**SECTION 31: Our First Brush with Node**

**What is Node?**

Node is a **javascript runtime environment**. Until recently, we could run javascript code in a web browser. Node is a javascript runtime environment that executes javascript code outside of the browser.

We can use the same javascript syntax we know and write server-side code instead of relying on other languages like python or ruby.

**Some Applications created using node:**

1. Web servers and writing server-side code for our applications
2. Command Line Tools
3. Native Apps (VSCode is a Node App)
4. Video Games
5. Drone Software
6. A whole lot more!

**Node REPL**

The javascript console in our browser follows REPL (Read, Evaluate, Print, Loop). Similarly, Node also has REPL. When we type node in the terminal, we use a REPL

**Node JS VS Client side JS**

Because Node does not run in the browser, we don’t have access to the window, document, DOM Apis etc.

Node comes with a bunch of inbuilt modules that does not exist in the browser, These modules help us in interacting with OS and files/folders

Just like **window** object in browser, we have **global** in node where functions like setTimeout, setInterval etc exists.

**Command for running JS files using Node**: node fileName.js

**Process and argv:**

Process object is a **global** object that provides information about and control over the current Node js process

Process allows us to get the version of the node, has methods that allows us to get input and write to the standard output, get current working directory (using **process.cwd()** command) and lots more

**process.argv** allows us to get the arguments from the command line interface.

Eg: the below code is saved in app.js file

const args = process.argv;

for(let i = 0; i < args.length; i++)console.log(args[i]);

and the below command is run

node app.js argument1 argument2

We get the following output

C:\Program Files\nodejs\node.exe

C:\Users\akundu1\Downloads\Web Development Bootcamp\nodejs\app.js

argument1

argument2

The first element is the path where node is installed, the second element is the path of the file that is executed, and the third and fourth elements are the arguments that we passed from node cli

//code to create html, css and js files in the folder whose name is passed as an argument

const fs = require('fs'); //file system is not present in global scope, to use this library, we have to use require

let folderName = process.argv[2];

try{

    if(folderName == undefined)console.log("Folder name undefined!");

    else{

        fs.mkdirSync(folderName);

        fs.writeFileSync(`${folderName}/index.html`, '');

        fs.writeFileSync(`${folderName}/app.css`, '');

        fs.writeFileSync(`${folderName}/app.js`, '');

    }

}

catch(error){

    console.log(error);

}

**Working with module.exports**

**module.exports object enables us to share javascript code between two files.**

Eg: we have two files math.js and app.js in the same folder as shown below

//math.js

const PI = 3.14149;

const add = (x, y) => x + y;

const mult = (x, y) => x \* y;

//app.js

const math = require('./math'); //remember to give the file path of the js file whose code you want to use

console.log(math);

when we run app.js using node app.js command, we get an empty object {}.

Note that **require** functions returns module.exports object of the required file. By default module.exports is an empty object. To include code of math.js, see below

//math.js

const PI = 3.14149;

const add = (x, y) => x + y;

const mult = (x, y) => x \* y;

module.exports.add = add;

module.exports.PI = PI;

module.exports.mult = mult;

//or

module.exports = {

    PI: PI,

    add: add,

    mult: mult

};

And now we can use the functions of math.js in app.js file.

**NOTE:** We can also require json files

const jsonData = require('./data.json');

**Requiring a directory**

If we require a directory, instead of a file, whatever the **index.js (Yes, the name has to be index.js)** exports will be exported by the folder

**NPM**

NPM stands for node package manager.

NPM is really two things:

1. A library of thousands of packages published by other developers that we can use for free.
2. A command line tool to install and manage packages in our node project.

To install a package, use npm install packageName

Eg: npm install express installs express package

When we install a node package, a folder called **node\_modules** and a file called **package.json** are added. node\_modules consists of the dependencies for the package and package.json consists information about the dependencies in json format.

npm install give-me-a-joke

This command installs give-me-a-joke package and to use the package, refer below

const joke = require('give-me-a-joke');

//here require is automatically going to look for the package in node\_modules folder

joke.getCustomJoke((joke) => console.log(joke));

**Adding global packages:**

Currently when we install a package, using npm install {packageName} command, we can use require(packageName) in our js file (WHICH is inside the directory where the package was Installed). Because node searches for package inside node\_modules which is in the current directory.

But what if we want to use the package outside the directory where the package was installed. Running the command require(packageName) in a js script outside the directory where the package is installed would give error.

For this reason, we can use global packages

To use global packages there are two steps to be followed:

1. To install packages globally, use **npm install -g {packageName}** (Here g flag stands for global).
2. To use the global package in a specific directory, go to that directory and run **npm link {packageName}.** It links the package to the current directory and now we can use the package using **require(packageName)**

We can see the directory of the global packages installed using **npm list -g** command.

**package.json:**

package.json is a file which consists of information like the dependencies used by the module, version, author, license, repository etc.

Whenever we make a new node project, use npm init command to create a package.json file for the node project. Now whenever we install a new package for our project, it would be added in the **dependencies** of package.json file.

Now whenever we share our node project, we often exclude the node\_modules folder. So to install the required packages, just type npm install command and what it does is it looks for the package.json file in current directory and installs all the dependencies which are there in package.json file.

**SECTION 33: STARTING SERVERS WITH EXPRESS**

**What is Express?**

Express is a “fast, unopinionated, minimalist web framework for Node.js”. It helps us build web apps!.

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

Its just an NPM Package which comes with a bunch of methods and optional plugins that we can use to build web applications and API’s.

Express helps us:

1. Start up a server to listen for requests
2. Parse incoming requests
3. Match those requests to particular routes
4. Craft our http response and associated content

**Difference between library and framework**

1. Library: When we use a library, we are in charge. We control the flow of the application code and we decide where to use the library
2. Framework: With frameworks, that control is inverted. The framework is in charge and we are just the participants. The framework tells us where to plug in our code. Express is a framework. With framework, we have to follow the rules of it.

**Express app example**

//express app example. go to localhost:3000 for the response

const express = require('express'); //returns a function

const app = express(); //run the function which returns an object

const port = 3000; //define the port

//The below code gets fired when we visit localhost:3000.

// '/' is used for home route

//req is an object for the incoming request which consists of field like headers, baseURL and many other properties

//and response is an object used to send corresponding response.

app.get('/', (req, res) => {

    //NOTE THAT WE CAN ONLY SEND ONE RESPONSE PER REQUEST.

    /\*This means if we have the following code

        res.send(response1);

        res.send(response2); //this response will never be sent

    \*/

    res.send('Hello World!'); //sending a string

    res.send('<p>some html<p>'); //we can send html too

    res.send({some: 'json'}); //we can send js object and it would turn that into json

});

//the below code gets fired when we visit localhost:3000/home

//rerouting

app.get('/home', (req, res) => {

    res.send("I have routed to home page");

});

//to send a post request, we use post method

app.post('/home', (req, res) => {

    res.send("This is a post request!");

});

//to send responses according to the path.

//the below code would fire when we request localhost:3000/home/{anything}

//Notice the colon

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

    //to get the value of anything, use req.params object

    res.send(`This is ${req.params.anything} page`);

    //now when you go to localhost:3000/home/chickens, you would recieve 'This is chickens page'

});

app.get('/home/:anything/:id', (req, res) => {

    const {anything, id} = req.params; //destructuring

    res.send(`Objects id is ${id} from ${anything} page`);

    //now when you go to localhost:3000/home/chickens, you would recieve 'This is chickens page'

});

//WORKING WITH QUERY STRINGS

// example format of query string: baseurl/search?q=dogs&color=red. Here q and color are queries

//if we visit localhost:3000/search?q=dogs&color=red, we would get this response

app.get('/search', (req, res) => {

    const {q, color} = req.query;

    res.send(`<p>Here is your query ${q} with color ${color}`);

});

//to send a generic reponse to every other invalid routes, refer below

//REMEMBER THIS MUST COME AT THE END OF ALL THE GET METHODS OTHERWISE IF IT IS AT

//THE TOP, THEN FOR VALID REQUESTS ALSO WE WILL GET THIS RESPONSE ONLY

app.get('\*', (req, res) => {

    res.send("I dont know this path");

});

//start up the server on the defined port and listen to the incoming requests

app.listen(port, () => {

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

});

**Nodemon:**

Whenever we make changes to our code, we have to restart the server for the changes to take place. We can avoid that by using nodemon [https://www.npmjs.com/package/nodemon](https://www.npmjs.com/package/nodemon%20)

**SECTION 34: CREATING DYNAMIC HTML WITH TEMPLATING**

**Templating:** templating allows us to define a “pattern” for a webpage, that we can dynamically modify. For eg, the search engine in google, it does not know what we are going to search, based on the search query, it creates a webpage based on the pattern on the fly. We can have dynamic values, dynamic elements, conditionals etc with templates.

For templating, there are many tools like **EJS (Embedded Javascript templating), handlebars, jade language, pug, nunjucks etc.** they all output HTML.

We will be using EJS because it is most commonly used and popular.

Install ejs using npm install ejs command.

**Example on how to use ejs:**

Index.js

const express = require('express');

const app = express();

//we are not requiring ejs, instead, by setting the view engine property,

//express behind the scenes will require this library

app.set('view engine', 'ejs');

//Now by default, when we are using view engine, express, by default is

//going to assume that our views will be in a directory named views

//so we have to store our html templates in views folder.

//default path is process.cwd() + '/views'

//we can set other names for the views directory using 'views' property of app.set() method

//app.set('views', 'some other name instead of views');

//Now the extension of the file with ejs has to be .ejs and not .html

const port = 3000;

app.get('/', (req, res) => {

    //to render our ejs file, use app.render() method

    res.render('home');

    //here node is going to look for views folder inside our CURRENT WORKING DIRECTORY

    //so we dont have to specify full path:

    //res.render('views/home') is same as res.render('home');

})

app.listen(port, () => {

    console.log(`LISTENING ON PORT ${port}`);

})

views/home.ejs (it is in the same directory as index.js)

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

</head>

<body>

    <h1>Our first templating</h1>

    <p>Lorem ipsum dolor sit amet consectetur, adipisicing elit. Incidunt debitis vel minus delectus accusantium impedit. Numquam mollitia porro, quod dolore debitis odio harum iusto recusandae voluptate sunt magni nemo sit!</p>

</body>

</html>

**Problem with views path:**

When we run node index.js from the directory where index.js and views folder are present, we get no errors. But what if we move back one directory using cd .. and we run node outer\_directory/index, So node is going to look for views folder inside outer\_directory and not in the directory where index.js is present. And that results in an error. To solve this error, refer below:

const express = require('express');

const path = require('path'); //no need to install path package, it is in-built with node.

const app = express();

app.set('view engine', 'ejs');

app.set('views', path.join(\_\_dirname, 'views')); //what this does is it sets the path of views folder to directory\_where\_index.js\_lives/views. \_\_dirname is the absolute path of the directory where index.js lives

//now when we run index.js from anywhere using node directory1/directory2/.../index, node is going to look for views in the directory\_where\_index.js\_lives/views

const port = 3000;

app.get('/', (req, res) => {

    //to render our ejs file, use app.render() method

    res.render('home');

    //here node is going to look for views folder inside our CURRENT WORKING DIRECTORY

    //so we dont have to specify full path:

    //res.render('views/home') is same as res.render('home');

})

app.listen(port, () => {

    console.log(`LISTENING ON PORT ${port}`);

})

**Passing data for js to ejs file:**

Index.js

const express = require('express');

const path = require('path');

const app = express();

app.set('view engine', 'ejs');

app.set('views', path.join(\_\_dirname, 'views'));

const port = 3000;

app.get('/random', (req, res) => {

    const randomNum = Math.floor(Math.random() \* 10) + 1; //random number between 1 to 10;

    const objToBePassedInEJS = {random: randomNum};

    res.render('home', objToBePassedInEJS); //pass in an object in render function and this object can be used in home.ejs file

})

app.get('/', (req, res) => {

    res.send("You are in home page!");

});

app.listen(port, () => {

    console.log(`LISTENING ON PORT ${port}`);

})

home.ejs

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

</head>

<body>

    <h1>Random Number Generator</h1>

    <p>Your random number is <%= random %></p> <!--Extracting the random field from the passed object-->

    <!--Notice the '<%= %>' sign....For more features refer the documentation of ejs-->

</body>

</html>

**Conditions in ejs Example:**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

</head>

<body>

    <h1>Random Number Generator</h1>

    <p>Your random number is <%= random %></p> <!--Extracting the random field from the passed object-->

    <!--Notice the '<%= %>' sign....For more features refer the documentation of ejs-->

    <% if(random % 2 === 0){ %>

        <h2>This is an even number</h2> <!--if random is even, render this-->

    <% }else{ %>

        <h2>This is an odd number</h2>

    <% } %>

</body>

</html>

**Loops in ejs example:**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

</head>

<body>

    <h1>All the cats</h1>

    <ul>

    <% for(let cat of cats){ %>

        <li><%= cat %></li>

    <% } %>

    </ul>

</body>

</html>

**Serving static assets in express:**

The static files which we want to send in response are known as static assets.

Use the following code to serve images, CSS files, and JavaScript files in a directory named public (directory name can be anything but commonly it is named as public):

app.use(express.static('public'))

Now keep all your static html, css and js files in this public directory.

The best practice is to make a separate folder of the each of these files as shown below:

public/

html/your\_html\_files

css/your\_css\_files

js/your\_js\_files

How to link these static files to the ejs files present in views folder?

All the ejs files have access to the public folder.

So lets say we have a folder structure like this:

public/

css/app.css

views/

home.ejs

To include app.css in home.ejs, we just have to reference the file path as shown below (Remember that the files inside views folder have access to public folder

<link rel="stylesheet" href="/css/app.css">

Here we have the same problem which we had with views path.

When we run node index.js from the directory where index.js and public folder are present, we get no errors. But what if we move back one directory using cd .. and we run node outer\_directory/index, So node is going to look for public folder inside outer\_directory and not in the directory where index.js is present.

**To solve this problem, we just have to modify the path in app.use(express.static()) method as shown below**

app.use(express.static(path.join(\_\_dirname, 'public')));

**EJS and Partials:**

The whole idea of partials is that if there is common stuff in the ejs files (like there is some html content which is common in all the ejs files), then we instead of writing that common content in all ejs files, we can make a separate ejs file which has that common content and include that ejs file in the other ejs files

Lets say we have the below content common in all ejs files:

<head>

    <meta charset="UTF-8">

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

    <title><%= name %></title>

    <link rel="stylesheet" href="/css/app.css">

</head>

So now we make a separate ejs file (Let’s name it as head.ejs) and head.ejs file contains the above content.

It is good practice to define the partials in partial folder in views folder as shown below:

views/

partials/

head.ejs

home.ejs

subreddit.ejs

subreddit.ejs:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title><%= name %></title>

    <link rel="stylesheet" href="/css/app.css">

</head>

<body>

    <h1> <%= name %> </h1>

    <h2><%= subscribers %> Total Subscribers</h2>

    <b><%= description %></b>

    <div>

        <% for(let i = 0; i < posts.length; i++){ %>

            <div>

                <p>Author: <%= posts[i].author %></p>

                <p>Title: <%= posts[i].title %></p>

                <% if(posts[i].img){ %>

                    <img src="<%= posts[i].img %> " alt="">

                <% } %>

            </div>

            <hr>

        <% } %>

    </div>

</body>

</html>

Now by including head.ejs, subreddit.ejs will look like this:

<!DOCTYPE html>

<html lang="en">

<%- includes('partials/head') %>

<body>

    <h1> <%= name %> </h1>

    <h2><%= subscribers %> Total Subscribers</h2>

    <b><%= description %></b>

    <div>

        <% for(let i = 0; i < posts.length; i++){ %>

            <div>

                <p>Author: <%= posts[i].author %></p>

                <p>Title: <%= posts[i].title %></p>

                <% if(posts[i].img){ %>

                    <img src="<%= posts[i].img %> " alt="">

                <% } %>

            </div>

            <hr>

        <% } %>

    </div>

</body>

</html>

**HTTP request methods or verbs**

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs. Each of them implements a different semantic, but some common features are shared by a group of them: e.g. a request method can be safe, idempotent, or cacheable.

**GET**

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

**HEAD**

The HEAD method asks for a response identical to a GET request, but without the response body.

**POST**

The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.

**PUT**

The PUT method replaces all current representations of the target resource with the request payload.

**DELETE**

The DELETE method deletes the specified resource.

**CONNECT**

The CONNECT method establishes a tunnel to the server identified by the target resource.

**OPTIONS**

The OPTIONS method describes the communication options for the target resource.

**TRACE**

The TRACE method performs a message loop-back test along the path to the target resource.

**PATCH**

The PATCH method applies partial modifications to a resource.

**SECTION 35: Defining RESTful Routes**

**Get and Post requests:**

Get:

1. Used to retrieve information.
2. Data is sent via a query string in the url ([www.example.com/search?q=dogs](http://www.example.com/search?q=dogs)).
3. Information is plainly visible in the url (as shown in the above url).
4. Limited amount of data can be sent (url sizes are limited).

Eg:

    <form action = "/movies" method = "GET">

        <input type = "text" name = "genre">

        <button>Submit</button>

    </form>

When the submit button is clicked, we will be redirected to movies?genre={movieName}

Post:

1. Used to post data to server
2. Used to write/create/update
3. **Data is sent via request body** **(IMPORTANT),** not a query string via url.
4. Can send any sort of data (Eg: JSON).
5. Eg: Signing up, adding a comment etc

Eg:

    <form action = "/movies" method = "POST">

        <input type = "text" name = "genre">

        <button>Submit</button>

    </form>

When the submit button is clicked, we will be redirected to movies (NO QUERY STRING IS INVOLVED)

Request is sent via request body.

**Parsing the request body:**

Eg:

index.html

<body>

    <h1>GET AND POST REQUESTS</h1>

    <h2>GET</h2>

    <form action = "http://localhost:3000/movies" method = "GET">

        <input type = "text" name = "genre" placeholder="Enter genre">

        <button>Submit</button>

    </form>

    <h2>POST</h2>

    <form action = "http://localhost:3000/movies" method = "POST">

        <input type = "text" name = "genre" placeholder="Enter genre">

        <button>Submit</button>

    </form>

</body>

Index.js

const express = require('express');

const app = express();

//get request

app.get('/movies', (req, res) => {

    const {genre} = req.query;

    res.send(`Your movie genre was ${genre}`);

});

//post request

app.post('/movies', (req, res) => {

    const {genre} = req.body; //will result in an error because req.body is undefined

    res.send(`Your movie genre was ${genre}`);

});

app.listen(3000, (req, res) => {

    console.log("Listening on port 3000!");

})

Now when we send a post request, req.body is printed undefined . Why? Here is the statement from the docs:

**req.body Contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as express.json() or express.urlencoded().**

This means by default req.body is undefined. The following example shows how to parse req.body()

const express = require('express');

const app = express();

//We have to use these methods for parsing the corresponding data as shown below

app.use(express.json()) // for parsing json data

//for parsing form data

app.use(express.urlencoded({ extended: true })) // for parsing x-www-form-urlencoded.

//get request

app.get('/movies', (req, res) => {

    const {genre} = req.query;

    res.send(`Your movie genre was ${genre}`);

});

//post request

app.post('/movies', (req, res) => {

    const {genre} = req.body;

    res.send(`Your movie genre was ${genre}`);

});

app.listen(3000, (req, res) => {

    console.log("Listening on port 3000!");

})

**REST:**

REST stands for **representational state transfer.**

“REST is an architectural style for distributed hypermedia systems” – Yikes

**Its basically a set of guidelines for how a client + server should communicate and perform CRUD (create, read, update and delete) operations on a given resource.**

**The main idea of REST is to treating data on the server-side as resources that can be CRUDed.**

**The most common way of approaching REST is in formatting the URLs and HTTP verbs (GET, POST etc) in your applications.**

[**https://en.wikipedia.org/wiki/Representational\_state\_transfer**](https://en.wikipedia.org/wiki/Representational_state_transfer)

//example of creating api with REST compliance:

//The below table represents the apis (Which is built in a demo application in REST Folder)

//This is not the one way of implementing RESTful APIs. But this is very //common to match different HTTP verbs with some resource (In this case //“comments”), some base URL and then adding on an id or some unique //identifier

|  |  |  |  |
| --- | --- | --- | --- |
| **NAME** | **PATH** | **VERB** | **PURPOSE** |
| **Index** | **/comments** | **GET** | **Displays all comments** |
| **New** | **/comments/new** | **GET** | **Form to create new comment** |
| **Create** | **/comments** | **POST** | **Create new comment on the server** |
| **Show** | **/comments/:id** | **GET** | **Details for one specific comment** |
| **Edit** | **/comments/:id/edit** | **GET** | **Form to edit specific content** |
| **Update** | **/comments/:id** | **PATCH** | **Updates specific comment on server** |
| **Destroy** | **/comments/:id** | **DELETE** | **Deletes specific item on the server** |

The below code will show an example of each of these requests:

const express = require('express');

const path = require('path');

const methodOverride = require('method-override'); //method-override package lets us use patch, put and other http verbs where client does not support it (In html, forms only support get and post)

const {v4: uuid} = require('uuid'); //uuid() will give us an unique id

const app = express();

app.set('view engine', 'ejs'); //setting the view engine to ejs

app.set('views', path.join(\_\_dirname, 'views'));

app.use(express.urlencoded({ extended: true })); //for parsing form data

app.use(methodOverride("\_method"));

let comments = [

    {

        id: uuid(),

        username: 'Todd',

        comment: 'lol that is so funny'

    },

    {

        id: uuid(),

        username: 'Skyler',

        comment: 'I like to go birdwatching with my dog'

    },

    {

        id: uuid(),

        username: 'Sk8erBoi',

        comment: 'Plz delete your account, Todd'

    },

    {

        id: uuid(),

        username: 'onlysayswoof',

        comment: 'woof woof woof'

    }

];

//homepage

app.get('/', (req, res) => {

    res.render('home');

});

//getting all the comments

app.get('/comments', (req, res) => {

    res.render('comments/comments', {comments}); //index.ejs shows all the comments

});

//render adding a new comment page

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

    res.render('comments/new'); //new.ejs shows the form to submit a new comment

});

//get one specific comment details

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

    const {id} = req.params;

    const comment = comments.find((comment) => comment.id === id);

    res.render('comments/show', {comment});

});

//edit a specific comment

app.get('/comments/:id/edit', (req, res) => {

    const {id} = req.params;

    const comment = comments.find((comment) => comment.id === id);

    res.render('comments/edit', {comment}); //gets a form to edit comment

});

/\*  HTTP verb: PATCH

    PATCH is used to make partial modifications to a resource: (In this case we are only making changes to the comment text

    and not the username). So we should do this in PATCH request.

    HTTP verb: PUT

    PUT is used to modify the whole resource with the information recieved in the payload

    (payload means the data we send to the server with verbs like post, patch, put etc)

    See, we can still use post request for this but it is generally a good idea to follow the guidelines

\*/

//We are using method-override package to send patch request. (Originally we are sending a post request in html forms, but using method-override, we are overriding with a PATCH request)

app.patch('/comments/:id', (req, res) => {

    const {id} = req.params;

    const comment = comments.find((comment) => comment.id === id);

    comment.comment = req.body.newCommentText;

    res.redirect('/comments'); //redirect back to comments page (just like post request)

});

//delete a comment using DELETE http verb

app.delete('/comments/:id', (req, res) => {

    const {id} = req.params;

    comments = comments.filter((comment) => comment.id !== id);

    res.redirect('/comments'); //redirect back to comments page (just like post request);

})

//creating a new comment

/\*in this post method, we send 2 requests:

1) POST REQUEST to /comments

2) REDIRECT (GET REQUEST) to /comments

\*/

app.post('/comments', (req, res) => {

    const {username, comment} = req.body;

    const id = uuid(); //generate a new id

    comments.push({id, username, comment});

    //why are we redirecting to /comments

    //instead of res.render('comments/comments', {comments});

    //because in post request, when you refresh (A form resubmission alert is given), and if we press ok,

    //we again send the post request with the previous data

    //to prevent that, we use the redirect method (which sends a get request to /comments)

    res.redirect('/comments'); //it will include a status code of 302

});

app.listen(3000, (req, res) => {

    console.log("Listening on port 3000!");

});

**SECTION 36: Our First Database: MongoDB**

According to Mongo’s homepage, it is “the most popular database for modern applications”. It is commonly used with Node.

Mongo is a document database, which we can use to store and retrieve complex data from.

WHY USE A DATABASE? (Instead of just saving to a file)

1. Databases can handle large amounts of data efficiently and store it compactly.
2. They provide tools for easy insertion, querying and updating of data
3. They generally offer security features and control over access to data.
4. They (generally) scale well.

**SQL vs NO-SQL Databases**

|  |  |
| --- | --- |
| **SQL Databases (Structured Query language)** | **NO-SQL Databases** |
| SQL databases are relational databases. We predefine a schema of tables before inserting anything. (Like a spreadsheet). We cannot alter the structure for one entry. Every entry has to follow the pattern the table defines. | NoSQL databases do not use SQL. There are many types of no-sql databases, including document, key-value and graph stores. |
| MySQL, Postgres, SQLite, Oracle, Microsoft SQL Server | MongoDB, Couch DB, Neo4j, Cassandra, Redis |

|  |  |  |
| --- | --- | --- |
| Id | Comment | Post\_id |
| 1 | That is so funny | 1 |
| 2 | This is not funny at all | 1 |
| 3 | Ughhhhh | 2 |

**SQL Example vs NO-SQL Example**

|  |  |  |
| --- | --- | --- |
| ID | Author | Post |
| 1 | Colt | Blah blah |
| 2 | Tyra | Papapapa |

Table1: Posts Table

Table2: Comments Table of the corresponding posts (These two tables are related)

Whereas in NO-SQL database, the data may be stored like this

{

    "posts": [

        {

            "id": 1,

            "author": "colt",

            "post": "blah blah",

            "comments": [

                {

                    "id": 1,

                    "comment": "this is so funny"

                },

                {

                    "id": 2,

                    "comment": "this is not funny at all"

                }

            ]

        },

        {

            "id": 2,

            "author": "Tyra",

            "post": "papapapa",

            "comments": [

                {

                    "id": 3,

                    "comment": "ughhhh"

                }

            ]

        }

    ]

}

**Why are we learning mongo?**

1. Mongo is very commonly used with Node and Express (MEAN and MERN stacks)
2. It is easy to get started with (although it can be tricky to truly master)
3. It plays particularly well with javascript
4. It’s popularity also means there is a strong community of developers using mongo.

**Installing Mongo**: <https://zarkom.net/blogs/how-to-install-mongodb-for-development-in-windows-3328>

**Mongo Shell:**

**Text

Description automatically generated**

show databases/dbs is going to list all the databases

db is going to list the current database that we are using

use {databaseName} is going to see if there is a database with the given name, if there is not, then it is going to create a database with the given name.

The reason that the last showdbs is not showing the “animalShelter” because the “animalShelter” database has no data in it. When it will have some data, it will be shown.

**What is BSON?**

JSON has several disadvantages:

1. JSON is a text-based format, and text parsing is very slow
2. JSON’s readable format is far from space-efficient, another database concern
3. JSON only supports a limited number of basic data types.

Due to these disadvantages, MongoDB uses BSON which stands for Binary JSON. BSON stores the data in binary which is space-efficient and also it provides support for more datatypes like date, raw binary etc.

MongoDB stores data in BSON format both internally, and over the network, but that doesn’t mean you can’t think of MongoDB as a JSON database. Anything you can represent in JSON can be natively stored in MongoDB and retrieved just as easily in JSON.

To know more about BSON: <https://www.mongodb.com/json-and-bson>

REFER THIS LINK FOR BELOW 4 TOPICS: <https://www.mongodb.com/docs/manual/crud/>

MongoDB uses collection and documents.

What are collections? **Collections store documents which store data in key value pairs.**

What are documents? **Documents store data in field-value pairs. (JSON format is most popular)**

**Inserting with Mongo:**

Refer this link: <https://www.mongodb.com/docs/manual/tutorial/insert-documents/>

Text

Description automatically generatedText

Description automatically generated

Notice that we are not inserting the data in JSON format, we are inserting the data in javascript object format and that data will be converted into BSON.

In the above database “animalShelter”, we have created 2 collections “dogs” and “cats” and inserted documents (documents are just data in key value pairs).

**Finding with Mongo:**

Refer this: <https://www.mongodb.com/docs/manual/reference/method/db.collection.find/>

db.collection.find() returns every document of the collection.

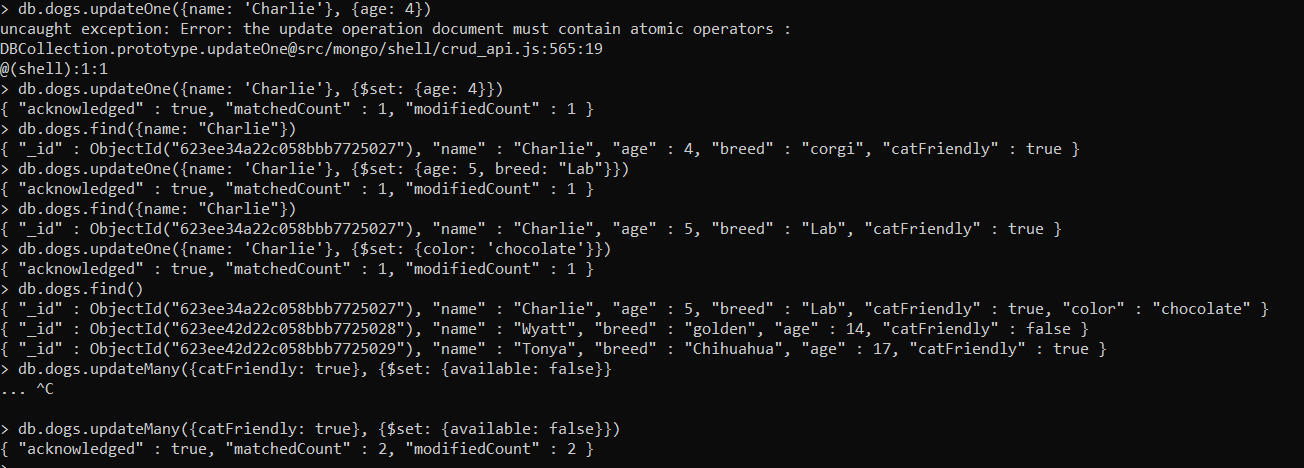
db.collections.find({query}) returns the document which matches the query. Refer the example given below

Text

Description automatically generated

**Updating with mongo:**

**Refer this:** [**https://www.mongodb.com/docs/manual/tutorial/update-documents/**](https://www.mongodb.com/docs/manual/tutorial/update-documents/)

****

Notice the special operator “$” in the update method

**Deleting with mongo:**

**Refer this:** [**https://www.mongodb.com/docs/manual/tutorial/remove-documents/**](https://www.mongodb.com/docs/manual/tutorial/remove-documents/)

**Graphical user interface

Description automatically generated**

**Query Operators:**

**Refer this:** [**https://www.mongodb.com/docs/manual/reference/operator/query/**](https://www.mongodb.com/docs/manual/reference/operator/query/)

For eg to get dogs whose age is >= 10 etc. (Query based operators)

**SECTION 37: MONGOOSE**

To connect our application with the mongoDB database, refer this: <https://www.mongodb.com/docs/drivers/>

**What is mongoose?**

Mongoose is an ODM (Object data mapper or Object document mapper). ODMs like Mongoose map documents coming from a database into usable javascript objects.

The central point of using mongoose is to provide an easier way to interact with a mongo database from our javascript

Mongoose provides ways for us to model out our application data and define a schema. It offers easy ways to validate data and build complex queries from the comfort of JS.

It connects mongo to nodejs

**Homepage:** [**https://mongoosejs.com/**](https://mongoosejs.com/)

**Connecting mongoose to mongodb:**

//Install mongoose package using npm i mongoose

const mongoose = require('mongoose');

//The default port for mongoDB is 27017 as defined in

//https://www.mongodb.com/docs/manual/reference/default-mongodb-port/

//here movieApp is the collection name that we are going to use. If it does not exist, it will create a new collection with name movieApp

//the following line returns a promise

mongoose.connect('mongodb://localhost:27017/movieApp')

.then(() => console.log("Connection open!"))

.catch(err => console.log("Oops", err));

**Our first mongoose model:**

//Install mongoose package using npm i mongoose

const mongoose = require('mongoose');

//The default port for mongoDB is 27017 as defined in

//https://www.mongodb.com/docs/manual/reference/default-mongodb-port/

//here movieApp is the database name that we are going to use. If it does not exist, it will create a new database with name movieApp in mongo

//the following line returns a promise

mongoose.connect('mongodb://localhost:27017/movieApp')

.then(() => console.log("Connection open!"))

.catch(err => console.log("Oops", err));

//For models, refer this: https://mongoosejs.com/docs/models.html

//Models are basically classes from which documents are defined. Models are used for creating and reading documents from MongoDB database.

//schema defines how the data in the document will look like (defines the structure of the data in a document)

//create a schema

const movieSchema = new mongoose.Schema({

    title: 'string',

    year: 'number',

    score: 'number',

    rating: 'string'

});

//create a model

//The first argument is the collection name and the second argument is the schema

const Movie = mongoose.model('Movie', movieSchema);

//create a new document in 'Movie' collection

const amadeus = new Movie({

    title: 'Amadeus',

    year: 1984,

    score: 9.2,

    rating: 'R'

});

//save the document in mongodb database

amadeus.save(); //it returns a promise

**Model.insertMany();**

//what Model.insertMany does is, it takes multiple documents and it saves it in mongo database.

//so we dont have to use save() method

Movie.insertMany([

    {title: 'Amelie', year: 2001, score: 8.3, rating: 'R'},

    {title: 'Alien', year: 1979, score: 8.1, rating: 'R'},

    {title: 'The Iron Giant', year: 1999, score: 7.5, rating: 'PG'},

    {title: 'Stand By Me', year: 1986, score: 8.6, rating: 'R'},

    {title: 'Moonrise Kingdom', year: 2012, score: 7.3, rating: 'PG-13'}

]).then(data => {

    console.log("IT WORKED!");

    console.log(data);

})

**Finding with mongoose:**

//Movie.find does NOT return a promise, it is a promise like object

//So if we use async await, we would have to use exec() method as it returns a full fledged promise as defined here: https://mongoosejs.com/docs/promises.html

Movie.find({rating: 'PG-13'})

.then(data => console.log(data));

//return all the movies whose year >= 2010

Movie.find({year: {$gte: 2010}})

.then(data => console.log(data));

**Updating with mongoose:**

//res does not contain the updated data, it contains the acknowledgement info whether the updation was successful or not

Movie.updateOne({title: 'Amadeus'}, {year: 1984})

.then(res => console.log(res));

Movie.updateMany({title: {$in: ['Amadeus', 'Stand By Me']}}, {score: 10})

.then((data) => console.log(data));

**Delete with Mongoose:**

//deleting

Movie.remove({title: 'Amelie'})

.then((msg) => console.log(msg));

//deleteMany

Movie.deleteMany({year: {$gte: 1999}})

.then((msg) => console.log(msg));

**Mongoose schema validations:**

mongoose.connect('mongodb://localhost:27017/movieApp')

.then(() => {

    console.log("Connection open!");

    /\*We may think that after connecting with mongo,

    all the insert, update, delete, find etc operations should be done IN THIS CALLBACK

    (as of now we are doing these operations outside).

    What happens is mongo uses OPERATION BUFFERING. It ensures that all these operations are done after connecting to mongoDB

    So we can write our code outside this callback

    \*/

})

.catch(err => console.log("Oops", err));

//before we used to create a schema like this:

const productSchema = new mongoose.Schema({

    name: 'string',

    price: 'number',

});

//But we can write the schema as below which allows us to add more properties or validations

const productSchema = new mongoose.Schema({

    name: {

        type: 'string',

        required: true

    },

    price: {

        type: 'number',

        required: true  //required means this field cannot be empty

    }

});

//so if we create a product with only name field, this is going to throw an error

const Product = mongoose.model('Product', productSchema);

const bike = new Product({name: 'Mountain Bike'});

bike.save()

.then(data => console.log(data))

.catch(err => console.log(err));

**Additional Schema Constraints:**

Refer: [**https://mongoosejs.com/docs/schematypes.html**](https://mongoosejs.com/docs/schematypes.html)

Eg:

const productSchema = new mongoose.Schema({

    name: {

        type: 'string',

        required: true,

        lowercase: true, //if name is "BIKE", then it will get saved as "bike"

        trim: true, //"  BiKE  " will be saved as "bike"

        maxLength: 20 //Name should have maxLength of 20

    },

    price: {

        type: 'number',

        required: true,  //required means this field cannot be empty

        min: 20 //minimum price should be 20

    },

    onSale: {

        type: 'boolean',

        default: false //if this parameter is not included in document, it would have a default value of false

    },

    categories: {

        type: [String] //categories is an array of strings

    },

    qty: { //nested

        online: {

            type: 'number',

            default: 0

        },

        inStore: {

            type: 'number'

        }

    }

});

**Validating Mongoose updates:**

//creating a document with the above model

const tirePump = new Product({name: 'Tire Pump', price: 15, categories: ['cycling']});

tirePump.save()  //will fail because price is less than 20

.then(data => console.log(data))

.catch(err => console.log(err));

//creating a document with the above model

const tirePump = new Product({name: 'Tire Pump', price: 25, categories: ['cycling']});

tirePump.save()  //will get saved in database because price > 20

.then(data => console.log(data))

.catch(err => console.log(err));

Now what if we update the tirePump document?

//the below code works even if the price is less than 10.

Product.findOneAndUpdate({name: 'Tire pump'}, {price: 10}, {new: true})

.then((data) => {

    console.log(data);

})

.catch((err) => {

    console.log(err);

});

**So validations only work in default when we create a new document.** To enforce validations in update method also, we have to set the runValidators option to true as shown below

//the below code now works because runValidators is set to true

Product.findOneAndUpdate({name: 'tire pump'}, {price: 10}, {new: true, runValidators: true})

.then((data) => {

    console.log(data);

})

.catch((err) => {

    console.log(err);

});

**Mongoose validation errors:**

See the price and size property from the below schema

const productSchema = new mongoose.Schema({

    name: {

        type: 'string',

        required: true,

        lowercase: true, //if name is "BIKE", then it will get saved as "bike"

        trim: true, //"  BiKE  " will be saved as "bike"

        maxLength: 20 //Name should have maxLength of 20

    },

    price: {

        type: 'number',

        required: true,  //required means this field cannot be empty

        min: [20, 'price must be greater >= 20'] //minimum price should be 20 and if it is not, then the error msg which is defined in the second element will be shown

    },

    onSale: {

        type: 'boolean',

        default: false //if this parameter is not included in document, it would have a default value of false

    },

    categories: {

        type: [String] //categories is an array of strings

    },

    qty: { //nested

        online: {

            type: 'number',

            default: 0

        },

        inStore: {

            type: 'number'

        }

    },

    size: {

        type: 'string',

        enum: ['S', 'M', 'L', 'XL'] //the size should be one of the values from this array

    }

});

**Mongoose Instance Methods:**

productSchema.methods.greet = function(){

    console.log("Hello");

    console.log(`${this.name}`);

}

productSchema.methods.toggleOnSale = function(){

    this.onSale = !this.onSale;

    this.save(); //after toggling, save in db

}

const Product = mongoose.model('Product', productSchema);

const bike = new Product({name: 'Mountain Bike', price: 25});

bike.greet();

bike.toggleOnSale();

**Adding Model Static Methods:**

productSchema.statics.fireSale = function(){

    //this does not refer to the instance or object

    //this refers to the model class

    //The reason we did not use the below line is because we do not want to hardcode the model name

    //Product.updateMany({}, {onSale: true, price: 0});

    this.updateMany({}, {onSale: true, price: 0});

}

const Product = mongoose.model('Product', productSchema);

Product.fireSale();

**Mongoose virtuals:**

Refer this: <https://mongoosejs.com/docs/tutorials/virtuals.html>

**Mongoose middleware:**

Refer this**:** [**https://mongoosejs.com/docs/middleware.html**](https://mongoosejs.com/docs/middleware.html)

Mongoose middleware allows us to run some functions before or after a query is run.

Usecase: Say, we want to delete an user, so before deleting an user, we should delete all the posts and comments of the user that he has created over the years.

Eg:

const personSchema = new mongoose.Schema({

    first: String,

    last: String

});

//the callback will run before the save method.

personSchema.pre('save', async function(){

    console.log("ABOUT TO SAVE!!!")

});

//the callback will run after the save method

personSchema.post('save', async function(){

    console.log("JUST SAVED!!!")

});

const Person = mongoose.model('Person', personSchema);

const k = new Person({first: 'Kristen', last: 'Sun'});

k.save()

**SECTION 38: PUTTING IT ALL TOGETHER: EXPRESS + MONGOOSE**

**Express + Mongoose Basic Setup:**

const express = require('express');

const app = express();

const path = require('path');

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/movieApp')

.then(() => {

    console.log("Mongo connection open!");

})

.catch(err => console.log(err));

app.set('views', path.join(\_\_dirname, 'views'));

app.set('view engine', 'ejs');

app.get('/', (req, res) => res.send("home"));

app.listen(3000, () => {

    console.log("Listening on port 3000");

});

**Refer the project in mongo\_express folder**

**SECTION 40: MIDDLEWARE: THE KEY TO EXPRESS**

**Intro to express middleware:**

**Express middleware** are functions which have access to request, response objects, and the next middleware function (which is denoted by next()) and run during the request/response lifecycle.

1. Middleware are just functions
2. Each middleware has access to the request and response objects.
3. They can end the request-response cycle
4. OR call the next middleware function

Refer: <https://expressjs.com/en/guide/using-middleware.html>

//express.json() returns a function, similarly express.urlencoded() and methodOverride()

//These functions which are passed in app.use() get executed everytime whenever a request comes

app.use(express.json()); //for getting json data in req.body

app.use(express.urlencoded({ extended: true })); //for getting form data in req.body

app.use(methodOverride("\_method")); //for using http verbs other that GET or POST in html forms

//this will execute every time when a request comes to /campgrounds/anything (BE CAREFUL OF MAINTAINING THE CORRECT ORDER OF APP.USE)

//app.use is a middleware because it has access to req, res objects

app.use('/campgrounds', (req, res, next) => {

    console.log("HELLO SERVER");

    ///next is going to execute next matching middleware.

    next();

}, (req, res, next) => {

    console.log("HELLO AGAIN! SERVER");

    next(); //next is going to execute next matching middleware.

});

//show all campgrounds

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

    Campground.find({})

    .then((campgrounds) => {

        res.render('campgrounds/index', {campgrounds});

    })

    .catch((err) => {

        res.send(err);

    })

});

**Using Morgan – Logger Middleware:**

Refer: <https://www.npmjs.com/package/morgan>

Morgan is a HTTP request logger middleware for node.js.

**Defining our own middleware:**

app.use((req, res, next) => {

    console.log("THIS IS MY FIRST MIDDLEWARE!");

    next(); //executes second MATCHING middleware

    console.log("THIS IS MY FIRST MIDDLEWARE!....AGAIN"); //this will be executed after the above next() is completely executed, but writing code after next() is not recommended. So what people do is they return next()…see the next code snippet

})

app.use((req, res, next) => {

    console.log("THIS IS MY SECOND MIDDLEWARE");

    next(); //executes third MATCHING middleware

}, (req, res, next) => {

    console.log("THIS IS MY THIRD MIDDLEWARE");

    next();

});

app.use((req, res, next) => {

    console.log("THIS IS MY FIRST MIDDLEWARE!");

    return next(); //executes second MATCHING middleware, we are returning next so to ensure that no code in this middleware is run after calling next

})

app.use((req, res, next) => {

    console.log("THIS IS MY SECOND MIDDLEWARE");

    return next(); //executes third MATCHING middleware

}, (req, res, next) => {

    console.log("THIS IS MY THIRD MIDDLEWARE");

    return next();

});

**Setting up a 404 route:**

//add this code at the end. if nothing matches, then sends status code as 404 and 'NOT FOUND' message

app.use((req, res) => {

    res.status(404).send('NOT FOUND!');

})

**Protecting Specific routes:**

Sometimes we don’t want to go to a specific path without fulfilling some conditions first like the user should do the authentication first and then only he will have access to this path. For handling such situations, refer below

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

    const {password} = req.query;

    if(password === 'chickennuggets'){

        next();

    }

    else res.send('SORRY!');

}

//whenever we get to /secret route, verifyPassword middleware is going to be executed first

//and then the next callback if next() is called in verifyPassword middleware

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

    res.send("I wont tell my secret :)")

});

**SECTION 42: HANDLING ERRORS IN EXPRESS**

**Express built in error handler**

Refer: <https://expressjs.com/en/guide/error-handling.html>

In express, when we throw an error, express is going to catch that with its own error handler, Its builds a html page based on that error and shows it to us in the browser.

app.get('/', (req, res) => {

    throw new Error('BROKEN') // Express will catch this on its own.

  })

**NOTE: When we pass anything to next() except ‘route’, IT IS GOING TO INVOKE ERROR HANDLER**

//for errors occuring in asynchronous functions, we HAVE to MANUALLY pass the error in next() function

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

    try{

        throw new Error('BROKEN')

    }

    catch(err){

        next(err);

    }

  });

For every async function, use try catch block and inside catch, pass the error to next. And the errors generated by non async functions can be handled by the default express error handler.

When we get an “Unhandled promise rejection” error, it means the error is thrown by a asynchronous function and the error should be passed to next() function

**Defining Custom Error Handlers**

Refer: [https:/expressjs.com/en/guide/error-handling.html](https://expressjs.com/en/guide/error-handling.html)

**Express comes with a built-in error handler that takes care of any errors that might be encountered in the app. This default error-handling middleware function is added at the end of the middleware function stack.**

**If we pass an error to next() and we do not handle it in a custom error handler, it will be handled by the built-in error handler; the error will be written to the client with the stack trace. The stack trace is not included in the production environment.**

We define error-handling middleware functions in the same way as other middleware functions, except error-handling functions have four arguments instead of three: (err, req, res, next). For example:

app.use((err, req, res, next) => {

    console.error(err.stack)

    res.status(500).send('Something broke!')

  })

We define error-handling middleware **last**, after other app.use() and routes calls; for example:

const bodyParser = require('body-parser')

const methodOverride = require('method-override')

app.use(bodyParser.urlencoded({

  extended: true

}))

app.use(bodyParser.json())

app.use(methodOverride())

app.use((err, req, res, next) => {

  // logic

})

Eg:

//home page

app.get('/', (req, res, next) => {

    console.log("first");

    try{

        throw new Error("Hi");

    }

    catch(err){

        next(err); // Here, when we pass err to next, the next app.get() WONT GET TRIGGERED, INSTEAD, we DIRECTLY PASS the error to app.use((err, req, res, next) => {}) method

    }

});

//home page

app.get('/', (req, res) => {

    console.log("second")

    res.send("YELP CAMP!");

});

//The function inside app.use is called as error handling middleware

app.use((err, req, res, next) => {

    console.log(err);

    res.send(err.stack);

})

So if any piece of code throws an error, the error will be handled in the custom error handler middleware we defined below

If we want to trigger the built in error handler from the custom error handler, refer below

//home page

app.get('/', (req, res, next) => {

    console.log("first");

    try{

        throw new Error("Hi");

    }

    catch(err){

        next(err); // Here, when we pass err to next, the next app.get() WONT GET TRIGGERED, INSTEAD, we DIRECTLY PASS the error to app.use((err, req, res, next) => {}) method

    }

});

//home page

app.get('/', (req, res) => {

    console.log("second")

    res.send("YELP CAMP!");

});

//The function inside app.use is called as error handling middleware

app.use((err, req, res, next) => {

    console.log(err);

    next(err); //this next will look for the error handling middlewares defined below, if express does not find any error handling middleware, it will just trigger the default error handler of express

})

**Our Custom Error Class**

Refer: [https:/expressjs.com/en/guide/error-handling.html](https://expressjs.com/en/guide/error-handling.html)

In the docs it is written that

When an error is written, the following information is added to the response:

* The res.statusCode is set from err.status (or err.statusCode). If this value is outside the 4xx or 5xx range, it will be set to 500.
* The res.statusMessage is set according to the status code.
* The body will be the HTML of the status code message when in production environment, otherwise will be err.stack.
* Any headers specified in an err.headers object.

According to the first point, whenever an error occurs, the res.statusCode is set to err.status. That means, if we write our own error handling middleware, express will try to access err.status (The Error class DOES NOT have a status property)

So we can create our own custom error class as defined below

class AppError extends Error{

    constructor(message, status){

        super(); //calling the constructor of parent class

        this.message = message;

        this.status = status;

    }

}

throw new AppError("Error", 401); //in any of the get requests, we can throw an error like this

To handle the error, we can write our middleware like this:

app.use((err, req, res, next) => {

    const {message = 'AN Error has occured', status = '404'} = err;

    res.status(status);

    res.send(message);

});

**SECTION 43: YELP-CAMP: ERRORS AND VALIDATING DATA**

**Client-Side form validations:**

Refer: <https://getbootstrap.com/docs/5.0/forms/validation/>

**Basic Error Handler:**

When we get an “Unhandled promise rejection” error, it means the error is thrown by a asynchronous function and the error should be passed to next() function

**Server-side validations library:**

Refer: <https://www.npmjs.com/package/joi>

WE HAVE TO DO BOTH CLIENT SIDE VALIDATIONS AND SERVER SIDE VALIDATIONS.

CLIENT-SIDE VALIDATIONS:

E.g.: In forms, required, min, max parameters are used to validations

SERVER-SIDE VALIDATIONS:  
E.g.: We can do server-side validations using mongoose validations or using external libraries like joi

Why do we need to do both validations?

Because client-side validations are a joke, anybody can open the dev tools and modify, what if somebody sends a post, patch request to our site (For e.g. from postman). This is the reason we have to do CLIENT and SERVER-SIDE validations.

**SECTION 44: DATA RELATIONSHIPS WITH MONGO**

**One to Few Relationship using mongo:**

//one to few

//Here, user has few Addesses

const userSchema = new mongoose.Schema({

    first: String,

    last: String,

    addresses: [   //addresses is an array of these objects

//Here we are embedding a DOCUMENT inside a DOCUMENT

        {

//Now, when we create a document with this schema,

          //Every element of the addresses array WILL ALSO HAVE AN ID.

          //So, if we want to remove id, refer below

            \_id: {id: false},

            street: String,

            city: String,

            state: String,

            country: String

        }

    ]

});

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

const makeUser = async() => {

    const u = new User({

        first: 'Harry',

        last: 'Potter'

    });

    u.addresses.push({

        street: '123 Sesame St.',

        city: 'New York',

        state: 'NY',

        country: 'USA'

    });

    u.save().then((data) => console.log(data));

}

**One to many relationship with mongo:**

**REFER:** [**https://mongoosejs.com/docs/populate.html**](https://mongoosejs.com/docs/populate.html)

The one-to-few example above where we embed a document within a document works best when we have a small set of information.

One option is to store the data separately, but then store references to documents ID’s somewhere inside the parent as shown below

Basically referencing child document(s) in parent document

{

    farmName: 'Full Belly Farms',

    location: 'Guinda, USA',

    produce: [

        ObjectID('2863245354363'),

        ObjectID('2454354654674'),

        ObjectID('7855342354647')

    ]

}

const productSchema = new mongoose.Schema({

    name: String,

    price: Number,

    season: {

        type: String,

        enum: ['Spring', 'Summer', 'Fall', 'Winter']

    }

});

const farmSchema = new mongoose.Schema({

    name: String,

    city: String,

    products: [{type: mongoose.Schema.Types.ObjectId, ref: 'Product'}] /\*Product is the model name\*/

});

//The ref option is what tells Mongoose which model to use during population,

//in our case the Product model. All \_ids we store here must be document \_ids from the Product model.

//One thing to note here is we have farmSchema in which the products have a reference to 'Product' Model

//And the 'Product' model is defined after farmSchema is defined. But we will not run into any error

const Product = mongoose.model('Product', productSchema);

const Farm = mongoose.model('Farm', farmSchema);

//here each product will have an \_id defined

Product.insertMany([

    {name: 'Goddess Melon', price: 4.99, season: 'Summer'},

    {name: 'Sugar baby watermelon', price: 4.99, season: 'Summer'},

    {name: 'Asparagus', price: 3.99, season: 'Spring'}

]);

const makeFarm = async() => {

    const farm = new Farm({name: 'Full Belly Farms', city: 'Guinda, CA'});

    const melon = await Product.findOne({name: 'Goddess Melon'});

    farm.products.push(melon); //here we are pushing the melon, but in reality, the melon id is being saved.

    farm.save().then(() => console.log(farm));

    //if we see in node js terminal, we will see the full melon object being logged.

    //Actually only objectID of the melon object is stored (We can verify from the mongo shell)

}

makeFarm();

**Mongoose Populate:**

//now in Farm, the products array has object Ids, how to access the data of each product in farm document

//refer the below code

Farm.findOne({name: 'Full Belly Farms'})

    .populate('products')

    .then(farm => console.log(farm));

//here in populate, we pass the products field which is an array of ids

//Now when we log farm, we see the every product data in Farm and we can access the product data also

**One to bazillion mongo relationship:**

In this relationship, the child document will contain a reference to parent document unlink the one to many relationship where the parent document contains the reference to child documents.

This is used when there are many many documents related to a parent document.

Have to read more about the relationships.

**We can also store the child document’s id on parent document and parent document’s id on the child document (Both ways). But is it worth doing? We have to decide based on our use case.**

**REFER THIS LINKS FOR MONGODB SCHEMA DESIGN:**

[**https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1**](https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1)

[**https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-2**](https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-2)

[**https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-3**](https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-3)

**SECTION 45: MONGO RELATIONSHIPS WITH EXPRESS**

**Deletion Mongoose middleware:**

//mongoose middleware

//suppose we have a document that has a reference of many other documents

//for eg an user document has a reference to the user's posts documents

const postSchema = new mongoose.Schema({

    title: String,

    desc: String

});

const userSchema = new mongoose.Schema({

    first: String,

    last: String,

    posts: [{

        type: mongoose.Schema.Types.ObjectId,

        ref: 'Post'

    }]

});

const Post = mongoose.model('Post', postSchema);

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

//Now if we want to delete an user, the posts for that user also should be deleted

//To do that, we can use mongoose middleware as shown below

User.findByIdAndDelete(id);

//the below middleware will be triggered after the User.findByIdAndDelete(id) is executed. Why findOneAndDelete and not findByIdAndDelete? Because findByIdAndDelete method triggers findOneAndDelete. Read the documentation of mongoosh for more details

userSchema.post('findOneAndDelete', (deletedUser) => {

    Post.deleteMany({\_id: { $in: deletedUser.posts}});

//delete all posts whose id is in deleteUser.posts

})

**SECTION 47: EXPRESS ROUTER AND COOKIES (The examples are done in routerDemo folder)**

**Express router intro:**

Till now, we have written all our routes (Get, post, patch, delete etc) in a single file. When we have a larger app, it would be sensible to group our routes into separate files. The **Express Router** helps us with that.

Example of using routers:

Creating a file shelter.js

const express = require('express');

const router = express.Router();

router.get('/', (req, res) => {

    res.send("ALL SHELTERS");

});

router.get('/:id', (req, res) => {

    res.send("VIEWING ONE SHELTER");

});

router.get('/:id/edit', (req, res) => {

    res.send("EDITING THE SHELTER");

});

router.post('/', (req, res) => {

    res.send("CREATING ONE SHELTERS");

});

//export the router

module.exports = router;

Creating index.js

//require express

const express = require('express');

//get the shelter router

const shelterRouter = require('./routes/shelters');

//run express function

const app = express();

/\*

use the shelter routes (We pass the prefix /shelter)

in shelter router the below path is defined

router.get('/:id', (req, res) => {

    res.send("VIEWING ONE SHELTER");

});

and in the app.use() method, we passed a prefix /shelters

This means that when we go to http://localhost:3000/shelters/325fwe (any id),

the above get method will get fired

\*/

app.use('/shelters', shelterRouter);

app.get('/', (req, res) => {

    res.send("ROUTER DEMO");

});

//define port on which the server will start

const port = 3000;

//start up the server

app.listen(port, () => {

    console.log("SERVER RUNNING ON PORT:", port);

})

**Express router and middleware**

const express = require('express');

const router = express.Router();

//lets say, we want to run this middleware every time we get a request to the shelter route

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

    const {isAdmin} = req.query;

    if(!isAdmin)return res.send('SORRY! NOT AN ADMIN');

    return next(); //remember returning next() is a good practice because generally we dont want any code to execute after next()

}

/\*WE HAVE 2 choices:

first choice: Include this middleware in every route

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

    res.send("ALL SHELTERS");

});

router.get('/:id', verifyAdmin, (req, res) => {

    res.send("VIEWING ONE SHELTER");

});

router.get('/:id/edit', verifyAdmin, (req, res) => {

    res.send("EDITING THE SHELTER");

});

router.post('/', verifyAdmin, (req, res) => {

    res.send("CREATING ONE SHELTERS");

});

ELSE, write a app.use() method where we pass in the middleware and it will execute every time a request comes to the shelter route

\*/

router.use(verifyAdmin);

router.get('/', (req, res) => {

    res.send("ALL SHELTERS");

});

router.get('/:id', (req, res) => {

    res.send("VIEWING ONE SHELTER");

});

router.get('/:id/edit', (req, res) => {

    res.send("EDITING THE SHELTER");

});

router.post('/', (req, res) => {

    res.send("CREATING ONE SHELTERS");

});

//export the router

module.exports = router;

**Introducing Cookies:**

**What are they?**

1. Cookies are little bits of information that are stored in a user’s browser when browsing a particular website.
2. Once a cookie is set, an user’s browser will send the cookie on every subsequent request to the site.
3. Cookies allow use to make HTTP stateful (HTTP is stateless, it does not depend on the previous http requests, but with cookies, we can store some data and send to the server along with the http request).

We can use cookies to remember some information about some user and show relevant content to the user. We can store preferences of the user, do session management etc.

We can also use cookies to track users’ web browsing habits (Almost all websites use cookies to track our data).

To know about the cookies for any website, go to dev tools 🡪 Application 🡪Storage 🡪 Cookies

**Sending Cookies:**

//require express

const express = require('express');

//run express function

const app = express();

//once we go to this route, open the applications tab in dev tools and we will be able to see the cookies that we sent

app.get('/setname', (req, res) => {

    //send a cookie, first parameter is the key and second parameter is the value

    res.cookie('name', 'henrietta');

    res.cookie('animal', 'harlequin shrimp');

    res.send('SENT YOU A COOKIE!');

})

**Cookie Parsing Middleware:**

//require express

const express = require('express');

//run express function

const app = express();

//to get cookies from the browser, we have to install cookie-parser

const cookieParser = require('cookie-parser');

app.use(cookieParser()); //cookieParser() returns a function

app.get('/greet', (req, res) => {

    const {name = 'No-name'} = req.cookies;

    res.send('Hey there', name);

})

//once we go to this route, open the applications tab in dev tools and we will be able to see the cookies that we sent

app.get('/setname', (req, res) => {

    //send a cookie, first parameter is the key and second parameter is the value

    res.cookie('name', 'henrietta');

    res.cookie('animal', 'harlequin shrimp');

    res.send('SENT YOU A COOKIE!');

})

**Signing Cookies:**

If we go in the cookie parser documentation (<https://www.npmjs.com/package/cookie-parser>), It is written that “Optionally you may enable signed cookie support by passing a secret string, which assigns req.secret so it may be used by other middleware.”

Signed cookies is about making sure that the original data that we sent to the client, to the browser is still the data that is being sent back to us. ITS NOT ABOUT HIDING IT, KEEPING IT INVISIBLE OR SECRET.

//signed cookie example

//this string is going to be used by cookie parser for signed cookies, this string will be hidden in production code.

app.use(cookieParser('this is my secret'));

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

    res.cookie('fruit', 'apple', {signed: true});

    res.send("OK, SIGNED YOUR COOKIE");

});

//verify the cookie (to make sure that nobody tampered with our cookie, We can also modify the cookies in dev tools (in application tab))

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

    console.log(req.cookies); //we wont find signed cookies here

    console.log(req.signedCookies); //for signed cookies

    //if they are tampered, it will either be empty or the value of fruit will be set to false

})

**Section 48: Express Sessions and flash (The examples are done in express-sessions folder)**

**Introduction to Sessions:**

It is not very practical (or secure) to store a lot of data in client-side using cookies. This is where sessions come in.

Sessions are a server-side data store that we use to make HTTP stateful (HTTP is a stateless protocol). Instead of storing data using cookies, we store the data on the server side and then send the browser a cookie that can be used to retrieve the data (Here this cookie is used as the key).

What is the problem with cookies?

1. Cookies have a maximum size. A cookie can contain data up to 4096 bytes. A domain can have at most 50 cookies i.e., the size of **all** the cookie should not exceed 4096 bytes. That means, we can have 1 cookie of 4096 bytes or 2 cookies of 2048 bytes.
2. Cookies are not secure.

Note: **Session does not replace a database. Sessions data is stored on datastore and once the browser is closed (NOT THE TAB), session expires (If the max age of that session is not specified). This datastore can be implemented in Redis (Redis is a database which is used for short term data storage). If the max age of the session is specified, then the session will expire after that max age time irrespective of the browser is closed or not.**

Graphical user interface, application

Description automatically generated

const express = require('express');

const app = express();

//install express-session

const session = require('express-session');

//for each request, this will send the browser a cookie (The same cookie. And if the cookie is tampered, session expires)

//This cookie is used a key to retrieve data from data store in server side. We have to pass a secret to send a signed cookie

app.use(session({secret: 'This is not a good secret.'}));

app.get('/viewcount', (req, res) => {

    //right now, the data is stored in memory which is not advised for production level code

    //for production app, we can use redis or mongo datastore etc

    if(req.session.count)req.session.count++;

    else req.session.count = 1;

    res.send(`YOU HAVE VIEWED THIS PAGE ${req.session.count} times`);

});

app.listen(3000, () => {

    console.log("LISTENING ON PORT: 3000");

})

**Intro to flash:**

Refer: <https://www.npmjs.com/package/flash>

**SECTION 49: YELPCAMP: RESTRUCTURING AND FLASH**

**Breaking out Campground routes:**

We use router to divide the routes in separate files to increase readability. Convention is to create a routes folder and keep all our routes there (See the Yelp Camp Project for more details).

**Serving Static Assets:**

We make a public directory which would contain all our static html, css and js files (See Yelp Camp Project for more details)

**SECTION 50: AUTHENTICATION FROM SCRATCH**

**Authentication vs Authorization:**

1. Authentication (logging or signing in) is the process of verifying who a particular user is. We typically authenticate with a username/password combo, but we can also use security questions, facial recognition etc.
2. Authorization is verifying what a specific user has access to (Whether they are allowed to edit, remove, delete, what parts of the website they can go etc). Generally, we authorize after a user has been authenticated. “Now that we know who you are, here is what you are allowed to do and NOT allowed to do”

**How to (not) store passwords:**

1. Never store passwords in text as is in your database.
2. Rather than storing a password in the database, we run the password through a hashing function first and then store the result in the database.
3. Hashing functions are functions that map input data of some arbitrary size to fixed-size output values.

**Cryptographic Hash Functions:**

1. One-way function which is infeasible to invert.
2. Small change in input yields large change in output.
3. Deterministic algorithm: same input yields same output
4. Unlikely to find 2 outputs with same value
5. Password Hash Functions are deliberately slow. (**DELIBERATELY. Because if hash functions are fast, it would be easier to just brute force millions of passwords)**

Cookies are signed with sha256 hash function which is a fast hash function. It is not suitable to use the sha256 hash function for passwords because it is fast.

**Password Salts:**

1. Salting is basically an extra step that we take when we are hashing a password to make it harder to reverse engineer or harder to guess password. It is just an extra safe guard.
2. The problem: Lets say we are using a hashing function and if some one gets into our database, he could just use the hash function to hash the most commonly used passwords, get the hash values, and compare the hash values with the passwords in the database (This is a huge problem). To solve this issue, we can use **salts.**
3. A salt is a random value added to the password before we hash it.
4. It helps to ensure unique hashes and mitigate common attacks.

Eg: Let’s say the password is password, after salting, the password becomes password5334Hws, Then we hash password5334Hws (just prepending or appending is a bad idea, because we can also store the salts with passwords and the attacker can guess and store the hashed value in the database. We would also have to store the salt values of each user. This way, someone cannot reverse engineer the passwords if they know the hashing algorithm and also if many users use the same password, the salt values of those users are going to be different

FollowUp: Refer this: <https://stackoverflow.com/questions/1219899/where-do-you-store-your-salt-strings>

**Why are Salted Passwords Better?**

1. **Attackers will have to create a rainbow table for each salt (Rainbow table is basically a precomputed lookup table for millions and millions of passwords with hash value and the corresponding password). If there was no salt, then it would be easy to crack the password.**
2. **Two same passwords will have different salt and thus, different passwords.**

**Intro to Bcrypt**

1. BCrypt is a hashing algorithm that is recommended to hash passwords.
2. Install bcrypt using npm i bcrypt .
3. Refer: <https://www.npmjs.com/package/bcrypt> for documentation.

Note that the decryption is time-taking (With the help of hash or rainbow tables) and also protects if two users have same passwords (Since their hash values would be different)

const bcrypt = require('bcrypt');

//The time taken to create a salt does not depend on the salt rounds.

//Meaning, the time taken to create a salt with 1 round is same as the time taken to create a salt with 100 rounds

//More the salt rounds, more is the cost of processing the data.

//Ideally recommended value of saltRounds is 12

const saltRounds = 12;

bcrypt.genSalt(saltRounds, (err, salt) => {

    //console.log(salt); //each time a different salt is generated

});

//generate a hash of the password 'monkey'

//NOTE that the time taken in generating the password grows exponentially with the value of saltRounds

bcrypt.genSalt(saltRounds, (err, salt) => {

    console.log(salt);

    bcrypt.hash('monkey', salt, (err, hash) => {

        console.log(hash);

    })

});

//compare passwords with the hash

bcrypt.compare('monkey', '$2b$12$a7n.FsJsYtbqjOnT8OieDe2788N6bGduaU9UNA0LB.XnTKGCiEOQm', (err, res) => {

    console.log(res);

})

/\* How bcrypt works?

 \* Bcrypt generates a unique salt, combines it with the password

 \* and generates a hash from (salt + password)

 \* Format of the hash: $[algorithm]$[cost]$[salt][hash]

 \* Note that the hash is formed for (salt + password)

 \*

 \* The compare method works by taking the password, and the hash(from this hash, the salt is extracted),

 \* Then hash is generated for (salt + password), and then the generated hash value is compared with the passed hash value

 \* /

**Auth Demo (See Auth Demo Project):**

Here, we are using sessions and cookies to remember the logged in user (Remember that HTTP is stateless and we use cookies or sessions to make it stateful).

In this example, there is a secret route, which an user can go only if the user is logged in. To remember if the user is logged in, we are using sessions, which send the browser a cookie, and when the user is logged in, we had the userID to the cookie.

So whenever the user goes to the secret route

1. The cookie is being sent by the client to the backend, then the cookie is being validated to see if it has been tampered or not (We used signed cookies).
2. If the cookie is tampered, then the user is being redirected to the login page (And if the cookie is tampered, the session sends a new cookie)
3. If the cookie is not tampered, then the userID which was attached with the cookie is used and the secret page is shown

const express = require('express');

const app = express();

const mongoose = require('mongoose');

const bcrypt = require('bcrypt');

const path = require('path');

const User = require('./models/user');

const session = require('express-session');

app.set('view engine', 'ejs');

app.set('views', path.join(\_\_dirname, 'views'));

//for getting form data

app.use(express.urlencoded({ extended: true }));

//for sending a cookie (Session ID)

app.use(session({

        secret: 'THIS IS NOT A GOOD SECRET',

        resave: true,

        saveUninitialized: true

    }));

//register form get route

app.get('/register', (req, res) => {

    res.render('register');

});

//register form post route

app.post('/register', async(req, res) => {

    const {username, password} = req.body;

    try{

        const saltRounds = 12;

        const hashedPassword = await bcrypt.hash(password, saltRounds);

        // Store hash in your password DB.

        const user = new User({

            username: username,

            password: hashedPassword

        });

        await user.save();

        return res.redirect('/login');

    }

    catch(err){

        return res.send(err);

    }

});

//login form get route

app.get('/login', (req, res) => {

    res.render('login');

});

//login form post route

app.post('/login', async(req, res, next) => {

    const {username, password} = req.body;

    try{

        //find the user (Username is unique)

        const user = await User.findOne({username});

        //if user exists

        if(user){

            //get the hashed password from the database

            const hashedPassword = user.password;

            //check if the password matches with the hashed password

            const validPassword = await bcrypt.compare(password, hashedPassword)

            //if it is a valid password

            if(validPassword){

                //store the userId in the session (This id will be used to display pages according to the user who is currently logged in)

                req.session.userID = user.\_id;

                return res.redirect('/secret');

            }

        }

        return res.send("INVALID CREDENTIALS!");

    }

    catch(err){

        return res.send(err);

    }

});

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

    //if the user is not logged in, redirect to the login page

    if(!req.session.userID){

        return res.redirect('/login');

    }

    //else show the secret page

    return res.render('secret');

});

app.post('/logout', (req, res) => {

    //destroy the session

    req.session.destroy();

    return res.redirect('/login');

})

//connect to mongo using mongoose, database name is auth-demo

const mongoPort = 27017;

mongoose.connect(`mongodb://localhost:${mongoPort}/auth-demo`)

    .then(() => console.log("Mongo connection open at port:", mongoPort))

    .catch(err => console.log(err));

const port = 3000;

app.listen(port, () => {

    console.log("Server up and running at port:", port);

})

User.js model

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

    username: {

        type: String,

        required: true,

        minlength: 3,

        maxlength: 10

    },

    password: {

        type: String,

        required: true

    }

});

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

module.exports = User;

**Auth Demo: Refactoring to model methods**

Modified User.js model

const mongoose = require('mongoose');

const bcrypt = require('bcrypt');

const userSchema = new mongoose.Schema({

    username: {

        type: String,

        required: true,

        minlength: 3,

        maxlength: 10

    },

    password: {

        type: String,

        required: true

    }

});

//adding a static method to validate the credentials

userSchema.statics.findByUserNameAndValidatePassword = async(username, password) => {

    const foundUser = await User.findOne({username});

    if(!foundUser)return false;

    const validPassword = await bcrypt.compare(password, foundUser.password);

    return !validPassword? false: foundUser;

}

//pre method: The callback function will be called before user.save()

userSchema.pre('save', async(next) => {

    const saltRounds = 12;

    this.password =  await bcrypt.hash(password, saltRounds);

    next();

})

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

module.exports = User;

Modified index.js

const express = require('express');

const app = express();

const mongoose = require('mongoose');

const bcrypt = require('bcrypt');

const path = require('path');

const User = require('./models/user');

const session = require('express-session');

app.set('view engine', 'ejs');

app.set('views', path.join(\_\_dirname, 'views'));

//for getting form data

app.use(express.urlencoded({ extended: true }));

//for sending a cookie (Session ID)

app.use(session({

    secret: 'THIS IS NOT A GOOD SECRET',

    resave: true,

    saveUninitialized: true

}));

//register form get route

app.get('/register', (req, res) => {

    res.render('register');

});

//register form post route

app.post('/register', async (req, res) => {

    const { username, password } = req.body;

    try {

        // Store hash in your password DB.

        const user = new User({ username, password });

        await user.save();

        return res.redirect('/login');

    }

    catch (err) {

        return res.send(err);

    }

});

//login form get route

app.get('/login', (req, res) => {

    res.render('login');

});

//login form post route

app.post('/login', async (req, res, next) => {

    const { username, password } = req.body;

    try {

        const foundUser = await User.findByUserNameAndValidatePassword(username, password);

        if (foundUser) {

            //store the userId in the session (This id will be used to display pages according to the user who is currently logged in)

            req.session.userID = foundUser.\_id;

            return res.redirect('/secret');

        }

        return res.send("INVALID CREDENTIALS!");

    }

    catch (err) {

        console.log(err);

        return res.send(err);

    }

});

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

    //if the user is not logged in, redirect to the login page

    if (!req.session.userID) {

        return res.redirect('/login');

    }

    //else show the secret page

    return res.render('secret');

});

app.post('/logout', (req, res) => {

    //destroy the session

    req.session.destroy();

    return res.redirect('/login');

})

//connect to mongo using mongoose, database name is auth-demo

const mongoPort = 27017;

mongoose.connect(`mongodb://localhost:${mongoPort}/auth-demo`)

    .then(() => console.log("Mongo connection open at port:", mongoPort))

    .catch(err => console.log(err));

const port = 3000;

app.listen(port, () => {

    console.log("Server up and running at port:", port);

})

**SECTION 51: YELPCAMP: ADDING IN AUTHENTICATION**

**YelpCamp: Adding in Authentication**

Installing Passport (Library which is used for authentication with nodejs apps)

Refer this: <https://www.npmjs.com/package/passport-local-mongoose>

**Creating our user model**

const mongoose = require('mongoose');

const passportLocalMongoose = require('passport-local-mongoose');

//define the user schema

const userSchema = new mongoose.Schema({

    email: {

        type: String,

        required: true,

        unique: true

    }

});

/\* plug in the passport with the schema

 \* According to the docs, it will add in a username, salt and the hashed password.

 \* Additionally, it also adds some methods to the schema (See documentation

 \* for more details)

 \*/

userSchema.plugin(passportLocalMongoose);

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

module.exports = User;

**Configuring Passport**

To use passport, add these lines (These are taken from the documentation)

/\* TAKEN FROM THE PASSPORT DOCUMENTATION \*/

app.use(passport.initialize());

app.use(passport.session());

// use static authenticate method of model in LocalStrategy

passport.use(new LocalStrategy(User.authenticate()));

// use static serialize and deserialize of model for passport session support

passport.serializeUser(User.serializeUser());

passport.deserializeUser(User.deserializeUser());

**CurrentUser Helper**

Passport provides methods to know whether an user is logged in or not, details of logged in user etc as shown below

module.exports.isLoggedIn = (req, res, next) => {

    //check if an user is logged in

    if(!req.isAuthenticated()){

        return res.redirect('/login');

    }

    //get the current user details

    console.log(req.user);

    return next();

}

Put this middleware in index.js

//middleware to remember the user

app.use((req, res, next) => {

    //every response will have the current user details attached to it

    res.locals.currentUser = req.user;

})

What are locals? And why are we storing the user data like this?

See this: <https://stackoverflow.com/questions/24072333/difference-between-assigning-to-res-and-res-locals-in-node-js-express>

When we use res.locals, whatever data we pass in res.locals, the data becomes visible to all the view (ejs) templates. We don’t have to explicitly pass in the user data to each template.

**SECTION 52: YELP CAMP: BASIC AUTHORIZATION**

**Adding an author to the campground:**

Delete user => Delete Campgrounds and reviews of the user => Campground and reviews model both should have a user id in them

**SECTION 53: Controllers and star ratings:**

**Refactoring to campgrounds controller:**

The term controller comes from MVC (Model View Controller framework). We have our models (Campground, Reviews) organised; views (layouts in ejs) organised. But we did not work with controller. (MVC is an approach to structuring applications)

The concept is that all the data heavy stuff, modelling of data goes in model directory.

All the views, layouts, everything a user sees in views directory.

In controller, we put all the logic which executes during rendering of views or working with models.

We make a controller directory and in the directory, we make two files: campground.js and reviews.js

Controller

Campground.js

Reviews.js

After restructuring, controller/campground.js looks like this

const Campground = require('../models/campground');

//require the express error

const ExpressError = require('../utils/ExpressError');

module.exports.index = (req, res, next) => {

    Campground.find({})

        .then((campgrounds) => {

            return res.render('campgrounds/index', { campgrounds });

        })

        .catch((err) => {

            return next(new ExpressError(err.message, 400));

        })

}

module.exports.editCampground = async (req, res, next) => {

    const { id } = req.params;

    const { title, location, image, price, description } = req.body;

    try {

        await Campground.findByIdAndUpdate(id, {

            title: title,

            location: location,

            image,

            price,

            description

        }, { runValidators: true });

        res.redirect(`/campgrounds/${id}`);

    }

    catch (err) {

        //always return next() so that the code below next() does not execute after next() is executed

        return next(new ExpressError(err.message, 400));

    }

}

module.exports.deleteCampground = (req, res, next) => {

    const { id } = req.params;

    Campground.findByIdAndDelete(id)

        .then(() => res.redirect('/campgrounds'))

        .catch((err) => next(new ExpressError(err.message)));

}

module.exports.showSpecificCampground = async (req, res, next) => {

    const { id } = req.params;

    try {

        let campground = await Campground.findById(id).populate('reviews');

        res.render('campgrounds/show', { campground });

    }

    catch (err) {

        //always return next() so that the code below next() does not execute after next() is executed

        return next(new ExpressError(err.message, 404));

    }

}

module.exports.createCampground = (req, res, next) => {

    const { title, location, image, price, description } = req.body;

    const campground = new Campground({

        title,

        location,

        image,

        price,

        description,

        username: res.locals.currentUser.username

    });

    campground.save()

        .then(() => res.redirect(`campgrounds/${campground.\_id}`))

        .catch((err) => {

            //As this is an asynchronous function, we have to explicitly call next()

            //and pass on the error so that it can be caught by the error handling middleware functions

            return next(new ExpressError(err.message, 400));

        });

}

module.exports.renderNewCampgroundForm = (req, res) => {

    //render views/campgrounds/new.js

    return res.render('campgrounds/new');

}

module.exports.renderEditCampgroundForm = async (req, res, next) => {

    const { id } = req.params;

    try {

        const campground = await Campground.findById(id);

        return res.render('campgrounds/edit', { campground });

    }

    catch (err) {

        //always return next() so that the code below next() does not execute after next() is executed

        return next(new ExpressError(err.message, 404));

    }

}

//function to validate campground data

module.exports.validateCampgroundData = function validateCampgroundData(req, res, next) {

    const { title, location, image, price, description } = req.body;

    //check if any of the fields are undefined

    if (!(title && location && image && price && description))

        return next(new ExpressError("Some Fields are empty!", 400));

    else return next();

}

And routes/campground.js looks like this:

//campground route

//require express

const express = require('express');

//get the router

const router = express.Router();

//get the campground model

const Campground = require('../models/campground');

//require the express error

const ExpressError = require('../utils/ExpressError');

//require isLoggedIn middleware

const { isLoggedIn, isCampgroundOwner } = require('../middleware');

/\* require campground controller

 \* We are following MVC architecture. We have models, views and controller.

 \* The campground controller has the main logic for showing, deleting, adding, updating campground

 \*/

const campgroundController = require('../controllers/campground');

/\* get a form to add a new campground

 \* if an user is logged in, then only the user

 \* can add a campground. isLoggedIn middleware is used

 \*/

router.get('/new', isLoggedIn, campgroundController.renderNewCampgroundForm);

/\* make a post request to add new campgrounds

 \* after making a post request, validateCampgroundData function will run,

 \* if all the data looks good, then the next middleware will be invoked,

 \* or else the error handler middleware will be invoked

 \*/

router.post('/', isLoggedIn, campgroundController.validateCampgroundData, campgroundController.createCampground)

//show a specific campground

router.get('/:id', campgroundController.showSpecificCampground);

//edit a specific campground

router.get('/:id/edit', isLoggedIn, isCampgroundOwner, campgroundController.renderEditCampgroundForm);

//delete a specific campground, This will also trigger the post function defined in the camground.js (models/camground.js) which deletes all the reviews associated with it

router.delete('/:id', isLoggedIn, isCampgroundOwner, campgroundController.deleteCampground);

//put request for editing a campground

//after making a put request, validateCampgroundData function will run, if all the data looks good, then the next middleware will be invoked, or else the error handler middleware will be invoked

router.put('/:id', isLoggedIn, isCampgroundOwner, campgroundController.validateCampgroundData, campgroundController.editCampground);

//show all campgrounds

router.get('/', campgroundController.index);

//export the router

module.exports = router;

**Displaying star ratings and star ratings forms:**

Refer: <https://github.com/LunarLogic/starability>

**SECTION 54: YELP CAMP: IMAGE UPLOAD**

**Intro to image upload process:**

1. Regular HTML files can’t send files to the server
2. We need to store the images somewhere (We don’t store the images in mongo because the BSON documents’ maximum size should not exceed 16 MB)
3. For this project, we use cloudinary (Refer: <https://cloudinary.com/>), which stores images in their database for us. (we can also use AWS and other tools)

Form which accepts files -> take the image file -> store it in cloudinary and get the image url -> store the url in mongo

**The Multer Middleware:**

With our generic forms, we can put a file input in them, but the file information will not be sent properly. This has to with the encoding type attribute on a form.

Refer: <https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/enctype>

A screenshot of a computer

Description automatically generated with medium confidence

To take file input from a form, we have to set the enctype parameter to multipart/form-data.

In the views/new.js, setting the enctype to multipart/form-data in the form

 <form action="/campgrounds" method="post" novalidate class = "validated form" enctype="multipart/form-data">

                <div class="mb-3">

                    <label class = "form-label" for="title">Title: </label>

                    <input class = "form-control" type="text" id = "title" name = "title" required>

                    <div class="valid-feedback">Looks good!</div>

                </div>

….. rest html

But in order to parse multipart/form-data, we need to install **multer** middleware.

Multer middleware is used to parse multipart/form-data

Install the library using **npm i multer**

Refer this: <https://www.npmjs.com/package/multer>

**dotenv:**

Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env. install using npm i dotenv

Refer: <https://www.npmjs.com/package/dotenv>

Now that we have the cloudinary credentials (API\_KEY, API Environment Variable, API Secret), it is a bad practice to write the credentials in our js files.

We have to hide our credentials.

Here, we use .env file in which we write the credentials, and use Dotenv library to read the credentials from the .env file. The .env file is hidden (Means it does not show on file explorer or file picker)

**Uploading to cloudinary:**

For uploading the images to cloudinary and getting the image urls of the uploaded images, we would be using Multer Storage Cloudinary Library. Install using npm install multer-storage-cloudinary

Refer: <https://www.npmjs.com/package/multer-storage-cloudinary>

In the project, we make a folder cloudinary and in the folder, we make index.js file

Cloudinary/

Index.js

Cloudinary/index.js contains the configurations of cloudinary

const cloudinary = require('cloudinary').v2;

const { CloudinaryStorage } = require('multer-storage-cloudinary');

cloudinary.config({

  //get the credentials from the .env file

    cloud\_name: process.env.CLOUD\_NAME,

    api\_key: process.env.API\_KEY,

    api\_secret: process.env.API\_SECRET

})

const storage = new CloudinaryStorage({

  cloudinary: cloudinary,

  folder: 'YelpCamp',

  allowedFormats: ['jpeg', 'png', 'jpg']

});

module.exports = {

    cloudinary,

    storage

}

Do the image update and delete as practice