

IWT PROJECT

VIDEO COURSE PLATFORM

DECEMBER 15

PRESENTED BY:

PIYUSH MISHRA (*B120041*)

BYOMOKESH SENAPATI (*B120017*)



TABLE OF CONTENTS

ACKNOWLEDGEMENT	3
ABOUT THE PROJECT	4
EZ-Courses	4
DIRECTORY STRUCTURE	5
SYSTEM DESIGN	6
DATABASE MODEL AND SCHEMA	7
LOGIN FLOW	8
BACKEND ROUTES	11
USER INTERFACE	14
CONCLUSION	16

ACKNOWLEDGEMENT

We would like to acknowledge and give our warmest thanks to our supervisor Dr. Sanjay Saxena, who made this work possible. His guidance helped us through all the stages of making this project. We would also like to thank everyone that directly or indirectly helped us or influenced us in any way in making and completing this project.

We would like to express our gratitude towards the members of our college for their kind cooperation and encouragement which helped us in completion of this project. All of them have willingly helped us out with their abilities.

ABOUT THE PROJECT

EZ-Courses

‘EZ-Courses’ is an online learning and teaching marketplace. It allows instructors to build online courses on their preferred topics. Its learning experience arranges coursework into a series of modules and lessons that includes videos in playlist format.

The inspiration behind the project was to create an e-learning platform that is seemingly easy to use and with the completion of this project, we would be well versed with the system design and working of an e-learning platform and will be introduced to all the errors and obstacles that one would have faced while building such type of project.

This quote is quite literally what our mindsets have been for this project:

***“I hear and I forget. I see and I remember.
I do and I understand.”
- Confucius***

DIRECTORY STRUCTURE

ez-courses/

- ├─ public/ Contains all the compiled files and assets like logos, html files, manifest files, etc.
 - | ├─ index.html
 - | ├─ manifest.json
- ├─ server/ Contains all the necessary packages and files for running the server and the environment variables
 - | ├─ middlewares/
 - | ├─ routes/
 - | ├─ utils/
 - | ├─ index.js
 - | ├─ package-lock.json
 - | ├─ package.json
 - | ├─ database.sql
 - | ├─ .env
 - | ├─ package.json
- ├─ src/ Contains all the files for running the client side/frontend part of the application
 - | ├─ api/
 - | ├─ components/
 - | ├─ pages/
 - | ├─ App.jsx
 - | ├─ index.js
- ├─ .gitignore
- ├─ package-lock.json
- ├─ package.json
- └─ README.md Contains the details for running the project

SYSTEM DESIGN

For designing and making this full-stack project, we used the following technologies:

1. **PostgreSQL:** This is a free and open-source relational database management system emphasizing extensibility and SQL compliance.
2. **Express:** This is a web application framework for Node.js that will be used to build the backend of the project.
3. **React:** This is a JavaScript library for building user interfaces that will be used to build the frontend of the project.
4. **Node.js:** This is a JavaScript runtime that will be used to execute the backend code.

The system will be designed with a client-server architecture, where the frontend React components will make requests to the backend Express server with the help of a library 'axios', which will then retrieve and manipulate data from the Postgres database as we have described in the backend routes as needed. The server will also handle authentication and authorization by protected routes for the application, ensuring that only authorized users have access to certain features and data.

Overall, this stack provides a powerful and flexible technology stack for building full-stack web applications. With its combination of a popular SQL and scalable database with tons of in-built functionalities, web application framework, and frontend library, this stack enables developers to quickly and easily build robust and scalable web applications and guarantees high availability of service for clients.

DATABASE MODEL AND SCHEMA

We create two tables: users and courses, along with the extension of 'uuid-oss' which generated random hex like string for unique ID's.

```
CREATE EXTENSION IF NOT EXISTS "uuid-oss";
```

```
CREATE TABLE users(  
  user_id uuid PRIMARY KEY DEFAULT uuid_generate_v4(),  
  user_name TEXT NOT NULL,  
  user_email TEXT NOT NULL UNIQUE,  
  user_password TEXT NOT NULL,  
  courses TEXT[],  
);
```

```
CREATE TABLE courses(  
  course_id uuid PRIMARY KEY DEFAULT uuid_generate_v4(),  
  course_name TEXT NOT NULL,  
  course_price INT NOT NULL,  
  enrolled_members INT DEFAULT 0,  
  course_instructor TEXT NOT NULL,  
  course_link TEXT NOT NULL  
);
```

In the users table, the courses column is an array containing the course_id of each course the particular user has subscribed to. Basically, it's an array of foreign key's which is the course_id column of the courses table for the users table.

In our case, we have hosted the database on Railway which provides an easy and scalable solution for remote databases and it is easy for getting started.

LOGIN FLOW

The whole authentication and authorization architecture for this project is based on tokenized authentication with the help of JWT (JSON Web Tokens) which is a proposed Internet standard for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims. The tokens are signed either using a private secret or a public/private key.

Here, we are encoding the details of the user in the JWT token we are creating, such as:

- Login state – Boolean value
- User name
- User ID – random generated string by 'uuid' (JS library)
- User email

The token is then signed with a private key which we have previously generated or by a random string which we can paste in the environment variable file (.env) in the root directory of the project. We can generate a random string for this purpose by any method, one of which is:

```
var key = require('crypto').randomBytes(64).toString('hex');  
console.log(key) //secret-key
```

The environment variable file should be placed in the server folder and should contain four basic variables, which are:

- **ACCESS_TOKEN_SECRET** – Random generated string
- **REFRESH_TOKEN_SECRET** – Random generated string
- **COOKIE_DOMAIN** – localhost
- **DATABASE_URL** – string of the hosted Postgres DB(in this case, on Railway.app) or leave blank if using local DB

The JWT tokens are made using the following snippet and two tokens are set, one for access token (expires in short period of time for authenticating purposes) and refresh token (expires in longer period of time for remembering the logged in state of the user in the browser):

```
import jwt from "jwt";  
const user = { user_id, user_name, user_email, courses};  
const accessToken = jwt.sign(user, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '15m' });  
const refreshToken = jwt.sign(user, process.env.REFRESH_TOKEN_SECRET, { expiresIn: '15d' });
```

In the frontend part, the data for the auth states and details of the user are fetched using the library of 'axios' and using the data is passed on to the various child components through the native React Context API. The data is fetched on page load using the 'useEffect' hook. The data is then stored in an empty object created using the 'useState' hook of React which initializes a null object and then upon successfully fetching details from the backend route, stores it in the object populating it.

Next, we create an instance of a Context Provider which basically passes on the state to the child components and then wrap all the child components created in the app with this Context Provider with the value to pass being the auth value which in this case is the object which we populated.

```
const AuthProvider = ({ children }) => {
  const [auth, setAuth] = useState(null);
  useEffect(() => {
    const checkAuth = async () => {
      await axios({
        method: "get",
        url: "http://localhost:5000/api/auth",
        withCredentials: true,
        headers: { authorization: token },
      })
      .then((props) => {
        // console.log(props.data);
        setAuth(props.data);
      })
      .catch((err) => {
        console.log(err);
        if (err.response.status === 403) {
          setAuth(false);
        }
      });
    };
    // console.log(auth);
    checkAuth();
  }, [auth]);

  return <AuthContext.Provider value={auth}>{children}</AuthContext.Provider>;
};
```

BACKEND ROUTES

The backend handles the behind-the-scenes functionality, such as storing and retrieving data from a database, performing calculations, and handling server-side logic. Its responsible for providing an API (Application Programming Interface) that the frontend can communicate with. This allows the frontend to send requests to the backend, such as to retrieve or store data, and the backend can respond with the requested data or a status message indicating the success or failure of the operation. We are communicating with the PostgreSQL database with the help of the postgres wrapper library 'pg' which is a Postgres library for Express and NodeJS.

```
import pg from 'pg';
const poolConfig = {
  connectionString: process.env.DATABASE_URL,
};
const pool = new Pool(poolConfig);
```

The backend is also handling other tasks such as authentication, authorization, and security. Various routes are setup for providing all these functionalities. Some important routes are:

- **POST ('/api/users/'): Register Route:**
 - A user can register through this route where user provides email, name and password and then the password is hashed using the library 'bcrypt' and then is stored in the database.

```
import bcrypt from "bcrypt";
const hashedPassword = await bcrypt.hash(req.body.password, 10);
```

- **POST ('/api/auth/login'):** Login Route

- The user can login through this route where he/she provides email and password which is then compared to the hashed stored password in the db and then redirected if successful. A new JWT token is also generated and the '/api/auth/refresh_token' route is automatically called after this.

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

```
res.cookie('refresh_token', tokens.refreshToken, { ...(process.env.COOKIE_DOMAIN && {  
domain: process.env.COOKIE_DOMAIN }), httpOnly: true, sameSite: 'none', secure: true });
```

```
res.cookie('auth_token', tokens.refreshToken, { ...(process.env.COOKIE_DOMAIN && { domain:  
process.env.COOKIE_DOMAIN }), maxAge: 1296 * Math.pow(10, 6), httpOnly: false, sameSite:  
'none', secure: true }).header('auth_token', tokens.refreshToken);
```

- **GET ('/api/auth/'):** Auth Route

- The auth state is checked through this route which checks the cookie sent in the request header and returns a Boolean value if the cookie is valid or not along with the details of the user as an object if the cookie is valid.

- **GET ('/api/courses/'):** Get Courses Route

- This route just queries the 'courses' table of the database and returns all the courses registered in the table.

- **POST ('/api/courses/'):** Upload Course Route

- This route is used for uploading a course to the 'courses' table in the database where we run a simple insert sql query. The course name, course price, course instructor and course link are passed as parameters. The course link is just a YouTube playlist link for now, which is parsed to get the playlist ID and then we get the contents using the YouTube Data API (official Google API) in the frontend side using axios.

```
await pool.query(  
  'INSERT INTO courses (course_name,course_price,course_instructor,course_link) VALUES  
  ($1,$2,$3,$4) RETURNING *'  
  , [req.body.course_name, req.body.course_price, req.body.course_instructor,  
  req.body.course_link]  
  );
```

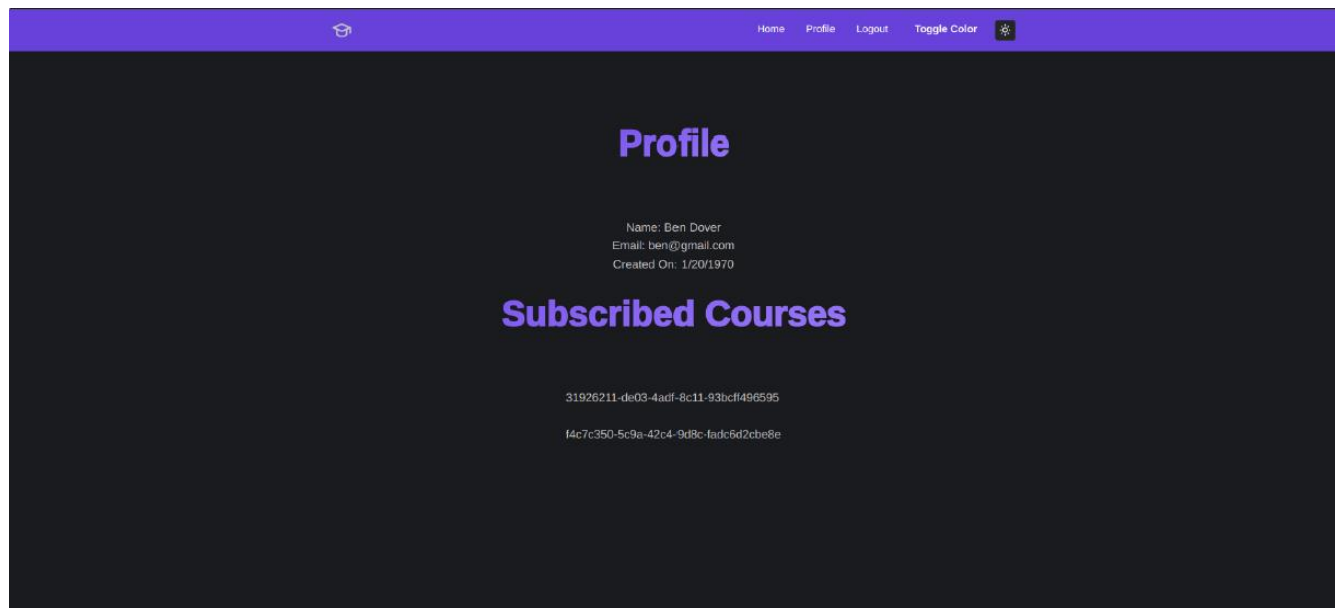
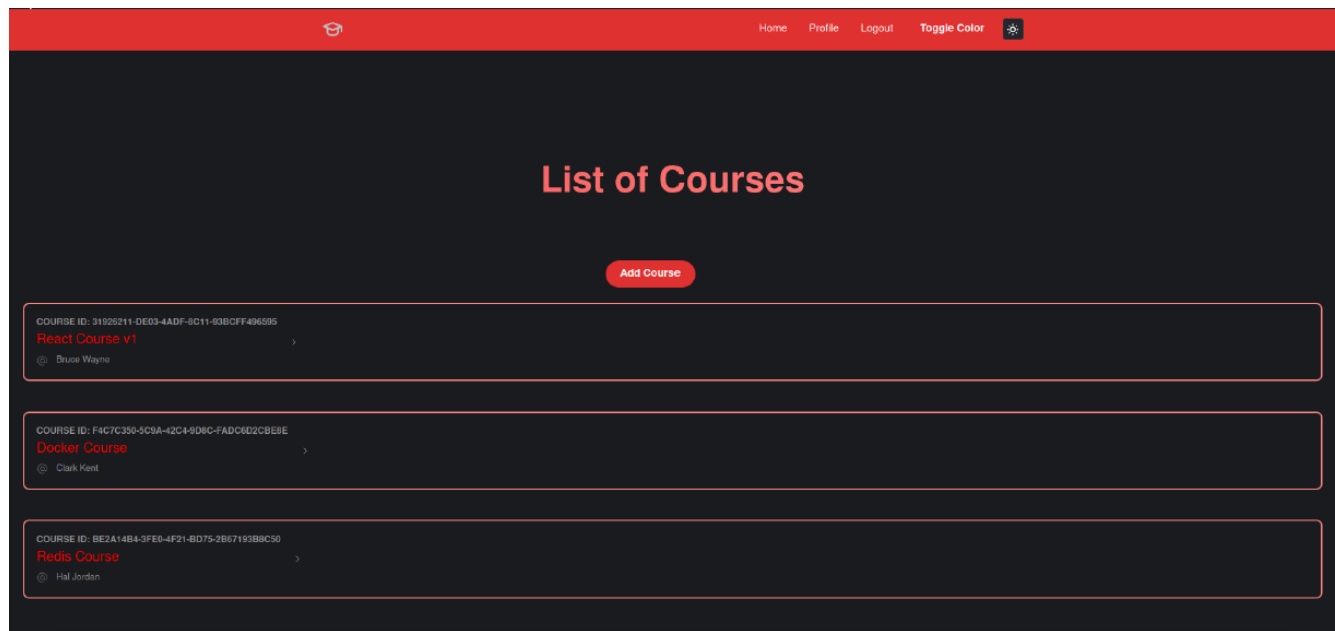
- **PUT ('/api/courses/add_course/')**: Course Subscribe Route
 - This route adds the course id to the array in the 'courses' column of the 'users' table for a particular user (the one currently logged in). We update the array with a new foreign key and also increment the counter of 'enrolled_members' column in the 'courses' table.



```
await pool.query(  
  'UPDATE users SET courses = ARRAY_APPEND(courses, $1) WHERE user_id = $2'  
  , [req.body.course_id, req.body.user_id]  
);  
await pool.query(  
  'UPDATE courses SET enrolled_members=enrolled_members+1 WHERE course_id = $1'  
  , [req.body.course_id]  
);
```


- **DELETE ('/api/auth/logout')**: Logout Route
 - This route is used for logging out the user by basically deleting the auth tokens set in the browser cookies which will then return an auth state of Boolean FALSE and prevent the user from browsing the site.

```
res.clearCookie('refresh_token');  
return res.status(200).clearCookie('auth_token').json({ message: 'Refresh token deleted.' });
```

USER INTERFACE



 [Home](#) [Profile](#) [Logout](#) [Toggle Color](#) 


 Intro to Class Components | Class Components in React tutorial

Watch later Share Info

1. Class Components Section intro


MORE VIDEOS


0:00 / 3:34 • Intro




Course Page

Course ID: 31926211-de03-4adf-8c11-93bcff496595

 Intro to Class Components | Class Components in React tutorial >

 ES6 (JavaScript) classes review | Class Components in React tutorial >

 Pass props to Class Components | Class Components in React tutorial >


Welcome back!

Do not have an account yet? [Create account](#)

Email *

you@mantine.dev

Password *

Your password 

Sign in

CONCLUSION

In conclusion, this Full-stack E-Learning app, 'EZ-Courses' is a valuable tool that hopes to achieve its goal of providing users with an accessible and engaging platform for learning. The app's key features, such as the ability to create and manage courses, enroll students, and track enrolled students, live streaming with history and profile page tracking subscribed courses, have been thought through and implemented in a way that's easy to understand for a client and hopes to improve the UX of the website.

During the course of the project, we faced several challenges, such as integrating the frontend and backend, and implementing the authorization flow and tracking the user's subscribed courses and fetching multiple foreign keys with a single query instead of compound and nested queries. However, through collaboration and hard work, we were able to overcome these challenges and deliver a high-quality product.

In the future, we plan to continue improving the app by adding additional features, such as support for live/real-time updates and progress-tracking, and by refining the user interface to make it even more user-friendly and improving the UX in overall. We also plan to explore new technologies and platforms to expand the app's reach and provide even more value to users. Overall, we are proud of EZ-Courses and thank you for your time in reading this report about the various aspects of the project.

THANK YOU
