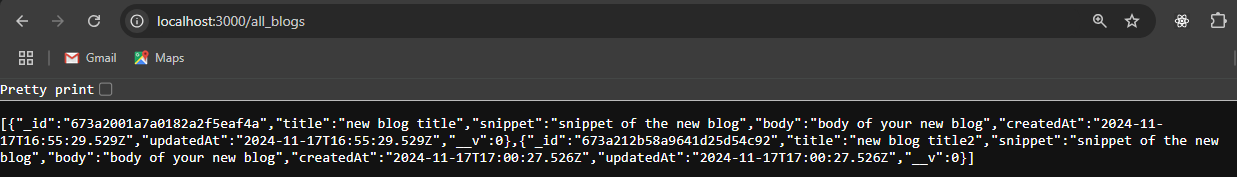
Similarly, if you go to the MongoDB and refresh your collection you will get similar result



* Use find() method on your Blog model to get all the blog documents from blogs collection in the db.
* Find() is asynchronous method and takes time to complete and provides a promise to attach then() method and when done returns a result we can act upon.



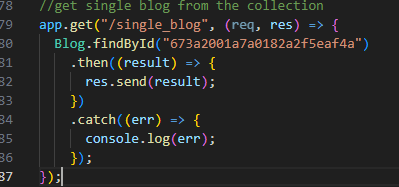
* Similarly check the result on the browser: <http://localhost:3000/all_blogs> //you get array of blog objects



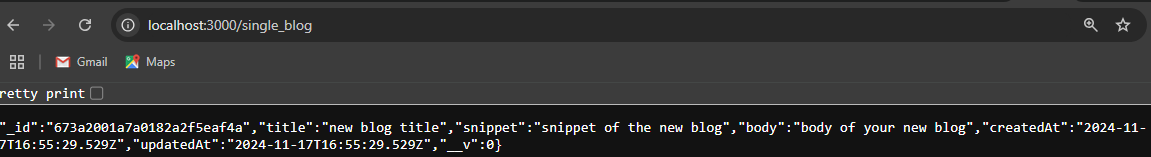
The video reference

<https://www.youtube.com/watch?v=bxsemcrY4gQ&list=PL4cUxeGkcC9jsz4LDYc6kv3ymONOKxwBU&index=12>

* Finding single blog:
  + We use findById() method on the “Blog” model to get single blog

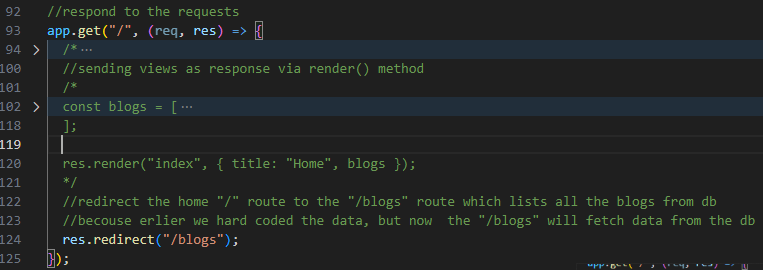


Result on <http://localhost:3000/single_blog>



**Outputting documents in views**

* Now lets return to our project and render the fetched data in our views when the corresponding route is requested.
* Note, since we need the previous routes for reference, we will create new once, and we can compare. So we have redirect the “/blogs” to “/”. And also looping through data in the views is already done before when we covered EJS templating engine, so we don’t have to redo that we jus maintain it.



* List all the blogs:
  + And the “/blogs” route will look like:



* + Now go to : <http://localhost:3000/blogs> and check the result

# 10 - Get, Post & Delete Requests

**Request Types:**

* **Get** request to get a resource (webpage, Json,)
* **Post** requests to create new data (e.g. New blog), usually via web forms
* **Delete** requests to delete data (e.g. delete a blog)
* **Put** requests to update existing data (e.g. update a blog)
* Note we can use the same routes for different types of requests: for example, we can get all the blogs on “/blogs” route via GET request and when we submit form on “/blog” route we send a POST request.

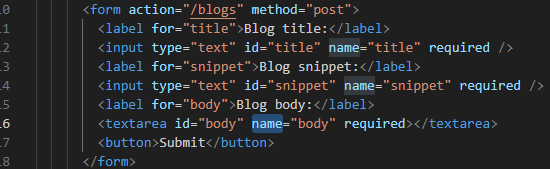


* These requests make CRUD application
* Hence we covered the “Get” requests earlier we don’t do it again, we look post, Delete and put(update)
* **POST requests:**
  + Consider when you create new blog on the “Create blog” form and submit the form, we fire a POST request to server with all the data filled on the form; then on the server we handle this post request by taking all that data, create new instance of a blog document using a Blog model and add this data to that instance and save that document to the database.
  + Steps:
    - Send the post request by submitting form data cy clicking button and using the “fetch API” or other asynchronous library from JavaScript or directly from the webform. For this example, we use the 3rd option sending the post request from webform itself
      * Go to the “createBlog.ejs” view, inside <form> tag add “action” attribute with the value(URL) to where we send this form request to, and specify the request “method” which is “POST”

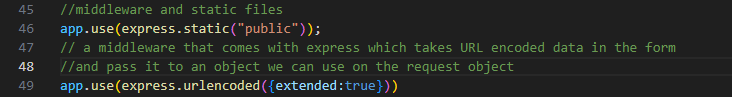


This means, when we click the submit button post request is send to the ‘/blogs’ URL.

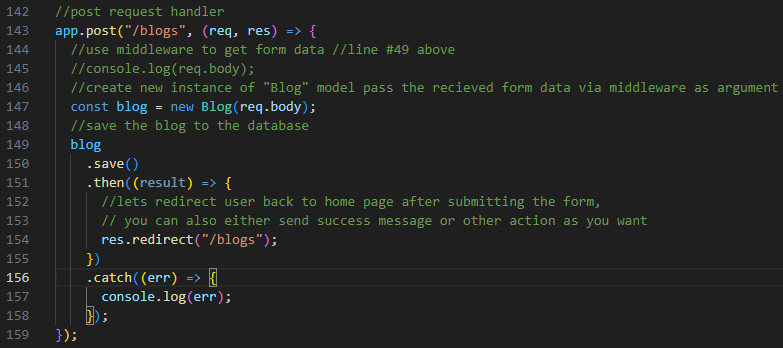
* + - * Add “name” attributes to the input fields in order to send the form data to the server during submission. So that when we access the form data on the server we use this “name” property values.



* + - Then handle this post request on the server
      * On the “app.js” go to “/blogs” route and create a post request handler. Inside this route use a middleware(line#49) to collect the data that comes from the form body,



* + - * Then create new instance of “Blog” model and pass the data(object) received from the form via middleware as argument.
      * Then save this blog object to the database via save() method



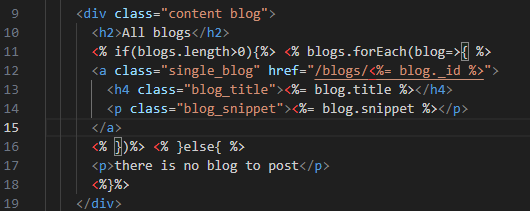
//submit a create blog form and see the result on the home page and MongoDB as well if the data is added and the post request worked

**Route Parameters**

* Are variable parts of a route that may change a value. For example the “id” can change depending on the blog we get

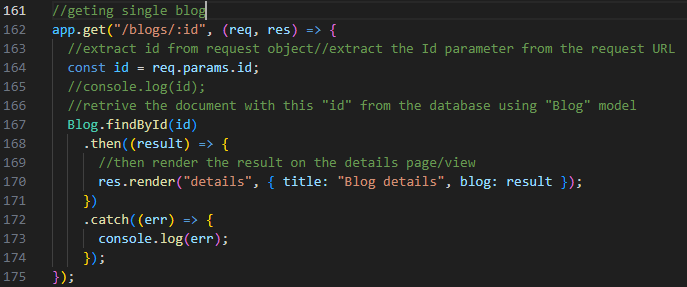


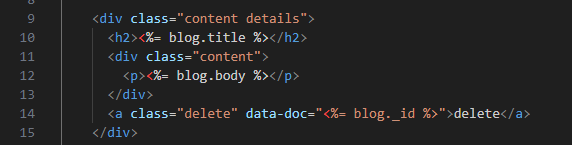
* In our project, to get id of particular id lets put each of the blogs on the home page in the link tag so that when that specific blog is clicked, we will get its corresponding id. So that the mongoose can convert this id to string and we can do the remaining request types( delete and update a blog)
* So lets modify our “index.ejs” view as follow

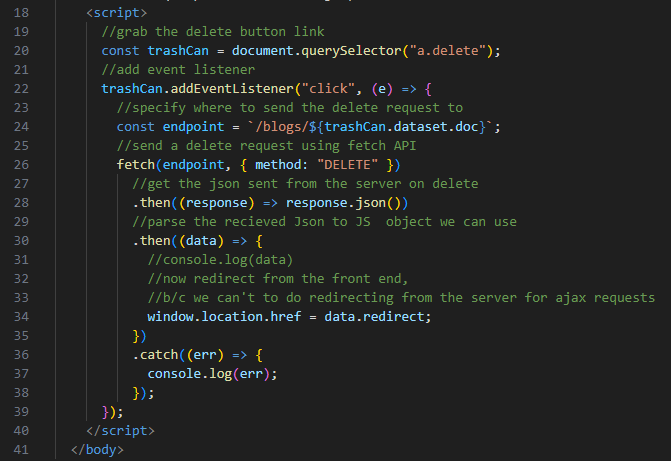


//each blog will result in /blogs/id when clicked because of the link tag

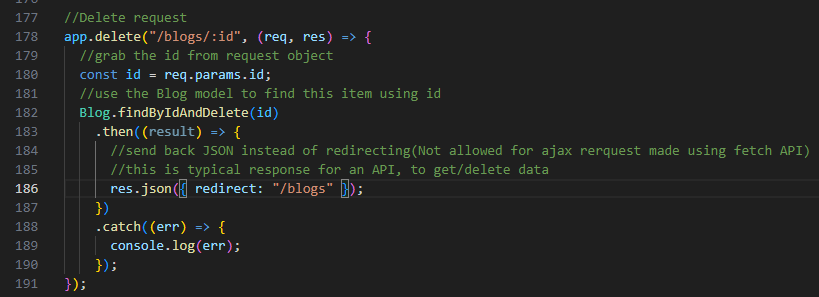
* Handle the request by extracting the “id” parameter from route URL in request handler
  + Go to “app.js” and extract the “id” from request object vie GET request
  + Note use “:” in front of route parameter to extract it, it if you omit it, you will get the variable itself (“id”) instead of the actual id string value.
  + Then retrieve the document with this “id” from the database and render it to the “*details.ejs*” view



* + “details.ejs” view, not the “blog” object is send on the line#170 of “app.js”
  + 
* Delete Requests:
  + In the details.ejs page/view, Write frontend JavaScript code that removes the blog from the view when delete button is clicked

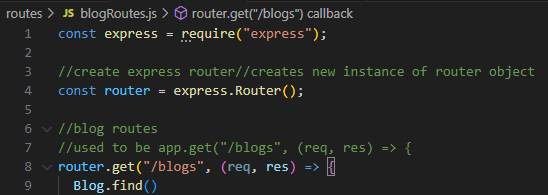
//details.ejs

* + Now go to “app.js” and handle this delete request using “findByIdAndDelete()” , then inside the promise / then() method, hence this is an ajax request made using the fetch Api, rather than webform(get blog request above), in node we cannot use a redirect as a response, we can able to send a JSON or text data back to the browser. b/c when the redirect is received in front end JavaScript we just created in <script> tag, the received data is a URL property where we want to redirect to and that is done on frontend( we can’t do it on the server b/c this is an ajax request)



# 11 - Express Router & MVC

* Express router comes with Express and we can use it to manage all of our route more efficiently.
* With express router we can split our routes into different files and manage them in small groups of routes that belong together which makes the app more modular and easier to maintain.
* Steps:
  + Create separate folder to hold your routes; call it “routes” in the project root
  + Then create JS file for each route types that can be grouped together(for example “/blogs” route, so cut and move them to this separate file named “blogRoutes.js”)
  + Hence the express app variable “app” is not defined in the “blogRoutes.js”, require the express package in this “blogRoutes.js” file and create new express router and attach the request handlers to the router instead of the “app”

//blogRoutes.js

* + Then move the “Blog” model import from the “app.js” to “blogRoutes.js” because we are not using it inside “app.js” anymore rather it is needed in the “blogRoutes”



* + Then at the bottom of the file export the router

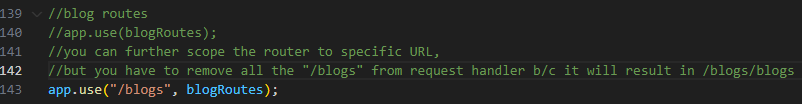


* + Then import it in the “app.js” and use it inside our app



//app.js

* In general express router allow us to extract routes into different files(like mini app) then import them in our app and use them like middleware are used.
* You can even scope these routers to specific URL. as follow:



* + This will apply this “blogRoutes” only when URL begins with “/blogs”
  + But for this to work, you have to remove all the “/blogs” route from all the ***route handlers;*** otherwise, you will be adding extra “/blog” URL parameter to the route which results in “/blogs/blogs”.
  + i.e. when you scope the router like we did here, you no longer need “/blogs” route in the request handler function( remove them and place the root route only). so change all the routes in request handler function from this  to 

**MVC Basics**

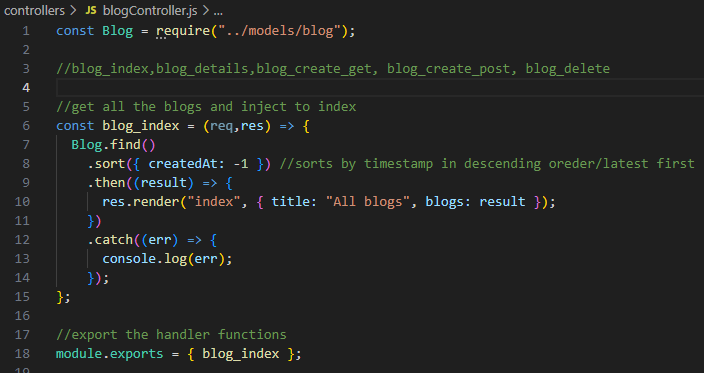
* stands for Model, View, controller
* MVC is a way of structuring our code and files
* Keeps our code more modular, reusable and easier to read and maintain
* We have already seen views(in EJS and html templates and models(in mongoose schema and models)
  + Views are the UI made via HTML templates
  + Models are how we describe our data structure and we use them to interact with the DB( like the one we used in mongoose to model our data)
  + Controllers: forms the link between our model and our view. They are like the middlemen that use models to get data and then pass that data into the view



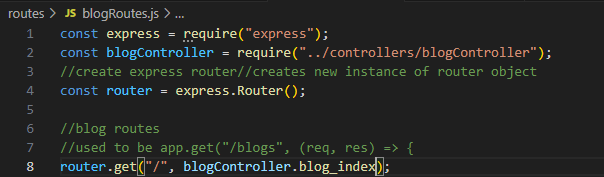
* MVC architecture allow us to split our code into different area of files, and each area of file has its own job to do.
  + So our route file matches incoming request and passes those request to the correct controller function,
  + then the controller communicates with model to get data into the view
  + then the view renders that data into its template and get it send to the browser

**Controllers**

* so we extract our route handler functions into a separate controller file and reference those controller functions from route file(“blogRoutes”).
* Steps:
  + Create a folder called “controllers” in the project root
  + Inside this folder create a “blogController.js” file and create the request handler functions inside this file. And use the MDN naming conventions for controller functions as follow
    - create “blog\_index()” function to get all the blogs and inject into index view
    - create “blog\_details()” to get details of single blog
    - create “blog\_create\_get()” to send back the actual form
    - create “blog\_create\_post()” to add new blog and
    - create “blog\_delete()” to delete specific blog
  + Import the “Blog” model into this file
  + Move all the implementation of the handler function to this file
  + Export all the functions



* + Import and use it in the routes file (blogsRoutes.js)



* + Do so for all the remaining handler functions,
  + This will separate all controller logic from the routes

Adding delete image

* Drop the trashcan icon in public folder and reference it from your “details” view

Handle 404 Page

* In the “blog\_details” function( in blogController.js) catch blog, instead of consoling the error respond with the 404 page if the route is invalid.
* 