Adding Local Authentication to the Project Starter

CUNY Tech Prep

November 2021

Disclaimer:
Implementing Authentication is a critical part of any application. The steps we will go over here today are for learning purposes.

Vulnerabilities and threats evolve daily and a best practice of today may not be proper tomorrow. When implementing security aspects of our applications we have to stay up-to-date with known vulnerabilities and evolving best practices.

For applications with real-world users it is recommended that you use an authentication service like Autho or a well vetted solution.

Auth0 Tutorial for Express.js: https://auth0.com/blog/create-a-simple-and-secure-node-express-app/
OR Auth0 in React: https://auth0.com/docs/quickstart/spa/react

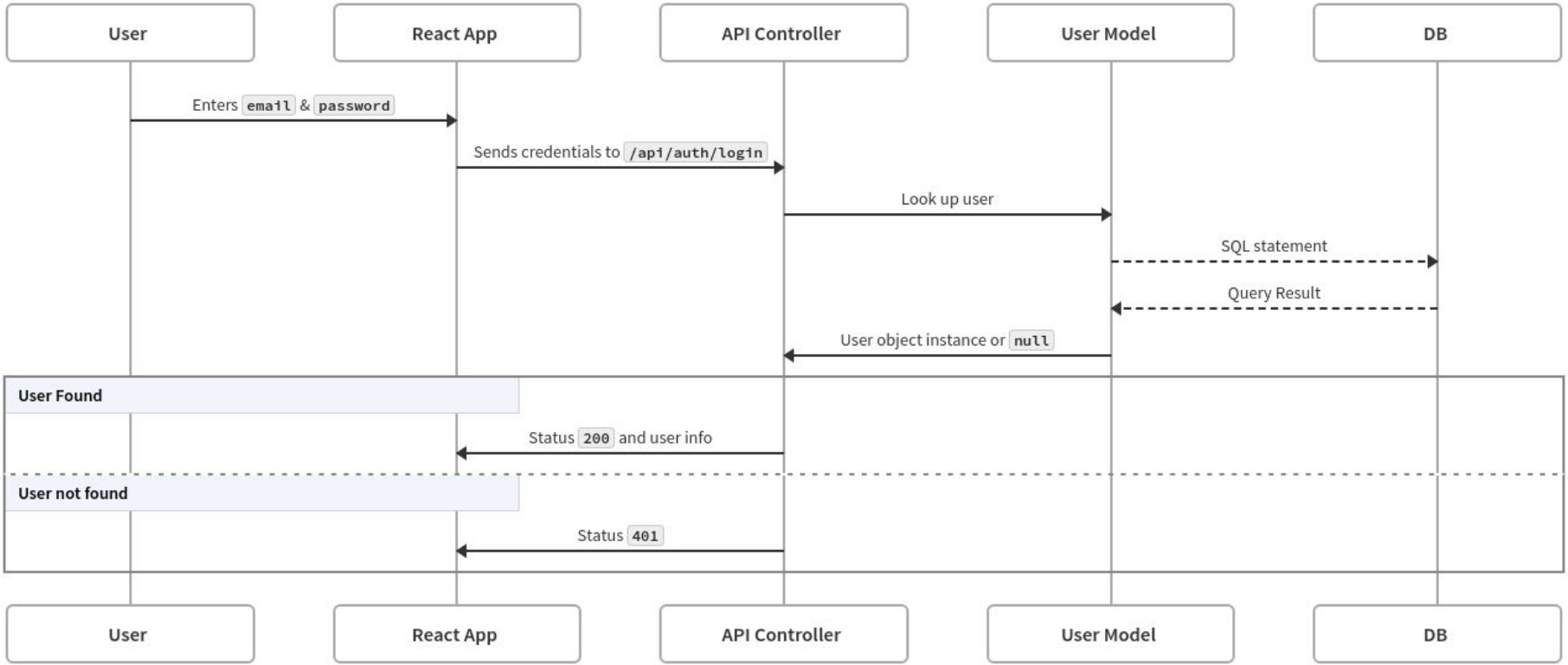
What are we implementing: **Authentication using email & password**

Two Major Components:

- Backend
  - Store user info
  - Securely store passwords
  - Verifies passwords
  - Manage user sessions using cookies
  - Protect api routes to authenticated users only
- Frontend
  - Renders Login/Signup forms
  - Holds user login state
  - Protects pages/routes to authenticated users only

Demo

Login and Sessions



Let’s build it out

Backend Components

Backend: install dependencies

Use well vetted and updated packages to handle authentication

- bcryptjs (about bcrypt algorithm)
- express-session
- passport
- passport-local

```
$ npm install bcryptjs express-session passport passport-local
```

Backend: add User model

Create a User model to store our **email** and **password** credentials

Ensure that we store the **hashed password** and NOT the plaintext password

Use bcryptjs to perform password hashing

Backend: add authentication middleware

We will add the passport LocalStrategy to our project

The LocalStrategy is responsible for handling logins

Passport takes care of serializing our user identifier through:

- `passport.serializeUser(...)`
- `passport.deserializeUser(...)`

Learn more about data serialization

Backend: init session & authentication middleware

Add express-session to handle session cookies

Initialize express-session and passport as express middleware in app.js

express-session must be setup *before* passport

Do not expose your session secret (*use .env file*)

Backend: add auth controller

Add an authentication controller for signup, login, and logout API endpoints

- `POST /api/auth/signup`
- `POST /api/auth/login`
- `POST /api/auth/logout`

Mount the controller and test using insomnia app

Backend: protect api endpoints

Create a middleware function to protect endpoints

Only authenticated users should be allowed to create, update, and delete a post

- `POST   /api/posts/`
- `PUT    /api/posts/:id`
- `DELETE /api/posts/:id`

All users can view posts.

- `GET /api/posts/`
- `GET /api/posts/:id`

Test with insomnia

Frontend Components

NOTE: these instructions currently use React Router v5

Frontend: add and use AuthContext

Use react context to globally hold our authentication state

AuthContext will hold:

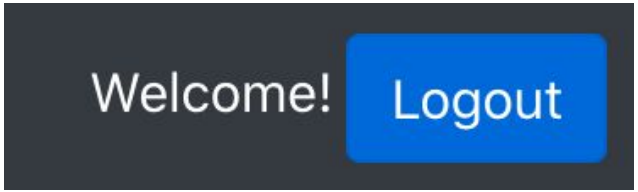
- `authenticate()` function for login
- `signout()` function for logout
- `user` object with current user info
- `isAuthenticated` boolean to know if user is logged in or not

Source: https://v5.reactrouter.com/web/example/auth-workflow - https://ui.dev/react-router-v5-protected-routes-authentication

Frontend: add AuthButton component

Component will switch between a Login and Logout button depending on users authentication state

Will use AuthContext to check authentication state



withRouter documentation: https://v5.reactrouter.com/web/api/withRouter

Frontend: add LoginPage

Add a LoginPage with login form to accept users email and password

If login is successful redirect user to previous page or home (`/`)

Add `/login` route to the frontend

Source:

Frontend: add PrivateRoute component

Protect frontend routes for authorized users only

Create a PrivateRoute component to encapsulate logic

Apply PrivateRoute to the Create a micropost page (`/posts/new`)

Frontend: check authentication state on refresh

React state is reset with each browser refresh

We lose authentication state, although the session cookie is still valid

Add a check on app startup to determine if the user is already logged in

Use the useEffect hook

We now have authentication working on the backend and frontend

Let’s dive deeper into the concepts of authentication

Concept:

Vulnerable and Outdated libraries

Source: https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/

Project Dependencies (open source)

The Good

- Don’t reinvent the wheel
- Well vetted by the community
- Vulnerabilities are reported and corrected in the open

The Bad

- Unmaintained projects
- Malicious actors can take over the project
- Typosquatting

These apply to all programming languages and frameworks

```
$ npm install bcryptjs express-session passport passport-local
```

Concept:

Secure Password Storage

Source: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Hashing vs Encrypting Passwords

✅ Hashing

Hashing is a one-way function

Impossible to “decrypt” (or get back to) the original value

❌ Encrypting

Encryption is a two-way function

We can “decrypt” (or get back) the original value that was encrypted

Encryption

Use encryption for protecting data at rest

Encrypt data our app and user needs access to, but is otherwise sensitive

```
$ echo "BankAcct: 00123" | openssl enc -aes-256-cbc -base64
<ENCRYPTED_VALUE>

$ echo “<ENCRYPTED_VALUE>” | openssl enc -aes-256-cbc -base64 -d
BankAcct: 00123
```

Source: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

Hashing

Use for secrets only the user should know

The application only needs to verify that the secret is correct,
but cannot tell what the secret is

There is no way to reverse the hashing

```
$ echo -n 'iLoveCTP' | openssl sha1
6f83eb40e42457b11e559ea5e377c634ab0337b1
```

Password Hashing Algorithms

✅ Strong

- Argon2
- bcrypt
- scrypt
- PBKDF2

❌ Weak (never use these)

- md5
- sha1 (*git commits*)
- sha256

Source: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#password-hashing-algorithms

bcrypt

- Strong hashing function
  - Hard to come up with collisions
- Slow to compute
  - adjustable work factor
- Passwords are uniquely salted
- Algorithm is well tested by the community at large
- Which package do we install?

Users often reuse passwords across multiple sites.

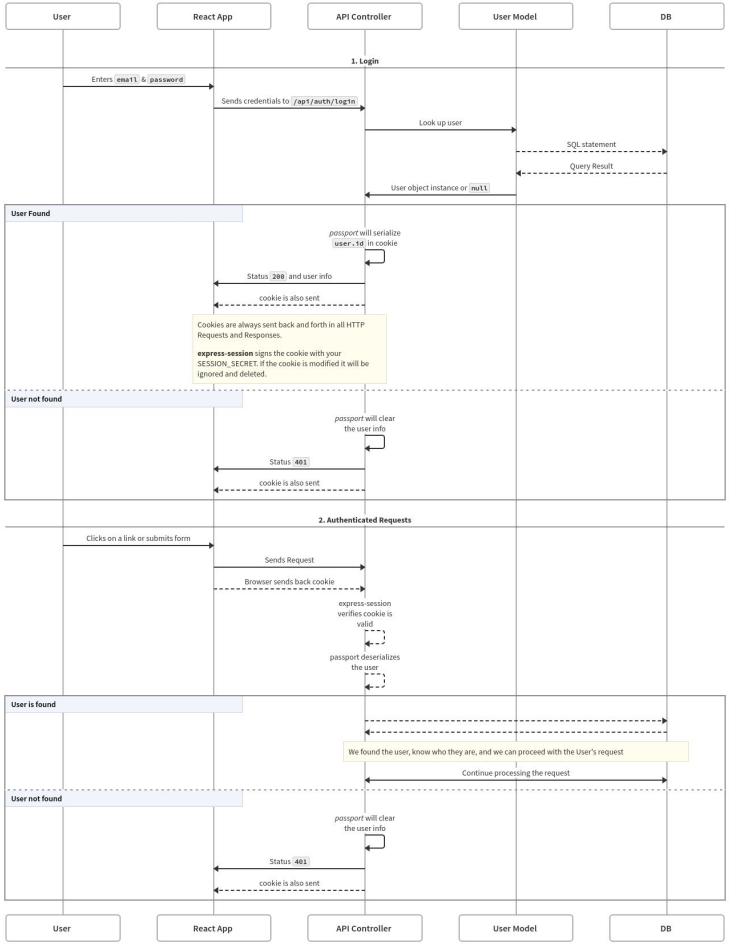
With proper hashing using unique salts we can make a best effort at protecting these users.

- Learn more about how attacks are carried out against leaked password databases

Concept:

Session cookies

Login and Sessions



Session Cookies

What is it:

- A cookie with a user identifier
- The cookie is signed in such a way that if it's tampered, attacker cannot create a corresponding signature
- API backend trusts the cookie if untampered

Concerns:

- Cookie is tampered with
- If SESSION_SECRET is leaked, attackers can create valid session cookies
- User's browser or environment gives a third party access to the session cookie

Concept:

Sensitive Data Exposure

Leaking/Logging Sensitive Information

Sensitive Information

- User PII (Personally Identifying Information)
- bcrypt hashes (*we just did this* 😵)
- Complete error messages with stack traces or debug information

Remedies

- Don't store sensitive data that is not needed (delete, forget, or never ask)
- Encrypt all sensitive data at rest
- Return only necessary information to client
- Don't log unnecessarily (console.log, printf, etc…) (Facebook 2019)

Frontend Concerns

- Protect cookies, tokens, and JWT
- Protect against XSS and CSRF
- Browser can be exploited
- Protect pages on the frontend & backend

MUST USE SSL/HTTPS

Otherwise everything we've done is useless

That's all for now 👋

If you find any issues with the presentation or code we'd appreciate your feedback here: https://github.com/CUNYTechPrep/project-starter/issues/new