**Documentation**

**GetMeACoffee**

**Table of Contents:**

1. Tech Stack
2. Setup
3. Frontend Development
4. Backend Development

**Tech Stack:**

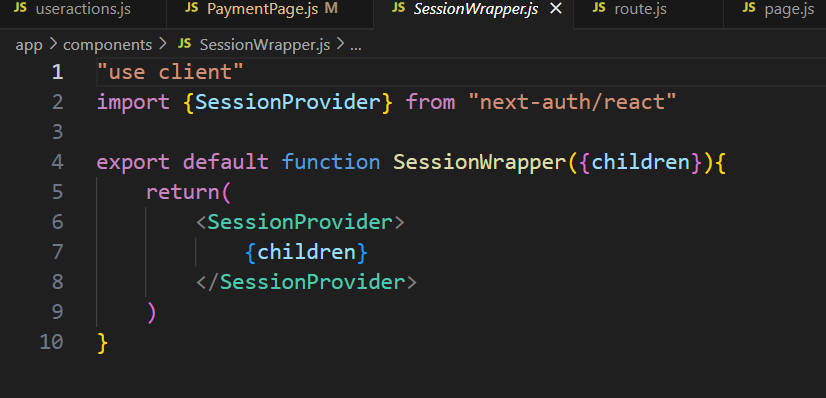
* Next JS [JavaScript, HTML, CSS, React, Node]
* Next-Auth JS
* Javascript
* MongoDB
* Razorpay API

**Setup**

1. Create next app (use JavaScript, tailwind, include react-router-dom).
2. Download necessary packages (mongoose, cors, razorpay, nextauth), also use npm i.
3. Download react-toastify to view alerts/tests.
4. Create a .env.local file just below root directory (under getmeacoffee/name of project).
5. Give your data for these fields.
   1. GITHUB\_ID=
   2. GITHUB\_SECRET=
   3. NEXT\_PUBLIC\_KEY\_ID=
   4. KEY\_SECRET=
   5. NEXT\_PUBLIC\_URL=
   6. NEXTAUTH\_URL=
   7. NEXTAUTH\_SECRET=

**Development (Frontend and Auth backend for GitHub)**

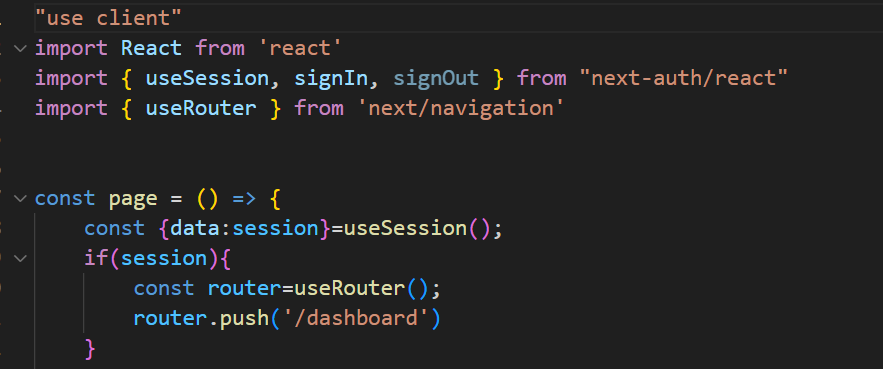
1. In the layout, we will have navbar, children (default) and footer.
2. In Navbar we will include.
   1. Logo (redirects to landing page).
   2. When not signed in:
      1. Sign in button
   3. When signed in:
      1. A drop-down menu
         * 1. Before activating drop down you can see logged in username.
           2. When dropdown is activated, you can see 3 options.
           3. Dashboard (you can save your details here).
           4. Profile page (donation page).
           5. Sign-out.
3. In footer, it’s the generic stuff.
4. In the main page.js, there will be landing page content including few links and a video.
5. Next step is implementing **authentication**, we will make use of next-auth for this task.
6. As of now we will setup only GitHub and follow the documentation to set it up.
7. We first create a **session wrapper** under components.



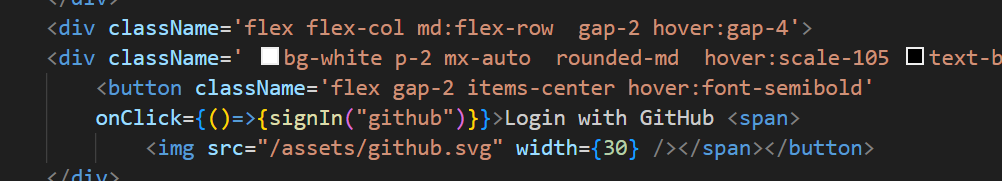
1. Use the session wrapper on layout.js components like this.



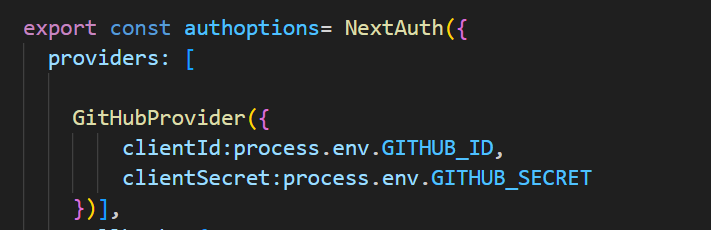
1. Create login page.
   1. Here with the help of useSession, signIn and signOut from next-auth we will enable authentication.



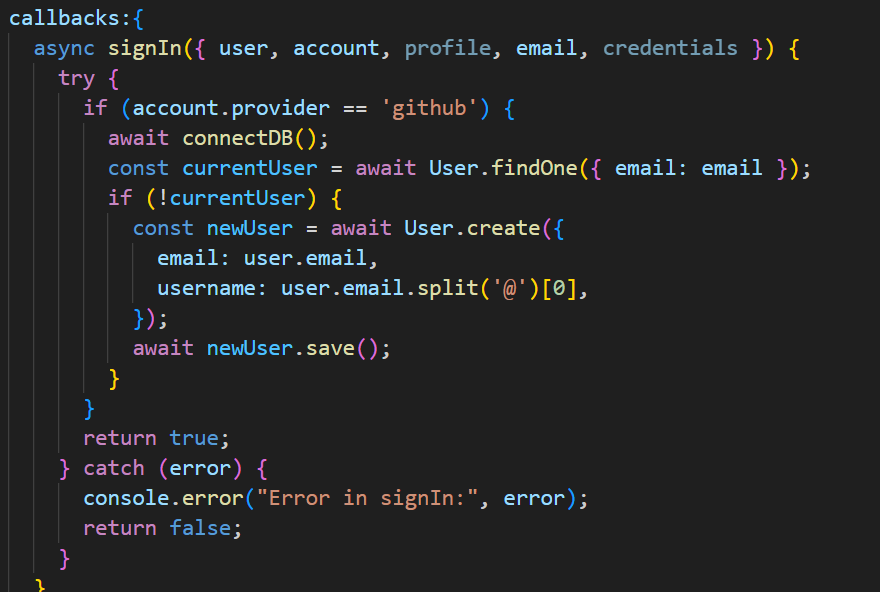
* 1. The above session data is used to see if the user has logged in, if yes, they are redirected to dashboard.
  2. Github login.

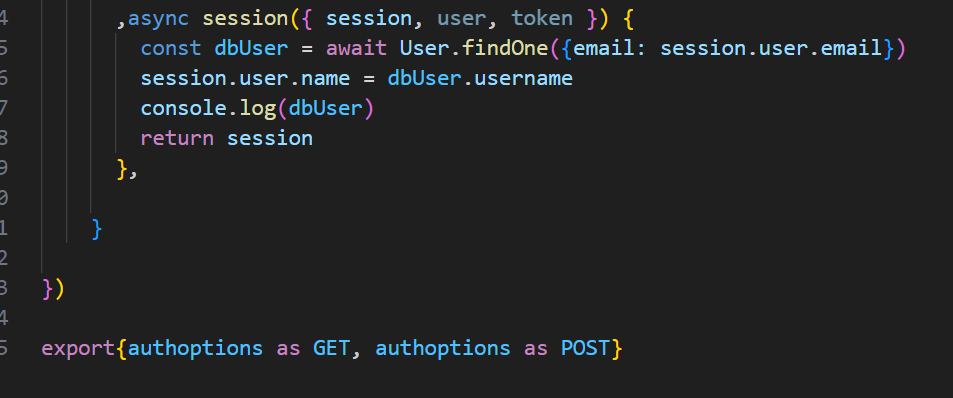


* 1. For the backend part we create a folder structure as follows. api/auth/[…nextauth]/route.js
  2. In route.js we specify the providers (in this case github) like so.



* 1. Then to tell next-auth what to do after a successful login. We write **callbacks** under providers.





* In the above code, we receive details from github after login.
  + We first check if the received email is present in the database.
  + If no, we create a new entry into the db and register the user and then return true.
  + If yes, we simply return true.
  + Authentication will fail if true is not returned.
* In the session function we retrieve the users name with the help of their email in the database and assign it to **session.user.name** for use in other pages.
* We finally return the session object.
* Then we export the main function as GET and POST.

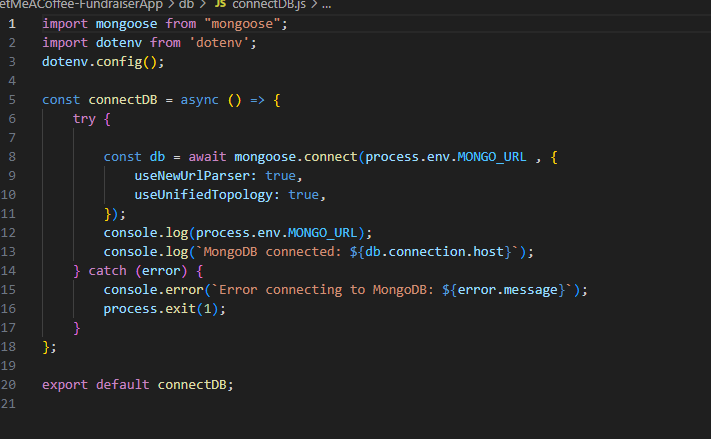
1. Dashboard
   1. This is the page user (person who wants donation) is redirected to after login.
   2. Here the user can:
      1. Add/edit name, username, cover picture, profile picture, razorpayid and razorpay secret.
2. Donation page/Profile page
   1. The endpoint of this URL will be the user’s name.
   2. In this page you can:
      1. Donate
      2. View the person/profile you’re donating to.
      3. See a list of supporters, number of payments and total received payments all fetched from backend services.
   3. Before the page loads, we check if the username exists in our db, if no, a **404** page is shown.

**Backend development**

1. **Setting up mongoDB Atlas**

* Go to mongoDB Atlas, register/login. Select your desired cluster (free was chosen in this project).
* Go to dashboard and connect to your cluster, MongoDB Atlas will provide you with a connection url.
* Copy and paste the link to your. env.local file inside the root folder (GetMeACoffee). Like MONGO\_URL=”your conn url”.

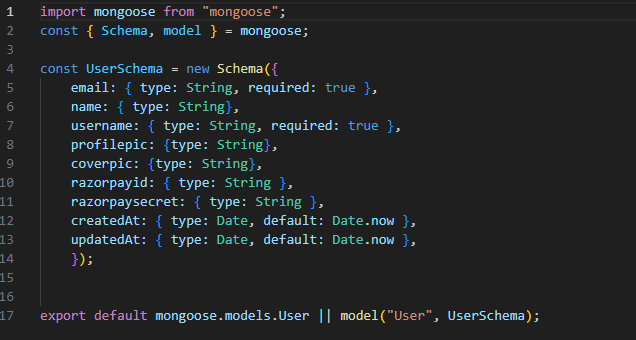
1. **Connecting to the database**

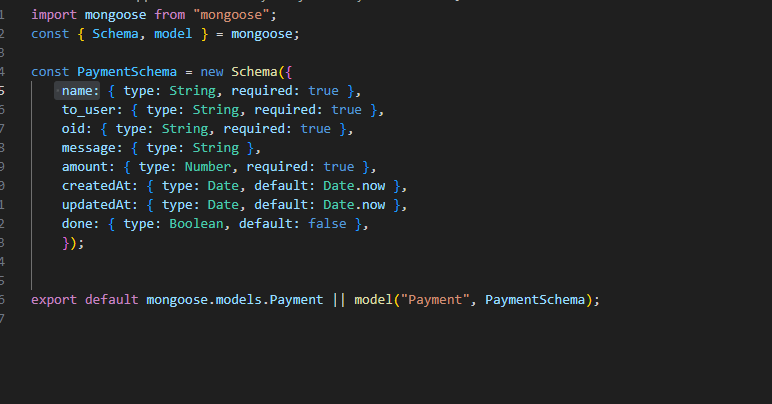
* Create a folder **db** outside app folder.
* Create a connectDB.js under db.
* Inside connectDB write the following. 
* The console logs help in debugging if any issue arises.

1. **Define your models**

We are using two models in this project.

1. User model with params (email, name, username, coverpic, profilepic, razorpayid, razorpaysecret, createdAt, updatedAt).
2. Payment model with params (name, to\_user, oid(orderid), message, amount, createdAt, updatedAt, done).

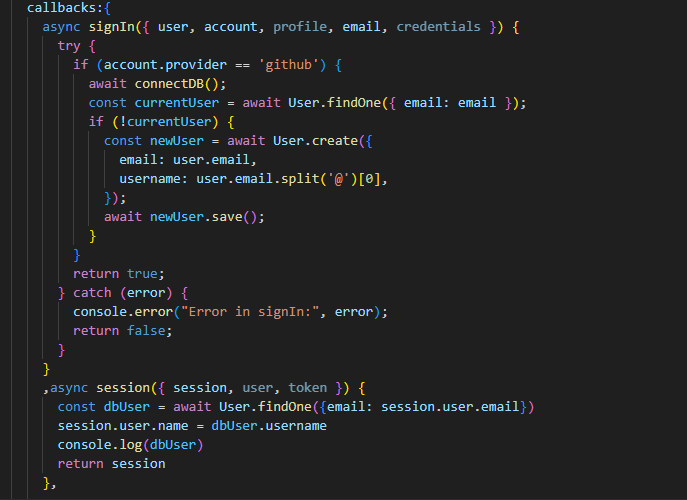


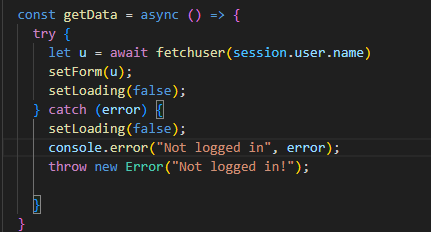


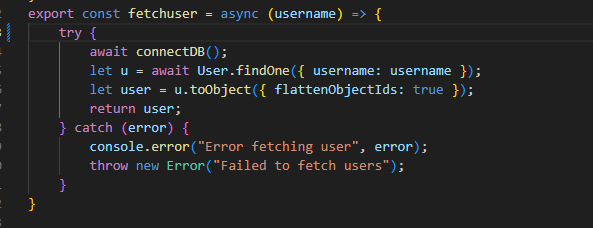
1. **Backend of login page**

* When we click on log in with GitHub, the signIn function of next-auth gets called and we are redirected to github for login.
* Once the authentication is done, user is redirected to dashboard.

1. **Backend of Dashboard**

* When the user is redirected to dashboard, a useEffect is triggered that waits for authentication of data to be loaded into the session and starts a loader.
* Once auth details are available, data is fetched from the backend to populate the dashboard form.
  + Email and username are filled by default.
  + How are they filled? The key is in api/auth/[…nextauth]/route.js.
  + In the above file, we have defined **callbacks** that tell auth what to do after a successful authorization. (This code has been shown earlier).
  + So, what does the above do?
    - We first check if account provider is github.
    - If yes, we check if the user exists.
    - If yes, return true so that auth can move to next step.
    - If no, register the user automatically, assign a username by extracting text before **@** of email.
    - Example-> email is [xyz@gmail.com](mailto:xyz@gmail.com), username is “xyz”.
    - After the above is done, return true.
    - In the next step, we must populate session data.
    - We get the username from db and assign it to session.user.name (session’s username).
    - Return the session.
  + Once the session data is loaded, **getData()** is triggered.
  + getData() is an async function that gets all related userdata (like email, name, username, coverpic, etc) from database based on the **session.user.name** which was returned by the **callback.**

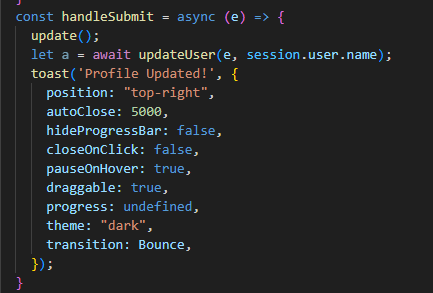


* + getData gets data from backend by calling a **backend** function called fetchuser defined under **actions/useractions.js **

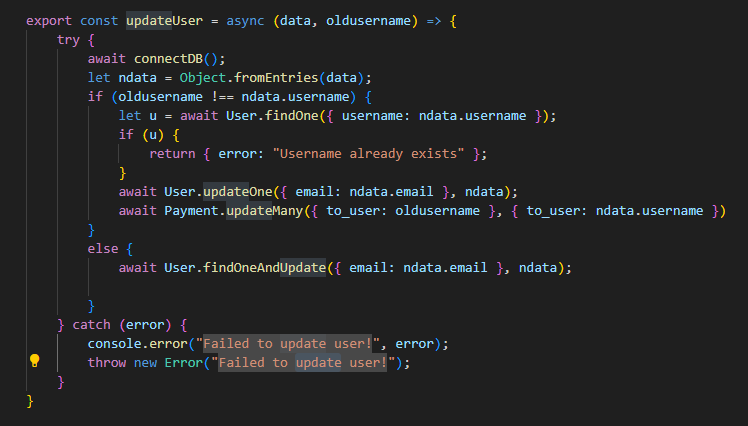
* If the user updates any data in the dashboard form, this **action** triggers a function called **handleSubmit.**

****

* **handleSubmit** is like so

****

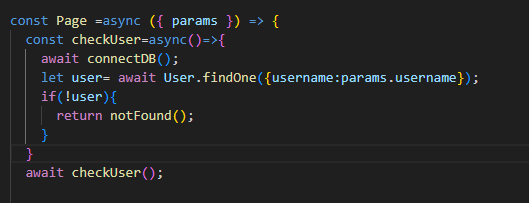
* In the handleSubmit:
  + The update() function in useSession is used to update the session data on the client side.
  + When you call update(), it sends a request to the session endpoint (/api/auth/session) to refresh the session data, after changes are made to the session (e.g., updating user details or roles).
  + Then, a backend function **updateUser** is called , its located in the same place as fetchuser.



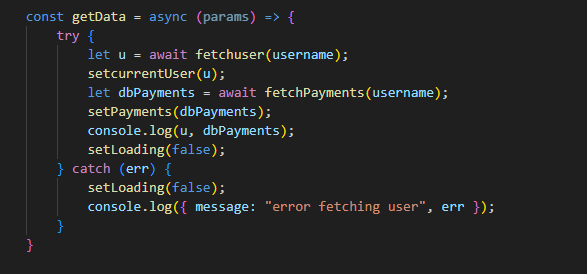
* + In this function, we receive receive, the form data (e from handlesubmit which contains form data and old username from session).
  + We check if old username is same as new username.
  + If yes, just update the other fields keeping email same as it used for lookup.
  + If no, check if the received username is present in db, if yes, return error that user exists.
    - If username doesn’t exist, update other fields using email for lookup.
    - Update payment details associated with oldusername to new username.

1. **Backend of user page**

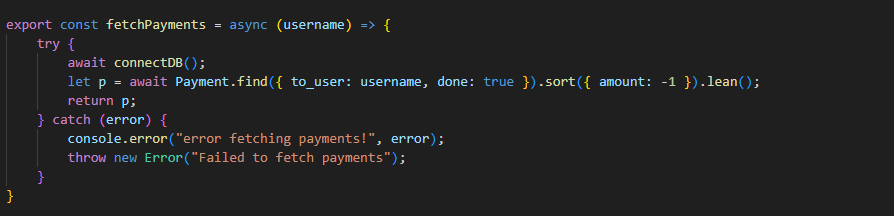
* Before loading the user, we check if the user exists (username).



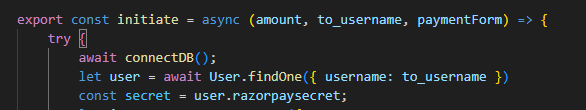
* If no, a 404 page is shown.
* If yes, the profile is shown.
* When the user page loads, a react component, PaymentPage is shown with a username passed from params (basically from the searchbar) as props.
* PaymentPage receives username as props.
* When this component mounts, a useEffect is triggered, that calls **getData()** which is not to be confused with the getData of dashboard, both are independent **local** functions.
* This getData() fetches userdetails as well as payment details like so.



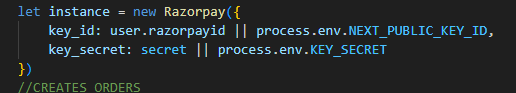
* currentUser and Payments are useStates that are defined in the code.
* Fetchuser is the same useraction backend function as discussed earlier.
* fetchPayments is like so, located in the same loc as fetchuser.



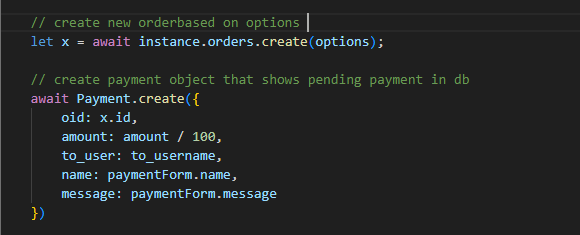
* In the above code, we simply fetch payments whose status is done and sort them in descending order of payment value.
* Payment backend- what happens when you click on donate now or any preset payment amount after filling in required details?
* When you click **Donate now** or **Donate re.10 etc.**
  + A function called **pay** is activated, it receives a number as a prop.
  + Once the function is called,
    - We call a function called **initiate** which is located in useractions and store this value in a variable called a.
    - In useractions,
    - First we find the associated username to which the payment will be going to.
    - And, we extract their razorpay key and secret.



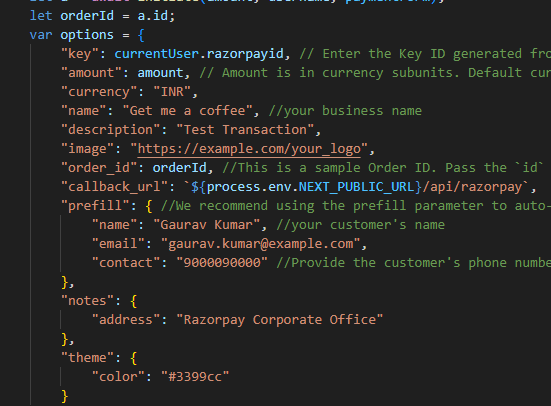
* + - new Razorpay(...) creates an object (instance) of the Razorpay class or library.
    - The constructor of the Razorpay class takes an object with configuration details (like key\_id and key\_secret) as input.
    - The instance now represents a connection or client for interacting with Razorpay's API.
    - instance is used to call Razorpay's methods, such as creating orders, verifying payments, etc.



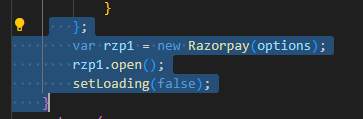
* + - The below code options object is passed as a parameter to a Razorpay API method (e.g., instance.orders.create(options)), specifying details of the order to be created.



* + - We then create an object with the above details and assign it to variable x and then return it.
    - Back in payment page.
    - These are pre-set key objects provided by razorpay for testing.



* + - Finally, we create a new razorpay instance using options and sets up the razorpay interface for checkout.



We have not explained the minute component details, this documentation covers only core features and functionalities of the website.