HANDLEBARS

Let us start our server. We will go back to the root page and we'll be in the login page. The idea is we should get information specific to the user. On the GET route of the home page, we are sending a home.html and in the POST route, it will handle data coming in from the home.html page. Then finally, there is a /profile path, in this, we will show user, there user name id and etc.

In home.html, we have a div with two input elements and one submit button. We have linked one browser.js and add axios strip and bootstrap.

So where are we storing our users? We need something to compare. For this purpose, we have created a **db** folder in which we have a users.json file, in which instead of having an array, we have user ID as key to which all the user details are paired as value.

All this functionality we have to implement. Let us check if we are able to get the details. In the network tab we can see the Request Method: POST in the Headerand hence we are able to capture the data in our terminal.

We are able to see signIn in the URL, but we need the profile page to load. Since the data is working well, we can start implementing the core of the logic.

We need to require the user.json file. So,

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

The userObject has all the details of whatever is inside the user.json file. Let us take a sample email ID and password and give them in the home.html file page for the testing purpose.

```
<div class="mb-3">
    <label for="exampleInputEmail1" class="form-label">Email address</label>
    <input type="email" name="email" value="Karley_Dach@jasper.info" class="form-contorl"
id="exampleInputEmail1" aria-describedby="emailHelp">
```

```
    <div id="emailHelp" class="form-text">We'll never share your email with anyone else.</div>
</div>

<div class="mb-3">
    <label for="exampleInputPassword1" class="form-label">Password</label>
    <input type="password" name="password" value="123abc" class="form-control"
id="exampleInputPassword1">
</div>
```

We would keep the GET request for now, and then let us see that if we click on the submit button, are we getting redirected to the **User Profile**. So, let us put redirect in POST route.

```
app.post('/signIn', (req, res) => {
    console.log(req.query)
    res.redirect('/profile')
})
```

We get redirected to the profile page in the URL and will see **User Profile** in the html window.

Imaging we have 15 routes and all them have similar time of behavior and we need to handle in the DOM manipulation part, so our frontend script would become huge. The requirement of having the /profile, so that it is much more accessible. If we have localhost:3000 and then clicked on submit and we got the user data. Now we bookmarked it and closed the tab and then come back later and open the page again, it would not show the user information in the URL, but will only open the signIn page. We can achieve this using local storage, but having /profile gives us more contextual meaning.

Using react part we can solve this easily. If we write some script which will generate a HTML page for each of the user, then as the user number increases so will the html files and this will take up space in the server environment.

Another way is, after redirecting to the profile, there are some res.header() in which we can set some data. When the javascript load for profile.html, then javascript can do certain stuff with the data received. The idea is again DOM manipulation. We are not dealing with direct html as programming is not possible in HTML.

Template Engines:

If we use a template way and some way of specifying the variable which would contain the value that the user is trying to access would solve the problem. There are tons of template engines. The template approach is not specific only to node and express. This is available in any other language. The syntax and packages would be different.

Template Engines is a way of specifying, which allows you to generate on the go in the server time, with the values that we have provided and then based on the processing that HTML is sent back to the client. Handlebar is one of them. other types are;

- Handlebar
- Ejs
- Pug
- Jade

https://expressjs.com/en/resources/template-engines.html

All of them achieve the same thing, the difference is in their syntax. So, how to use it inside the project?

Like we have defined a public directory, such that every file in it is publicly accessible, by using express.static, there is another convention as well. We will create another folder called views which would contain different views that the user might see. Coming back to handlebars, it is template engine will allow me to do whatever I was trying to do. it has some basic ideas of how to install and how we need to use it. inside HTML tag, if we put something in double braces, the value would be sent to the server.

The official handlebars package is not preferred because there is lot that we need to define ourselves as this works directly in the browser. So, the browser as well as server side put together, plus the handlebar being a superset template language called Mustache. All of this added give us 2.7MB file which would slow down the page. So instead of using this, we would use '**express handlebars**'. It is a wrapper upon the handlebar library itself. We just have to install it and its API and usage is much simpler.

We would first need to install it, then require it and tell express that we want to use handlebars. And then in the second line, app.set to view engine as handlebars. Handlebars would be the extension name as well.

Now, the project depends on express and express-handlebars; 2 packages.

```
const userObject = require('./db/users.json')
const expHbs = require('express-handlebars')
const express = require('express')
const app = express();
```

Now we need to set app.engine. After express() is created, then only after creating the instance of app, then we would put inside it,

```
app.engine('handlebars', express())
```

This will tell express server that every file ending with handlebar extension, we need to use the object of expHbs() gives. Now we need to set view engine.

For that view engine we would specify the file name extension → handlebar itselt.

```
app.set('view engine', 'handlebars')
```

Once express-handlebar is configure, now it is time to start using it. We will go inside the view folder and will create a new file **profile.handlebars**. In this file, we need html. We do not see any errors; this means all the template engines supports html. So we can still have our bootstrap logged into.

Since we have not created profile.handlebar file, instead of sending the profile.html file, we will say res.render(). This render() know that the express would look into view folder for the file name that we would enter in. We have already specified the extension for all the files in the view file. So we do not need to write .handlebars in the res.render(). The text mentioned inside the res.render should be the same as the file name in the view folder.

```
app.get('/profile', (req, res) => {
    res.render('profile')
})
```

To check if this works, we would mention in the profile.handlebars,

```
<h2>We have made handlebar working in the server</h2>
```

There is error because the package is expecting a particular directory structure.

Directory Structure:

```
.
├── app.js
└── views
    ├── home.handlebars
    └── layouts
        └── main.handlebars

2 directories, 3 files
```

We would give another folder **layouts** and inside it we would create a file as **name.handlebars**, and put in some basic html. We would some basic syntax of handlebars, **`{{ body }}`**.

We can see the complete html coming inside the webpage, but it is coming as a string instead being rendered as HTML. We have to give three braces instead of two. So in the main.handlebar file,

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    {{{body}}}
</body>
</html>
```

In profile.handlebar file,

```
<h1>User Profile</h1>
<h2>I have made handlebars working in my server</h2>
```

Now what is does, internally it is specific to handlebar. When we define a particular file name, handlebar goes inside the layout of the html first and then whatever content is inside the profile.handlbars, it puts it inside the main.html file and sends it back.

When you are defining a profile, whatever is directly inside the view folder. Once we send this to the render function, it goes into the main.handlebar file sees the basic layout and just by looking at {{{body}}} it is actually referring to the contents inside the view file or template file which we have specified in the app.get route in the index.js file. This gives us the output as;

User Profile

I have made handlebars working in my server

In the render function a second parameter can also be passed and it should always be an object. Since it is a simple one, we will create a inline object.

```
app.get('/profile', (req, res) => {
    res.render('profile', { name: 'Yash', age: 123 })
})
```

This exact same object is passed into the profile view. The object gets passed into the handlebar view, internally. So, we can say in profile.handlebars, we provide two curly braces '**{{}}**' and provide the variable in it,

```
<h1>User Profile</h1>
<h2>I have made handlebars working in my server</h2>
<h1>Your Name is {{ name }}</h1>
<h1>Your Age is {{ age }}</h1>
```

If we refresh, we can see,

User Profile

I have made handlebars working in my server

Your Name is Yash

Your Age is 123

So, now all we need to do is verify something and send the object to the res.render() in index.js file. We can focus on getting the user and then creating.

We observe that key is the object of the user. This property name is helptful when we are doing restful api. We will have to get the kkeys out and then iterate over them to get the parrticualr objects back. Since it is going to be an object, we need to get the keys name. in index.js file,

```
app.post('/signIn', (req, res) => {
    const keys = Object.keys(userObject)
    console.log(keys)
    console.log(req.query)
    res.redirect('/profile')
})
```

If we refresh, we have to go back to the route page and click on submit. In an array we have

```
[
   "1", "2", "3",
   "4", "5", "6",
   "7", "8", "9",
   "10"
]
```

This means, we are getting all the property names of the route level object. Now we can loop over the keys as it is an array using forEach(),

```
app.post('/signIn', (req, res) => {
    const keys = Object.keys(userObject)

    keys.forEach(prop => {
        const user = userObject[prop]
        console.log(user)
    })

    console.log(req.body)
    res.redirect('/profile')
})
```

Now if we submit, we can see all the details in the terminal. We have email ID inside the req.object.

```
app.get('/signIn', (req, res) => {
    console.log(req.body)
    const {email, password} = req.body
    const keys = Object.keys(userObject)

    keys.forEach(prop => {
        const user = userObject[prop]

        if(user.email === email) {
            if (user.password === password) {
                // user exist
                res.redirect('/profile')
                return false;
            }
        }
    })
})
```

If the user email and password matches, then we can redirect it to the profile page. We will put ***return false***, this will terminate the loop from executing afterwards. If we don't put this line, it will go to the next object and keep searching.

What if there was no user existing?

```
app.get('/signIn', (req, res) => {
    console.log(req.body)
    const {email, password} = req.body
    const keys = Object.keys(userObject)
    let userFound = false;

    keys.forEach(prop => {
        const user = userObject[prop]

        if(user.email === email && user.password === password) {
            // user exist
            userFound = true
            res.redirect('/profile')
            return false;
        }
    })
    if (userfound === false) {
        res.send('User Not Found')
    }
})
```

We are getting redirected to the profile when we enter the right values. So, our basic authentication is working.

Since we know the user is found, as an object we will pass it around. Then we just have to use it in the handlebars. The problem is we cannot pass data as redirect(), meaning I'm able to redirect, but cannot provide the data. There is a hack in achieving it, but we cannot use it in production level.

Using query parameters, we will convert it into a template string and then say,

```
app.post('/signIn', (req, res) => {
    console.log(req.body)
    const {email, password} = req.body
    const keys = Object.keys(userObject)
    let userFound = false;

    keys.forEach(prop => {
        const user = userObject[prop]

        if(user.email === email && user.password === password) {            // user exist
            userFound = true
            res.redirect(`/profile?name=${user.name}`)
            return false;
        }
    })
    if (userfound === false) {
        res.send('User Not Found')
    }
})
```

We able to see the user name in the URL. But we can see the query parameter inside the GET route of profile, will it not be easier if we found a way and directly pass the userObject and will able to use in the profile.handlebar file.

In the profile itself we are trying to get from the body and instead of redirecting,

`res.redirect(`/profile?name=${user.name}`)` we will wirte, `res.render('profile', user)`

We dint have any way to pass the redirect () so we moved all the logic inside profile itself. So, by this we don't have need for app.post route anymore which means in our home.html, we will change from

`<form action="/signIn", method="POST">` to `<form action="/profile", method="POST">`

With this, we can change the GET to POST in the index.js file. We will getting the entire data in the req.body and only difference would be we will be getting userObject and the output would be

User Profile

I have made handlebars working in my server

Your Name is Mrs. Dennis Schulist

Your Age is

We defined ***one route & one profile.handlebar*** and we can handle users appropriately irrespective of number of users. Now we have ability to create dynamic pages.

We can link up different CSS or make our own dynamic page with different styles.

In user, we have the entire object so from profile.handlebar we are able to reference the user details. We will now take phone number and add it to the file along with the website.

```
<h1>User Profile</h1>
<h2>I have made handlebars working in my server</h2>
<h1>Your Name is {{ name }}</h1>
<h1>You mobile number is {{ phone }}</h1>
<h1>Your website is {{ website }}</h1>
```

We see the address itself is another object. So how do we get the address part? We will use the address tag,

```
<address>
    <p>{{address.street}}</p>
    <p>{{address.suite}}</p>
    <p>{{address.city}}</p>
    <p>{{address.zipcode}}</p>
</address>
```

To get the latitude and longitude details of the user,

```
<address>
    <p>{{address.street}}</p>
    <p>{{address.suite}}</p>
    <p>{{address.city}}</p>
    <p>{{address.zipcode}}</p>
    <p>{{address.geo.lat}}</p>
    <p>{{address.geo.lng}}</p>
</address>
```

This is how we access the nested object. Array also can be done in the same way, but we cannot use map function.

The values for latitude and longitude will not be correct because we need to add another logic to it.