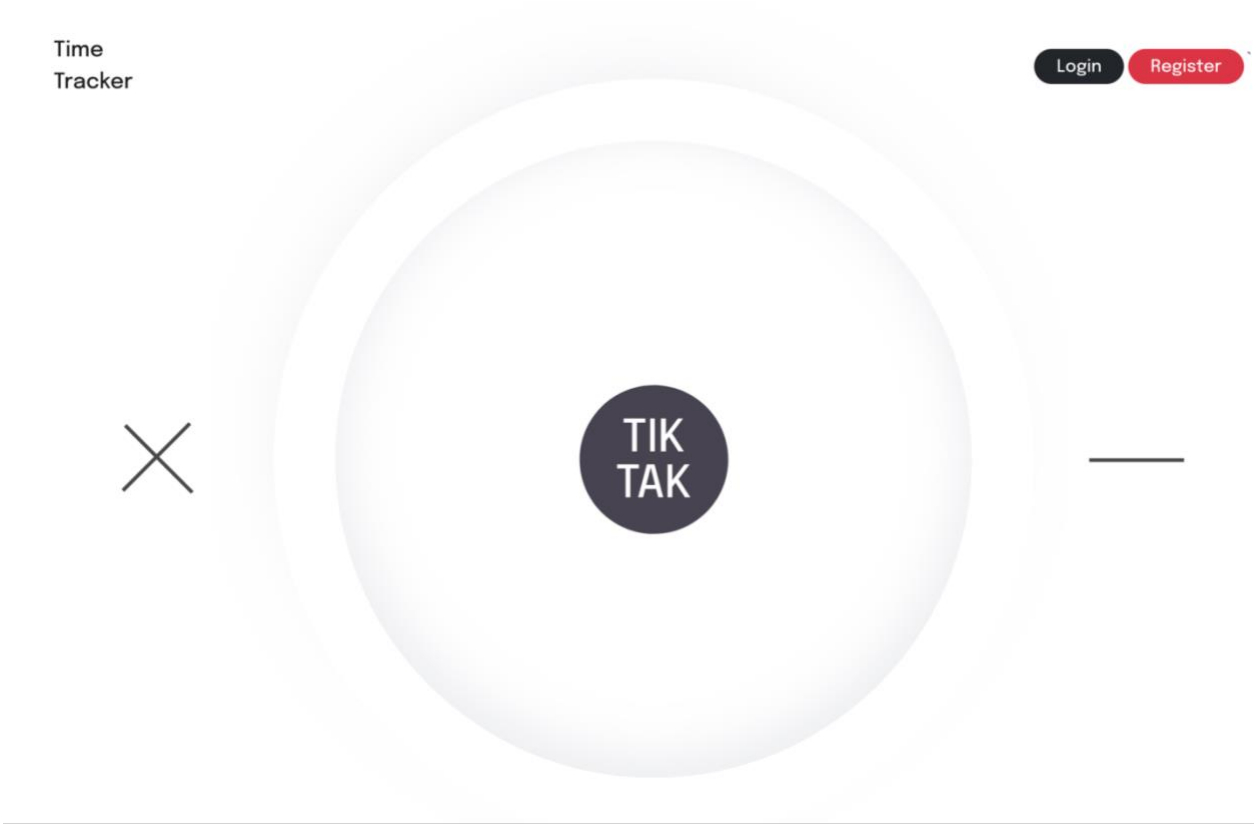
TIK TAK



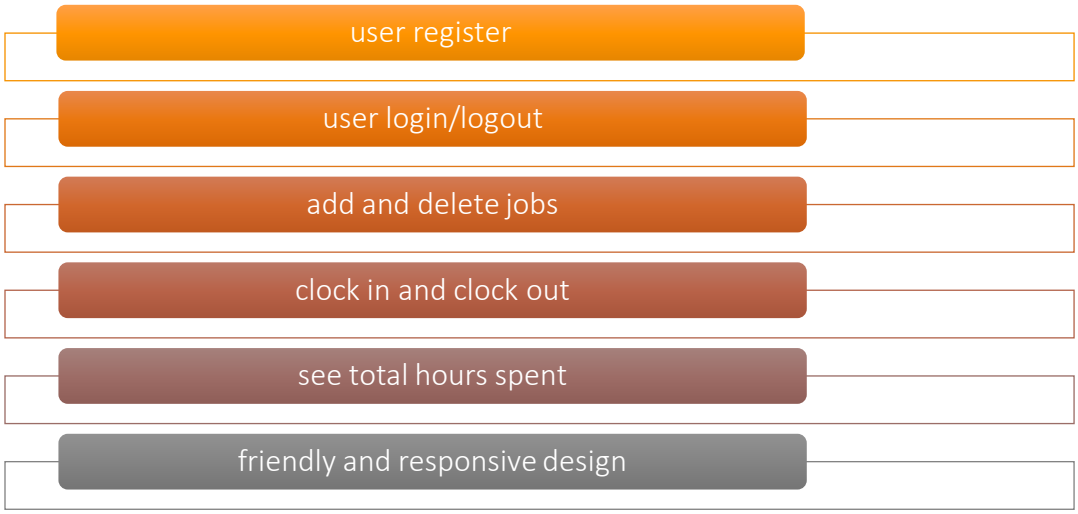
Prof. Vladimir Georgiev

Kristina Myftaraga

MERN

Tik Tak is a platform which allows users to manage the time they spend working on many projects or jobs.

Main functionalities:



Structure

I have used MERN stack to develop this application.

MONGO DB

Models:

- *Users (first name, last name, email, password, jobs)*
- *Jobs (name, user, punches, punched_in)*
- *Punches (job, user, active, timestamps)*

Un-relational database. All models are stored as documents in the database. That is why all the modes have a two-way relationship (example, users have a jobs field, and jobs have users' field)

EXPRESS

I have used Express API as a backend service for this project.

Three main directories are:

MODELS	CONTROLLERS	UTILS
Contain files, each file containing Model's information	Contain one file for each of the model-related endpoints. Example: users controller contains the endpoints for user registration and retrieving users data from the API. Punches controller contains the endpoints so the user can punch in and punch out.	Contains the configuration settings Connection with the mongodb database and the middleware Extracting the token from requests (– will explain this point in more details further in the project)

User Authentication

a) *For first time users who need to **register**, I have created an **endpoint /api/users** which expects a POST request with the following information:*

```
{email, first_name, last_name, password}
```

- *The password is hashed using the **bcrypt** library offered by node.*
- *After retrieving that information from the request, we save it in the database:*

```
const user = new User({
    email: data.email,
    first_name: data.first_name,
    last_name: data.last_name,
```

```
    passwordHash,
})
const createdUser = await user.save()
```

- *If all the fields are valid as we defined in the Mongoose model, then the object will be saved in the database and the variable createdUser will contain the registered User object.*

b) *For user **login**, I have created **the endpoint /api/login** which expects as request body and email and the password.*

- *First, we check for some user in the database with that email.*
- *If there is a match, then we again use the **bcrypt** library to hash the requested password and compare with the hashed password stored in the database.*
- *If these two passwords do not match, then a 401-status response is returned with error message:*

```
return response.status(401).json({
      error: 'invalid email or password'
    })
```

- *If the passwords match, then I have used the **jwt** library of node that takes the user object, and a Secret string as a second parameter to generate the token. This token will later be used for authorizing of the requests made by users.*

```
const userForToken = {
    email: user.email,
    id: user._id,
  }
const token = jwt.sign(userForToken, process.env.SECRET)
```

In the end, the response will contain the user information and the token generated:

```
Response
    .status(200)
    .send({ token, email: user.email, first_name: user.first_name })
```

User Authorization

Making direct requests to the api without an authorization header would work for endpoints like user registration, or user login. But when a user is logged in and tries to do some actions that is in some way personalized would not work. The user requests need a way to be recognized. This is achievable from the jwt token that is generated in the login endpoint. If the user somehow saves that token and later sends a request containing that token, the controller can use this token to verify the user who sent that request. But how does the controller retrieve that token?

*That is when the **middleware** comes into play: I have declared a middleware called tokenExtractor which as the name suggests, extracts the token from the request. This token is the same as the one created earlier in the login request and can be found in the authorization header.*

```
const tokenExtractor = (request, response, next) => {
  const authorization = request.get('authorization')
  var token = null
  if (authorization &&
authorization.toLowerCase().startsWith('bearer ')){
    token = authorization.substring(7)
  }
  // Assign to the request to later use it in the blogRouter
  request.token = token
  next ()
}
```

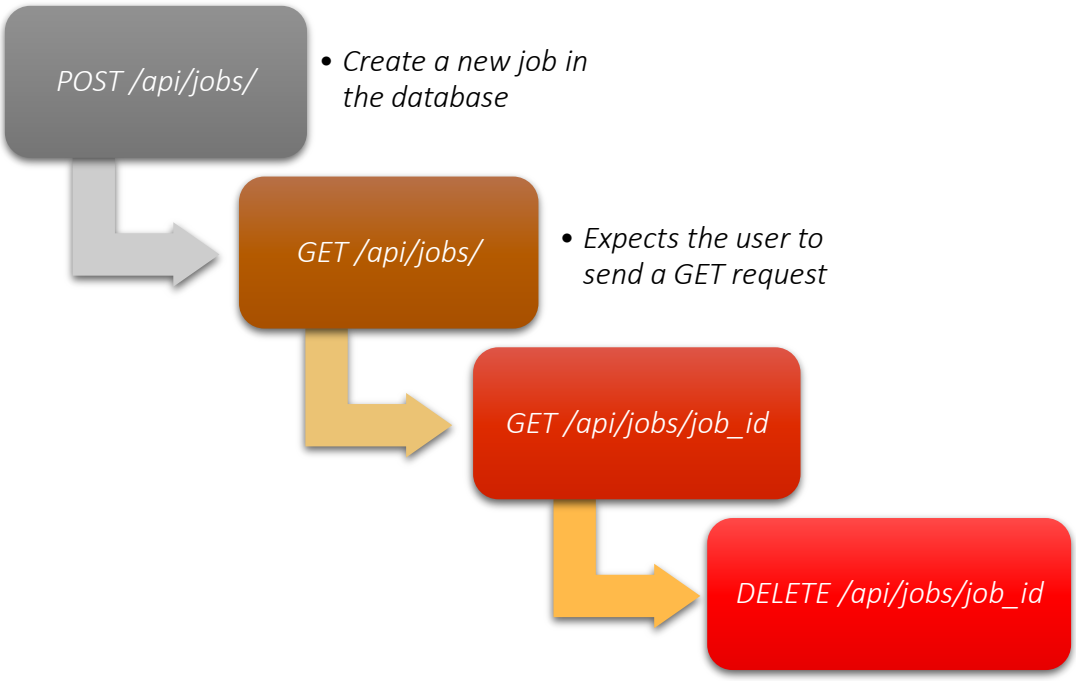
After this middleware is executed, then the request will contain the token that can be easily retrieved in the controller. For retrieving the user from request, I have implemented a helper function that can be reused throughout the project, because it is a very common functionality.

```
const get_user_from_request = async (request) => {
  const decoded_token = jwt.verify(request.token,
process.env.SECRET)
  if ( !request.token || !decoded_token.id ){
    return null
  }
  const current_user = await User.findById(decoded_token.id)
  return current_user
}
```

*The function jwt.verify is from **jwt** node library and it is the opposite of **jtw.sign**. Having the token and the secret string, it will give us an object containing the user. From there we can have its id. Taken the request as an argument, it tries to return the user object from the database and returns null if the token does not match any user id.*

Jobs

*As for the **jobs** controller, I have created the following endpoints:*



- ***POST /api/jobs/** to create a new job in the database*
    - *The endpoint will expect a post request containing the title of the job. When initializing the object to be stored in the database, I have put some default values in the create payload:*

```
const new_job = new Job({
      title: data.title,
      punches:[],
      punched_in: false
   })

new_job.user = curr_user
curr_user.jobs = [...curr_user.jobs, new_job]
```

- *The code is first creating the job object with the title that was received in the payload. The punches array is empty and punched_in field will by default be false.*
- *After this is created, the job also needs a user to be assigned. For that, the controller needs to retrieve the user information from the request. As mentioned earlier, I have created a get_user_from_request function that can help:*

```
const curr_user = await get_user_from_request(request)

if (!curr_user) return response.status(401).json({
error:'missing or invalid token' })
```

- ***GET /api/jobs/***
  - *This endpoint just expects the user to send a GET request. If a valid token is present in the authorization header, then this function will return all the job objects in the database that are related to the user who did the request.'*

- ***GET /api/jobs/job_id***
  - *This endpoint is similar to the above, but instead of returning all the jobs at once, it also expects an extra param in the endpoint url job_id, which would correspond to the id of the requested job object. If that id is valid, then the function will try first to retrieve it from the database. For simplicity reasons, I have also included the punches related with that job to be included in the payload.*
  - *This is what is returned from this endpoint:*

```
const object_to_return = {

    title: curr_job.title,

    punches,

    punched_in: curr_job.punched_in
```

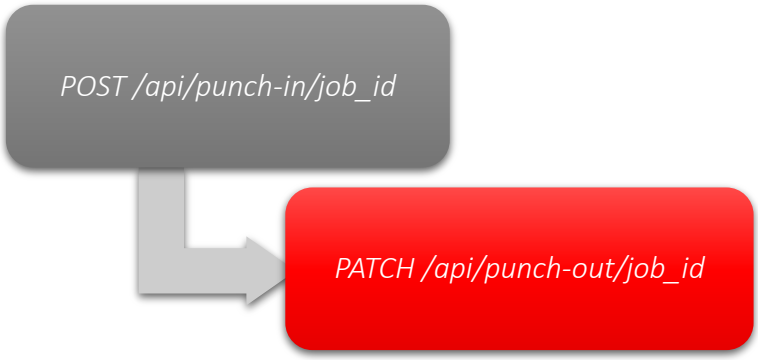
```
}

response.status(200).json(object_to_return)
```

- ***DELETE /api/jobs/job_id***
    - *As the request name suggest, this call will try to delete the job object with id = job_id. First it finds the job from database, and if it exists it will try to delete the job along with all the punches that were related to that job.*

Punches

The punches controller is very straightforward. It consists of two endpoints, one to create a Punc, and one to punch out (which means modifying the Punch object).



- ***POST /api/punch-in/job_id***
    - *This endpoint expects a job id as a request parameter and a valid authorization token (form authorization header) and it will create a Punch object.*
    - *If the above are valid, the creating the Punch will look as follows:*

```
const new_punch = new Punch({
    job: curr_job,
    user: curr_user,
    active: true
  })
  await new_punch.save()
  curr_job.punches.push(new_punch)
  curr_job.punched_in = true
  await curr_job.save()
```

- *Except creating the Punch object with the proper defaults (active=true), it is also important to modify the job object to relate to that punch. That is achieved here:*

```
curr_job.punches.push(new_punch)
curr_job.punched_in = true
```

- ***PATCH /api/punch-out/job_id***
  - *In this endpoint I don't create a new object, instead, mark the job object as not punched_in, retrieve the last active punch of that job and mark it as not active.*

```
curr_job.punched_in = false
await curr_job.save()
const curr_punch = await Punch.findOne({
    job: curr_job,
    active: true
})
curr_punch.active = false
```

React Frontend

*I have used **reactJS** as a library for the frontend.*

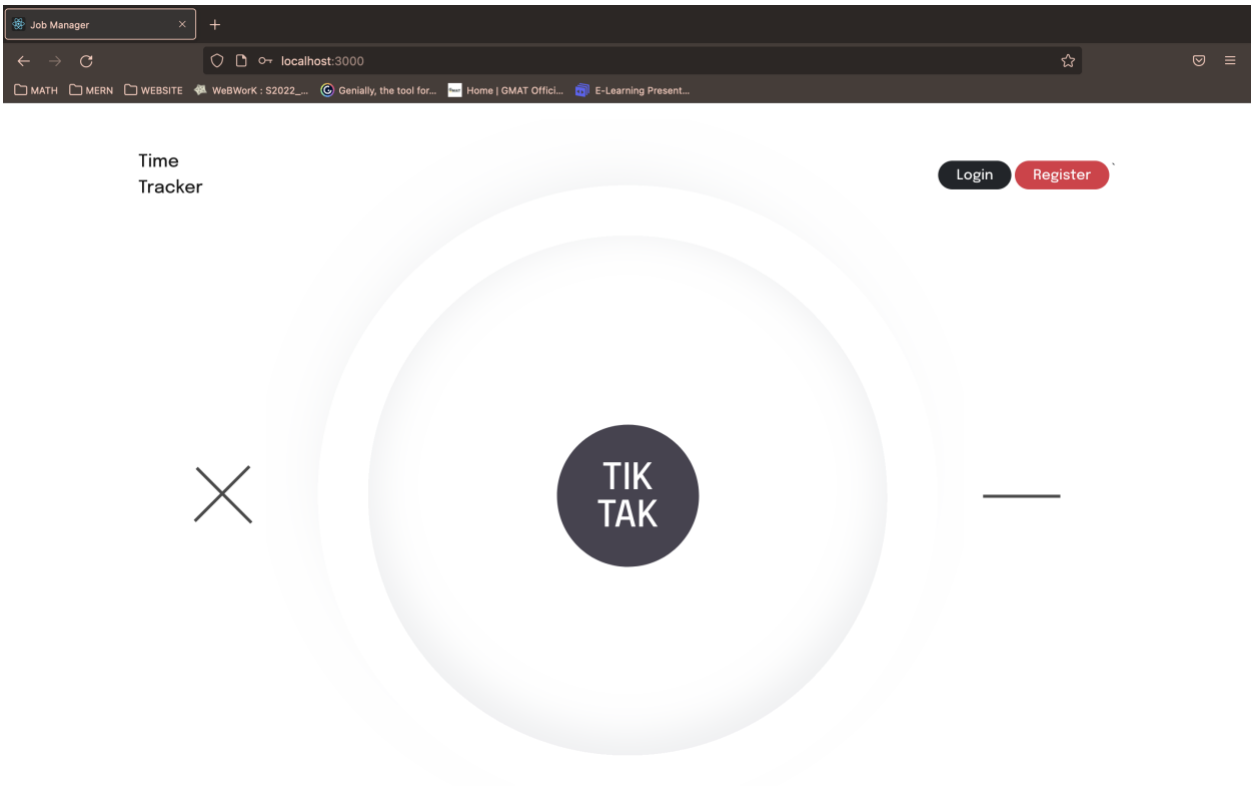


Figure 1 Landing page

For starting the react project template, I used the create-react-app command. In a matter of seconds, it generated a working project structure. I deleted some of the files because not all of them because I wouldn't need some features like testing etc.

Everything in the react app starts with the index.html file in the public folder. React is a single page application library so everything will ideally be rendered in a single file. It is neither developer or user friendly implement everything in the same page, so react has also some libraries like react-router, that stimulates routing.

I have used the following project tools and structure:

- ***React Query:** Data fetching library that is based on using hooks. I chose this over redux, because it is simpler to use and the fact that is using hooks makes it*

more reusable throughout the code and offers a lot of functionalities (like making event handling – errors, success, loading – very accessible).

- *For storing some of the data globally, like the user information or system messages, I have used the micro library:* ***recoil*** *and recoil persist. Again, I chose it because of its simplicity.*
- *Structure of the code:*
  - *Components: contain all the components and page used in the project.*
  - *customHooks: stores all the hooks for fetching the backend data*
  - *services: stores the functions that custom hooks use to communicate with backend*
  - *styles: css or scss files for styling.*
  - *other helpers: contain files like App.js, index.js.*
- *Data validation: Used formik Library. It is used to validate the input fields and gives immediate feedback given a validation schema.*

Authentication

First thing I worked on with this project was the user authentication: Register and Login. For this, I had to consume the already implemented endpoints of the express API. I created a folder named /services where I stored all the files that deal with backend communication. A very simple library that npm provides is axios – it is very simple to make calls with the backend with just a line of code.

```
export const handle_register = async (payload) => {
       const {data} = await axios.post(`${base_url}/users`, payload)
       return data}
```

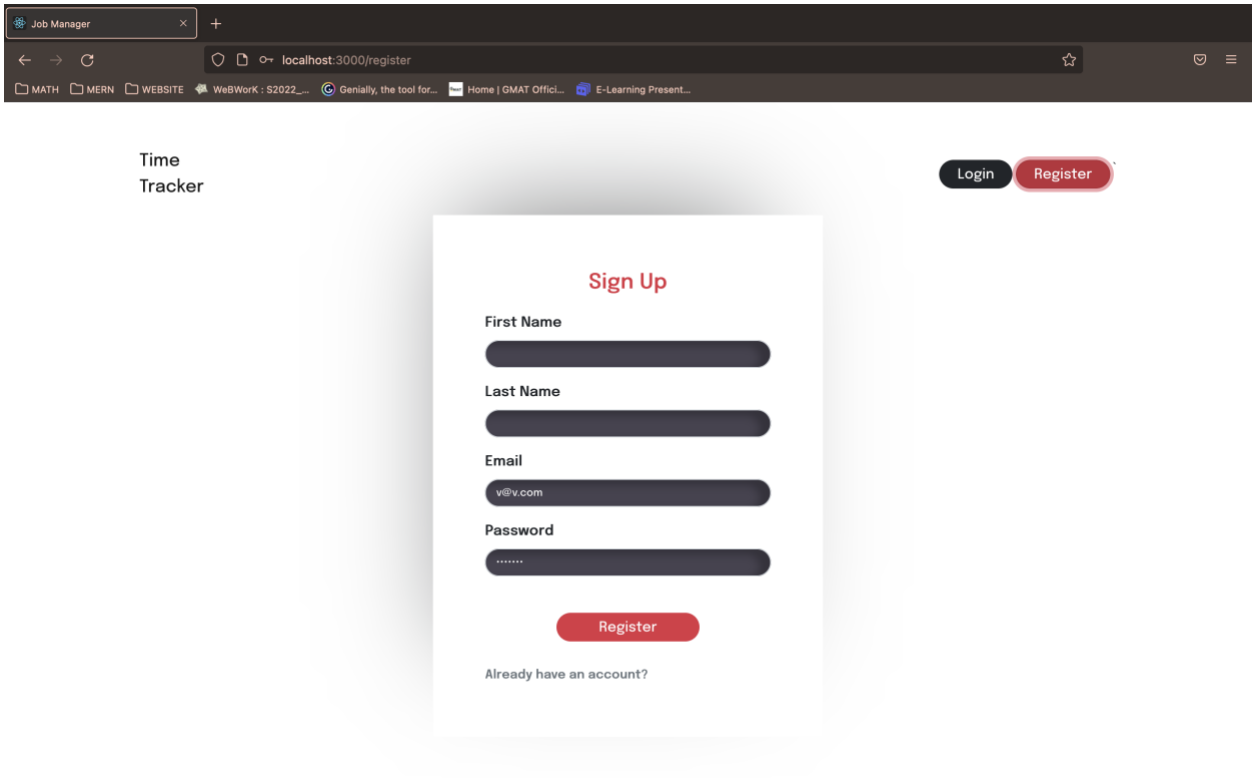


Figure 2 Register page

This is the form I have used to collect user information.

- ***Login**: When consuming the login endpoint from the backend, the frontend will get the response back with the user information and the token. This token needs to be saved somewhere. I chose to store it in the window localstorage. The login form is similar to the sign up form. When the form is filled and submitted, the following react query hook will execute:*

```
const  useLogin = () => {
const set_user = useSetRecoilState(user_state)
const set_error = useSetRecoilState(error_message)
const set_success = useSetRecoilState(success_message)

return  useMutation((payload)=>handle_login(payload), {
   onSuccess: data => {
```

```
            set_user(data)
            set_success('Successfully logged in!')
            window.localStorage.setItem('token',data.token)
            setTimeout(() => {
                  set_success('')
            }, 4000);
      },
      onError: () => {
            set_error('Invalid credentials!')
            setTimeout(() => {
                  set_error('')
            }, 4000);
      }})}
```

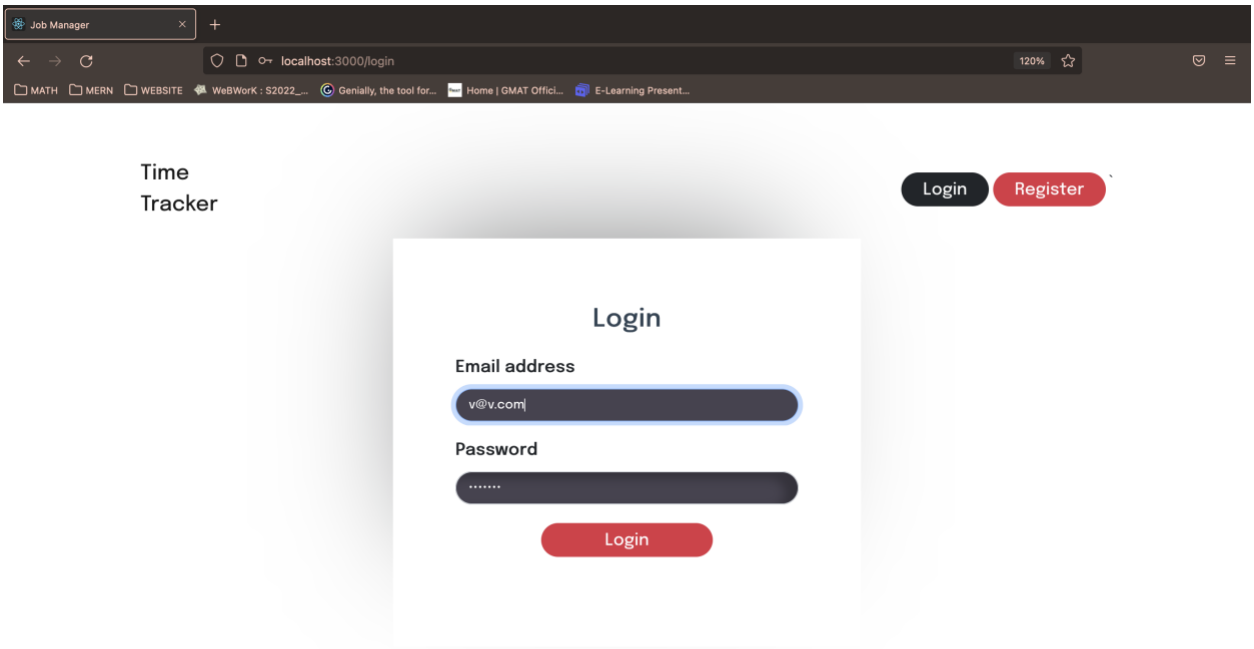


Figure 2: Login form

- *React query mutation offers by default callbacks like onSuccess and onError. In case of success, the data argument will contain the user data (along with token). This token is stored in the localstorage, user data in recoil, and the proper success message is showed to the user:*

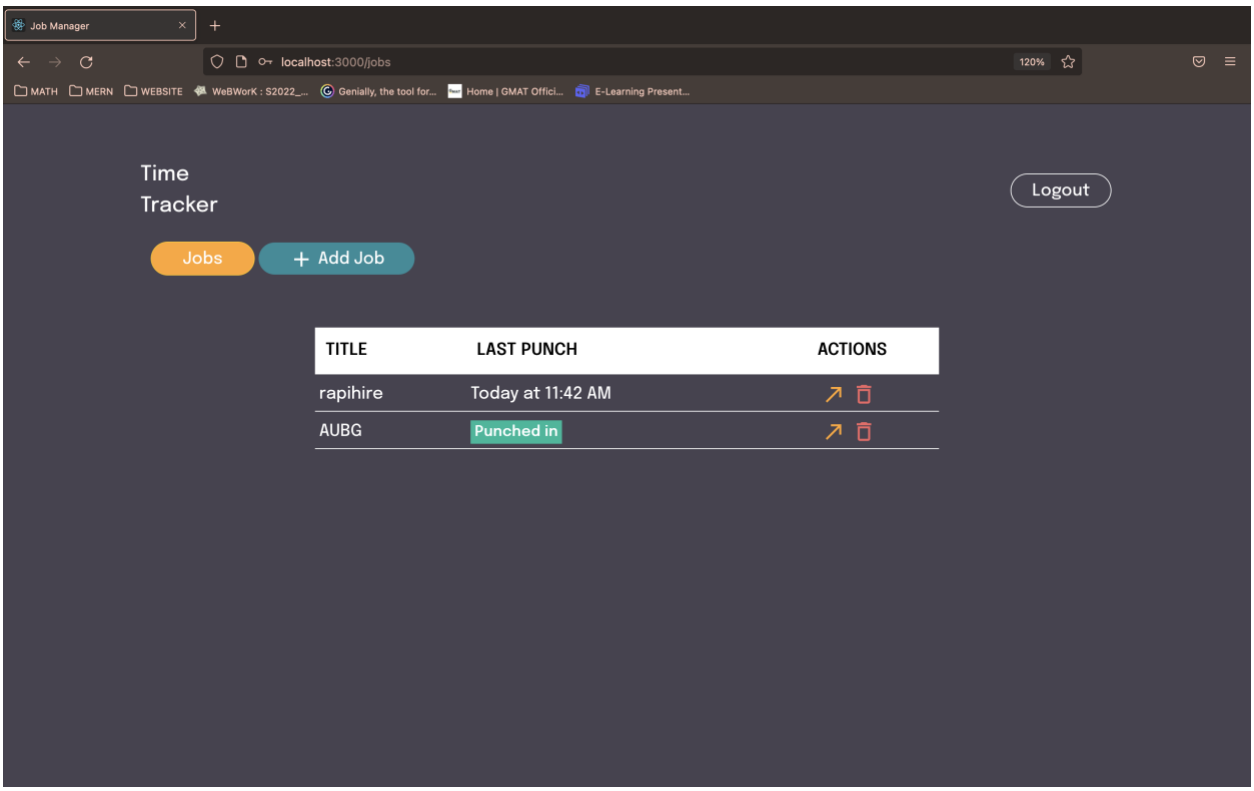


Figure 3 Home Page where users can check the jobs listed and the last punch in time. They can also navogate through the other pages such as adding another job, deleting jobs, or exploring the times spent on each project

Jobs and Punches

When in the home page, the user is shown the jobs added. Again, for fetching the jobs, react query hooks are used:

```
const {
        data: jobs_data,
```

```
    error,
    isLoading,
    isFetching,
  } = useMyJobs()
```

useMyJobs() is a hook similar to the useLogin. Here I am destructuring the results into data fetching indicators, like isLoading, isFetching. They are very helpful if I want to show users some proper feedback on what is currently happening after each action. For example, when data from the backend is loading, then using isLoading:

```
if (isLoading || isFetching) return 'loading'}
```

Similar logic is followed when creating new jobs, punching in and out, and deleting the jobs.

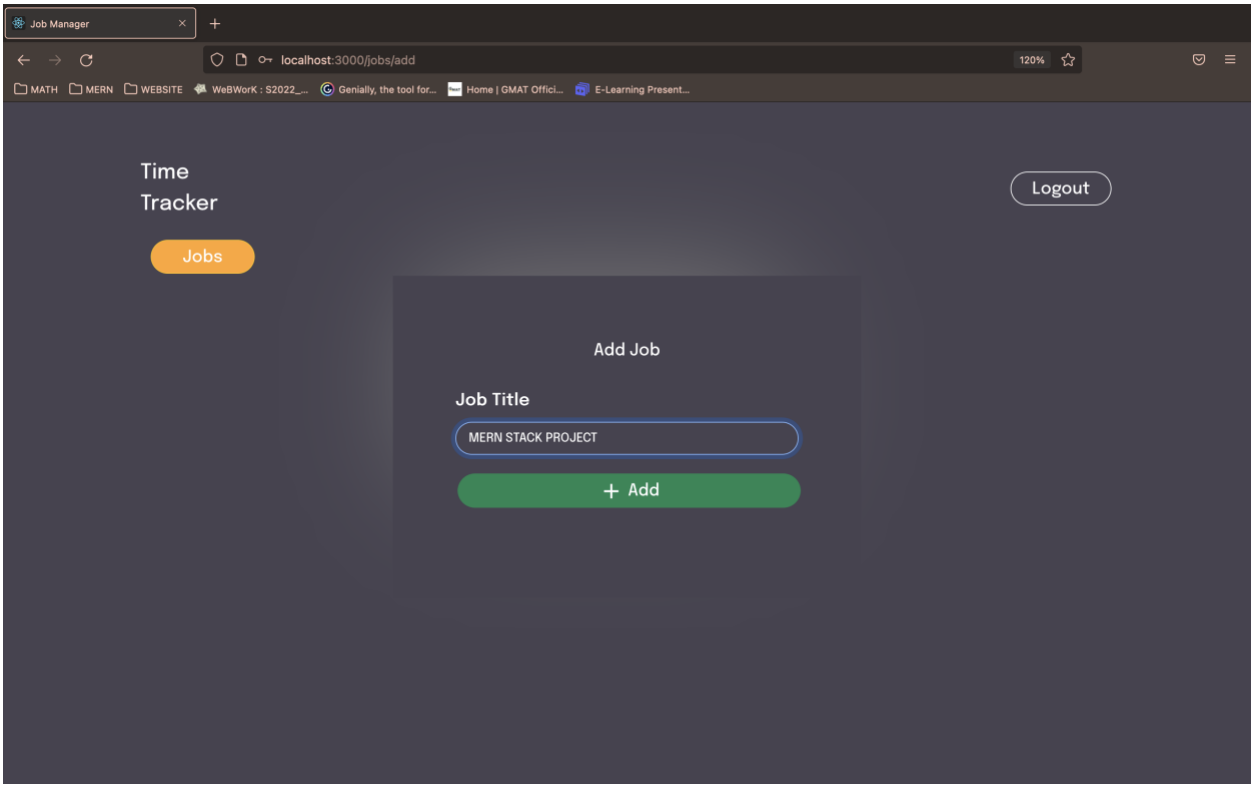
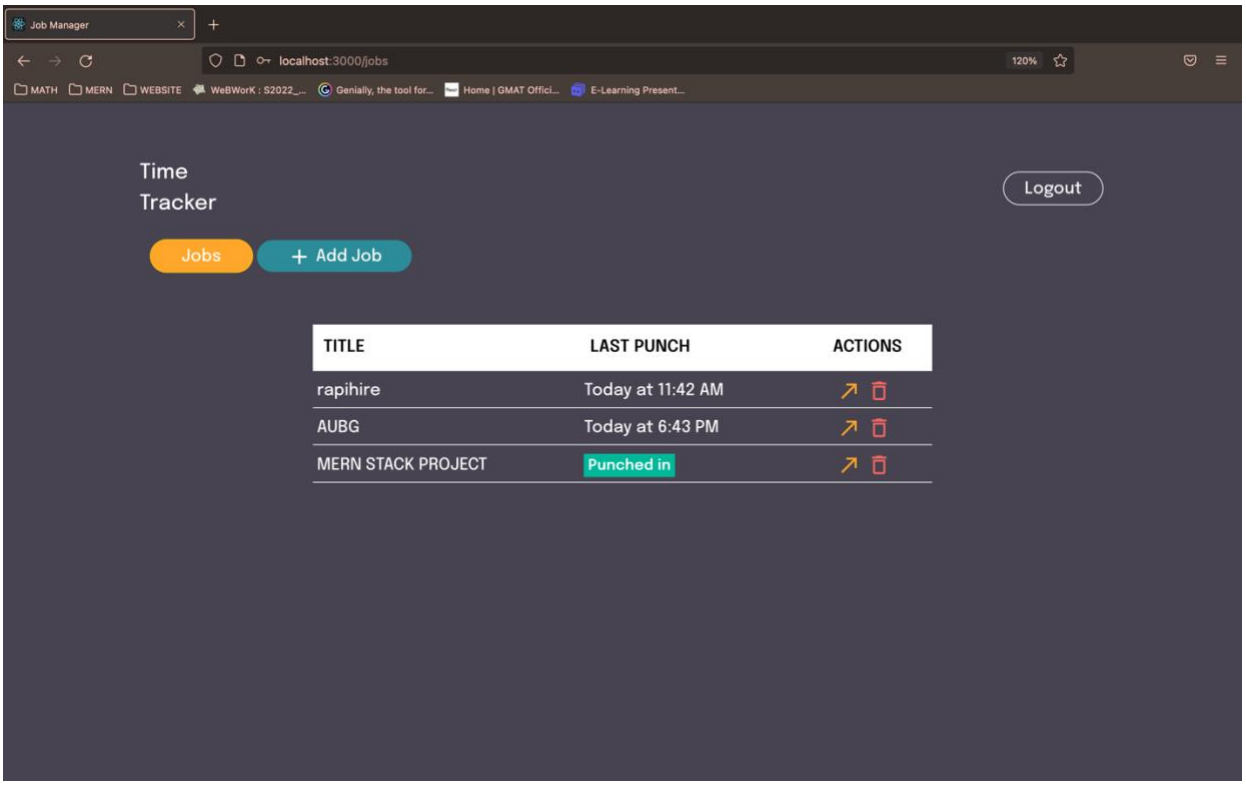


Figure 5 Add Job page

Figure 4 Total hours is calculated as the sum of all hours.



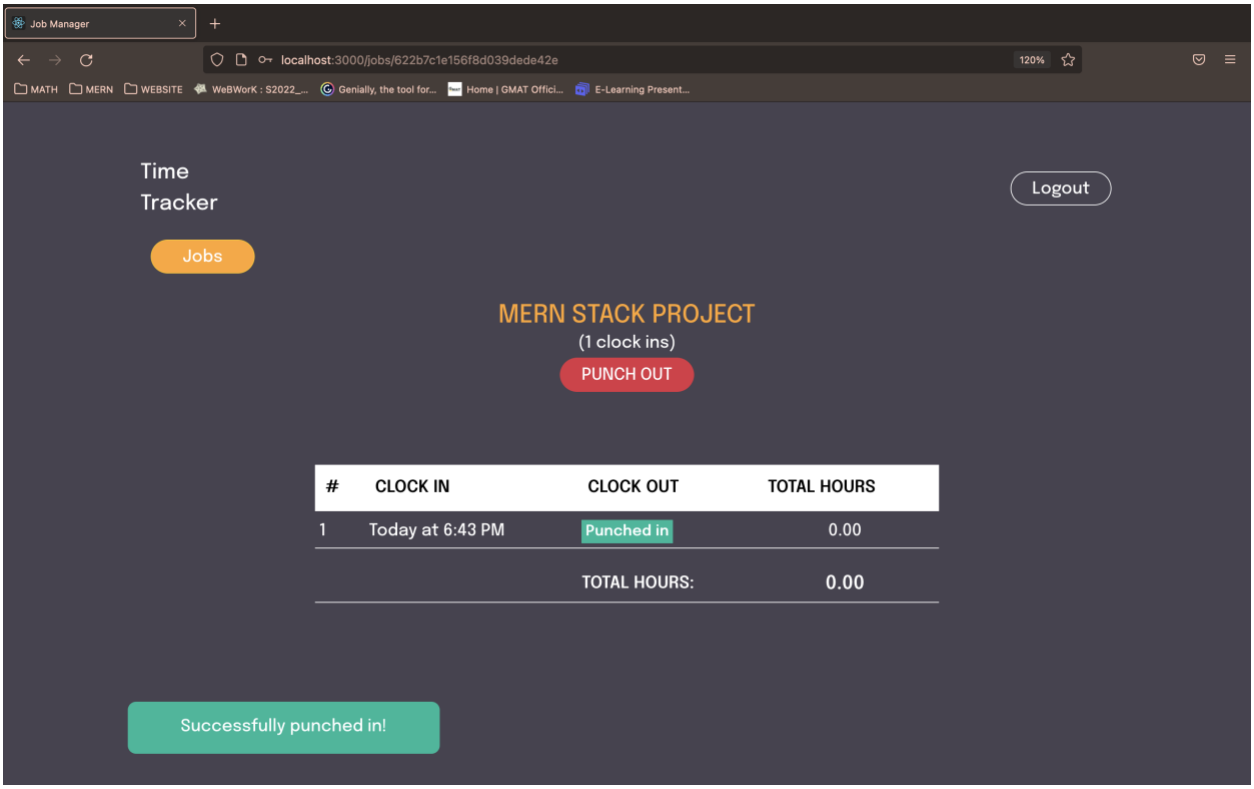


Figure 6 After the job is added we can click on the small arrow shown on the previous picture and afterwards clocking in. To punch out, it is enough to click "punch out"

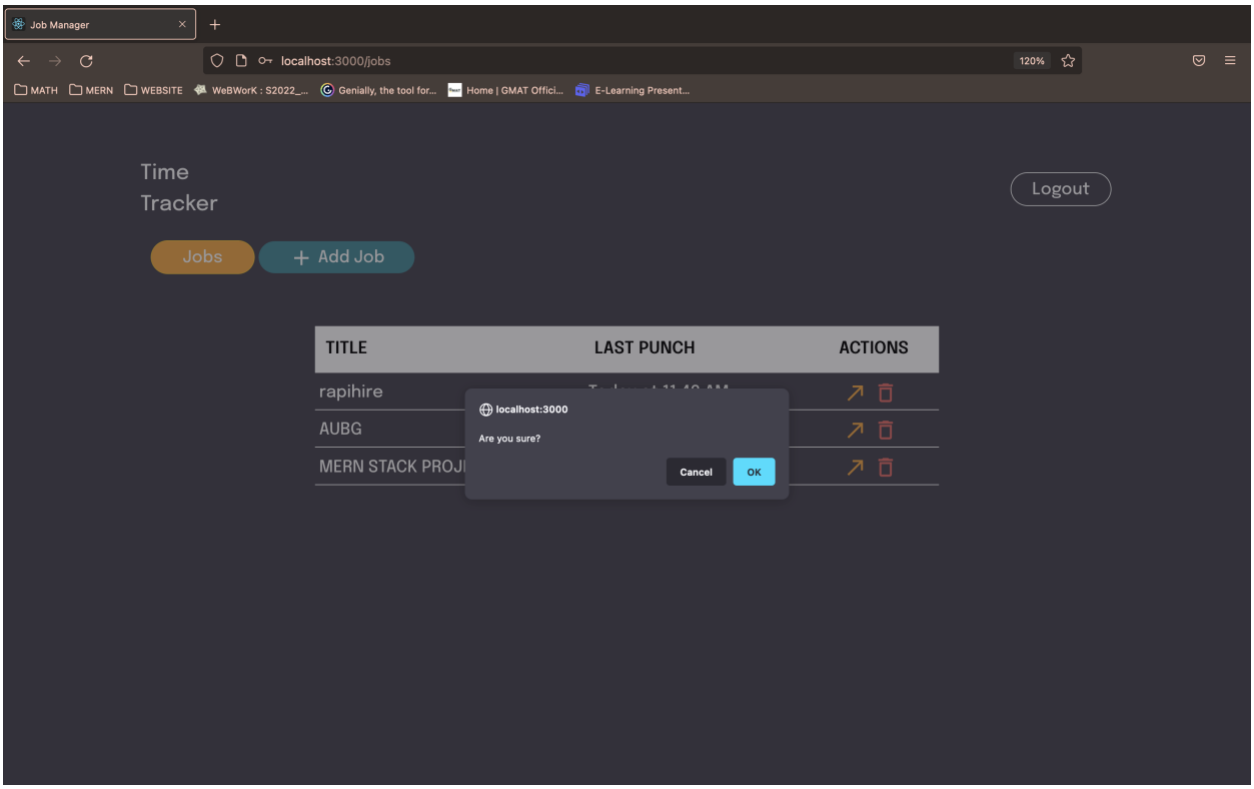


Figure 7 Delete a job