Monorepo and Hooks

CSC309

Kianoosh Abbasi

UNIVERSITY OF TORONTO

Fall 2024

This session

- Monorepo: React in Next.js
- Enhanced function components
  - Hooks
- API calls

React so far

- Enabled by importing some scripts to our HTML file
- JSX code must be translated to JS every time
- Very slow

React in Node.js projects

- Serves front-end code from a Node server
- When browser requests a URL, a series of HTML, CSS, and JavaScript files are returned
  - Containing compiled JavaScript components
- A pre-compiled and bundled build for production

React and Next.js

- Good news: Next.js already supports React!
- But wait! Isn't Next.js a backend framework?
- Answer: It's both frontend and backend!
- How is it possible? Is it a good thing?

Monorepo

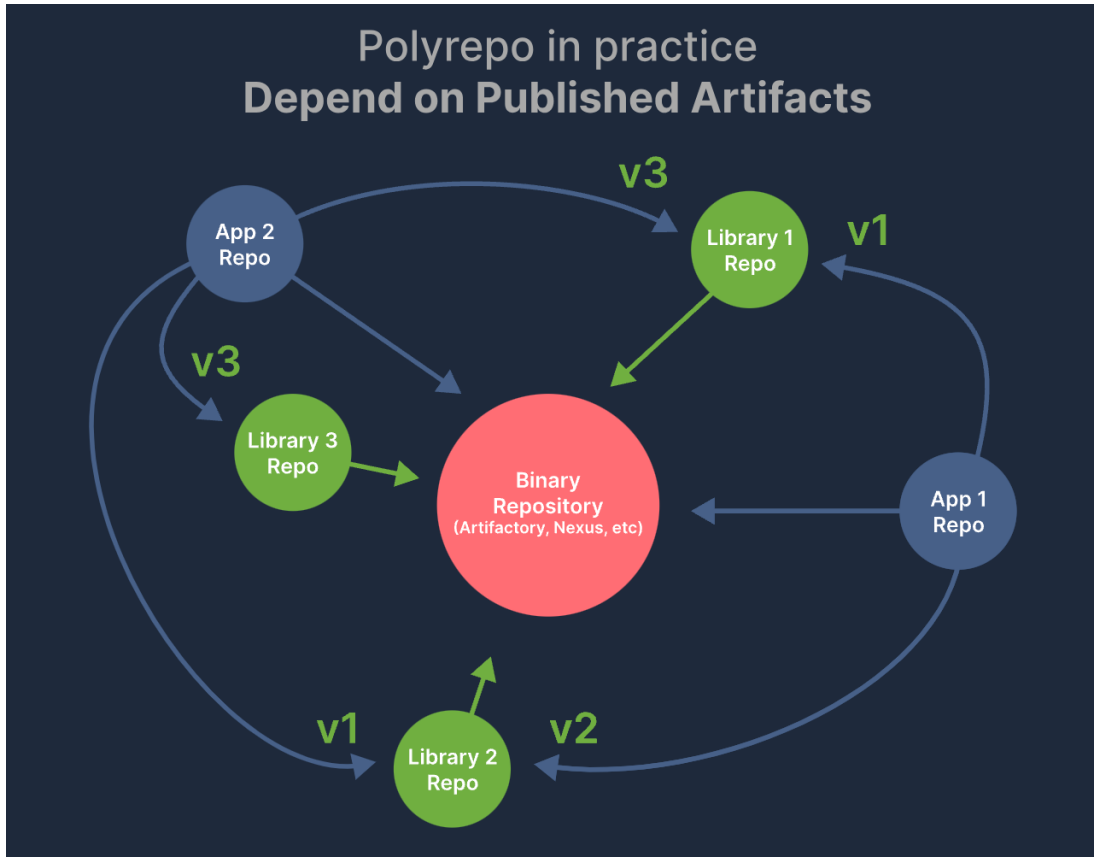
Visit https://monorepo.tools/

- The practice of having all your code in one repository
  - Backend, web frontend, mobile frontend, libraries, etc.

- Giant codebases like Google, Facebook, and Microsoft follow this practice

Monorepo benefits

- Does not deal with repo versions anymore
- Share types, utils, libraries
- Project is self-contained and easy to navigate



- Next.js is one step beyond monorepo, it's a monolith!
  - Monorepo: different Node projects (apps and shared packages) in one parent Node project:
  - Monolith: All the code in one Node project!

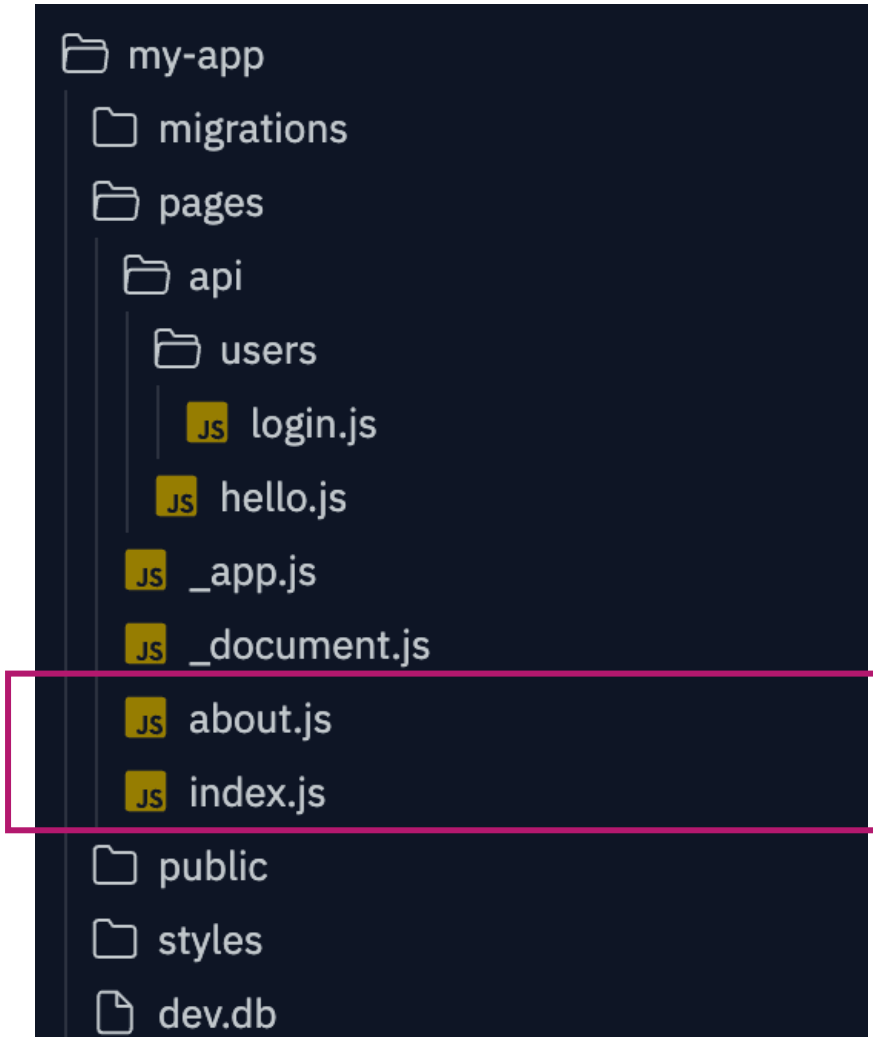
- Monolith is a good choice for small projects
  - Might need to migrate to monorepo if project gets bigger
  - Extract reused parts into separate Node projects

- In this course, we're happy with monolith of course!

React in Next.js

- Within the pages directory, every path that's outside /api is a front-end path
  - Except if it starts with underscore

- Define a React component in each file and make it the default export
  - Will be rendered when the path is accessed from browser



React in Next.js

- Creates the HTML and compiles the JSX for you!
- Cherry on the cake: You can import all your styles and assets (image, font, etc.) to your JS modules
  - Handled and served properly by the server
- The only programming language needed for a web app is JavaScript!

File structure

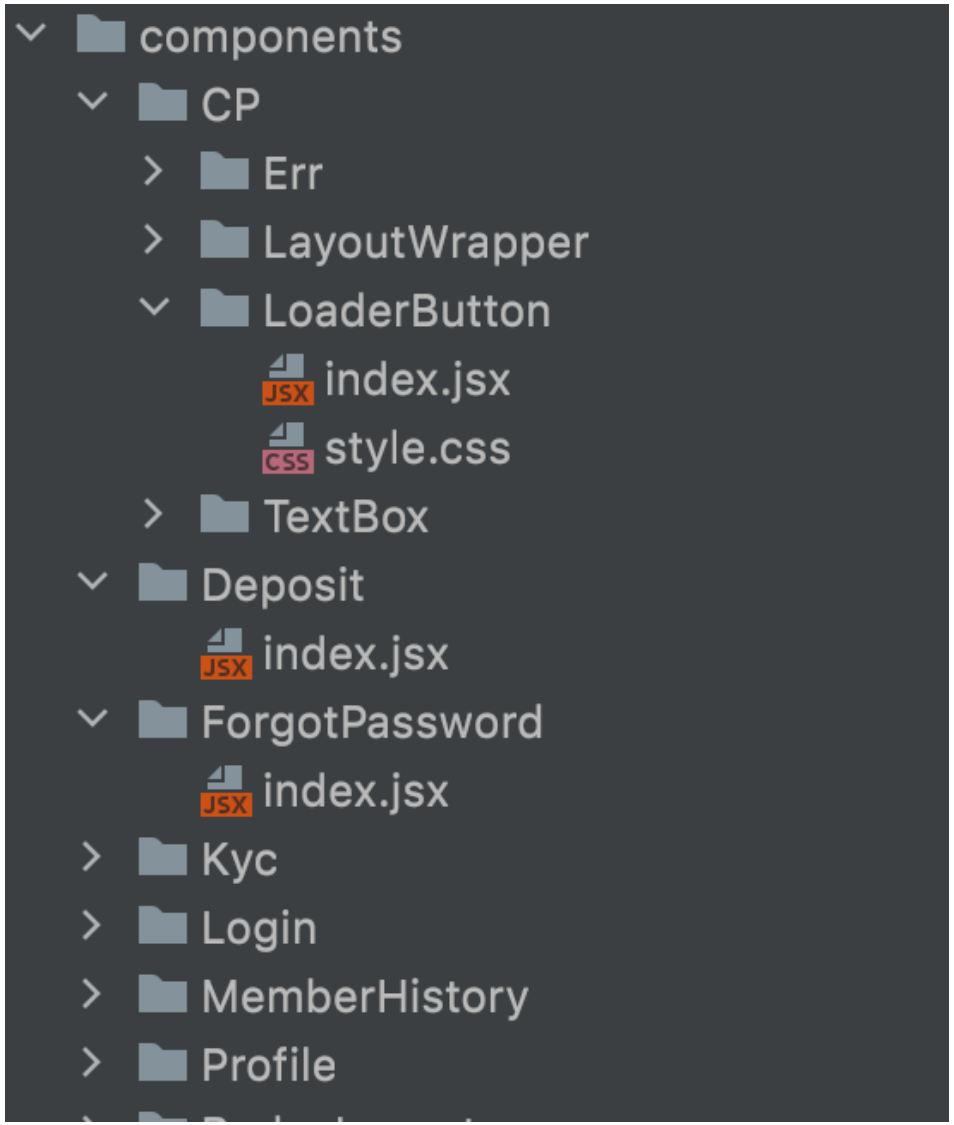
- It's a good practice to separate pages from components
  - You might not want your new files to be associated with paths

- Create a components directory
  - Might want to create a subdirectory for each components as well
  - Contains the main component (index.js) and child components

- Always have some re-usable base components
  - Inputs, forms, headings, etc.

File structure

- Import css files:
  ```
  import "@/styles/globals.css"
  ```
- Images and other static files can gather under the public directory
- Don’t make components too big:
  - Have nested, child components



Hooks

Hooks

- Great syntax sugars introduced in React 16.8
- No need to write verbose classes, constructors, and `setState` anymore
- You can move back to function components

useState

- State does not have to be one object anymore
- Define separate state variables via the `useState` hook

```
import React, { useState } from 'react';
```

- Returns the variable and update function
- Component gets re-rendered when the value changes

Example

```
const Status = (props) => {
    const [status, setStatus] = useState( initialState: "good");

    const toggleStatus = () => {
        setStatus( value: status === "good" ? "bad" : "good")
    }

    return (
        <>
            <h3>Situation is {status}</h3>
            <button onClick={toggleStatus}>toggle!</button>
        </>
    )
}
```

Benefits

Visit https://blog.bitsrc.io/6-reasons-to-use-react-hooks-instead-of-classes-7e3ee745fe04

- Function components instead of verbose class components
- Enables multiple state variables
- No more this, no more method binding
- Easy to share state with child elements
  - Each state variables comes with its own setter

Lifecycle

- So far, we only know to run code when render is called
  - In both class and function components
- You might not want to run code this way
  - Example: Sending a request upon load, accessing state values, etc.
- Adding lifecycle
  - In class components: `componentWillMount()`, `componentDidMount()`, `componentWillUnmount()`, etc.

useEffect

- A powerful hook to replace lifecycle functions
- Called when component mounts
- Also, can be called when something changes

- Import the hook

```
import React, { useState, useEffect } from 'react';
```

- Usage

```
useEffect(() => {
  console.log("This is called when component mounts")
}, [])
```

- Subscription
  - When any element of the array changes, the effect is invoked

```
useEffect(() => {
    console.log("props size or status has changed")
}, [status, props.length])
```

- Recommended to have a separate useEffect for different concerns

Benefits of hooks

```jsx
export function ShowCount(props) {
  const [count, setCount] = useState();

  useEffect(() => {
    setCount(props.count);
  }, [props.count]);

  return (
    <div>
      <h1> Count : {count} </h1>
    </div>
  );
}
```

```jsx
export class ShowCount extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  componentDidMount() {
    this.setState({
      count: this.props.count
    })
  }

  render() {
    return (
      <div>
        <h1> Count : {this.state.count} </h1>
      </div>
    );
  }

}
```

Benefits of hooks

```jsx
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```jsx
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

Notes

- Do not leave out the second argument
  - The effect would run at every re-render: inefficient

- The array should include all variables that are used in the effect
  - Otherwise, it might use stale values at re-renders

Exercise: a calculator with React

Fetch API

- The interface for browsers to send HTTP requests
  - Native support for REST framework
- Returns a promise
- Example:

```
let request = await fetch('/account/login/', {
  method: 'POST',
  data: {username: 'Kia', password: '123'}
})

const data = await response.json()
console.log(data);
```

API calls and hooks

- Example: fetching data on page load and adding it to state

```
const [holidays, setHolidays] = useState([]);

const fetchHolidays = async () => {
  const response = await fetch("https://canada-holidays.ca/api/v1/holidays");
  const data = await response.json();
  setHolidays(data.holidays);
}

useEffect(() => {
  fetchHolidays()
}, [])
```

Pagination

- Most times, GET APIs do not return all responses at once
  - Think of Google search results, Instagram posts, bank transactions
- Instead, they send results in pages

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "count": 10,
    "next": "http://localhost:8000/note/all?page=2",
    "previous": null,
    "results": [
        {
            "id": 10,
            "title": "Note 10",
            "content": "content 10",
            "last_udpated_on": "2021-01-09T01:44:33.645706Z",
            "is_active": true,
            "created": "2021-01-09T01:44:33.645745Z"
        },
        {
            "id": 9,
            "title": "Note 9",
            "content": "content 9",
            "last_udpated_on": "2021-01-09T01:44:29.487257Z",
            "is_active": true,
            "created": "2021-01-09T01:44:29.487295Z"
        }
    ]
}
```

API authentication

- First-party authentication
  - Store access/refresh token in client's persistent storage
  - Should not be deleted when tab/browser/computer is closed

- Web browsers: use `localStorage`

```
localStorage.setItem('access_token', access_token);
localStorage.getItem('access_token');
```

- Set `Authorization` header with appropriate value

API authentication

- Third-party authentication
  - Used when contacting external APIs
    - Maps, weather, payment, etc.

- Authentication is different
  - Our entire app is now a client to that third-party system
  - End-user cannot login to that system

- Solution: API keys
  - Either permanent (no expiration) or very long lived (months/year)

CORS

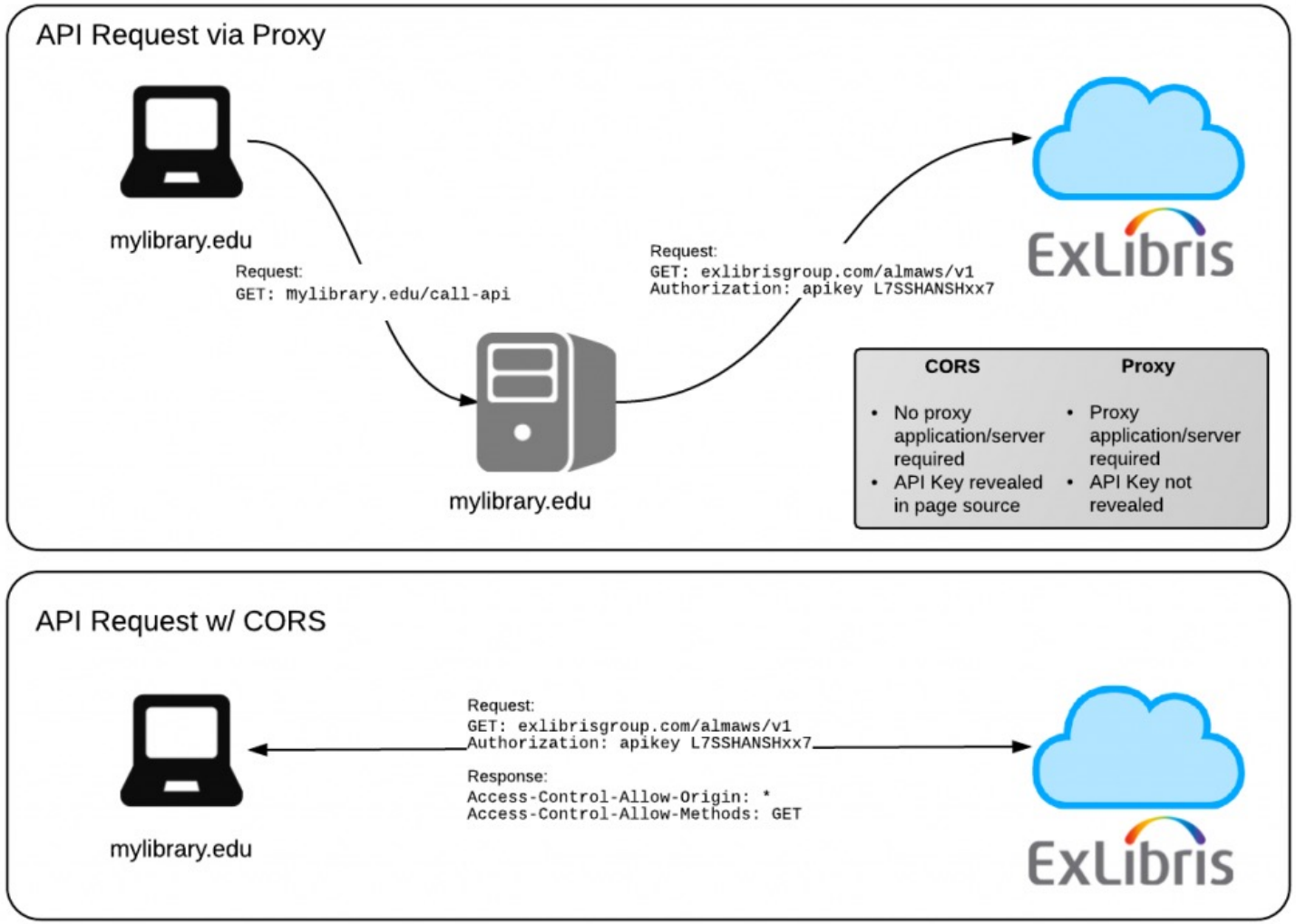
- Having front-end contacting third-party APIs is very dangerous
  - Client will have access to the API key
  - Significant security/financial issue

- Cross-Origin Resource Sharing (CORS)
  - A client should only request to URLS with the same domain
  - Browser block you from fetching a different domain
  - API servers also block requests coming from a browser

CORS

- Solution: Backend proxy
- Implement a backend API that requests the third-party service and returns the response
- Advantages:
  - API key is not exposed
  - More control over what data is transferred
  - More control over who accesses the data
  - Better logging and monitoring

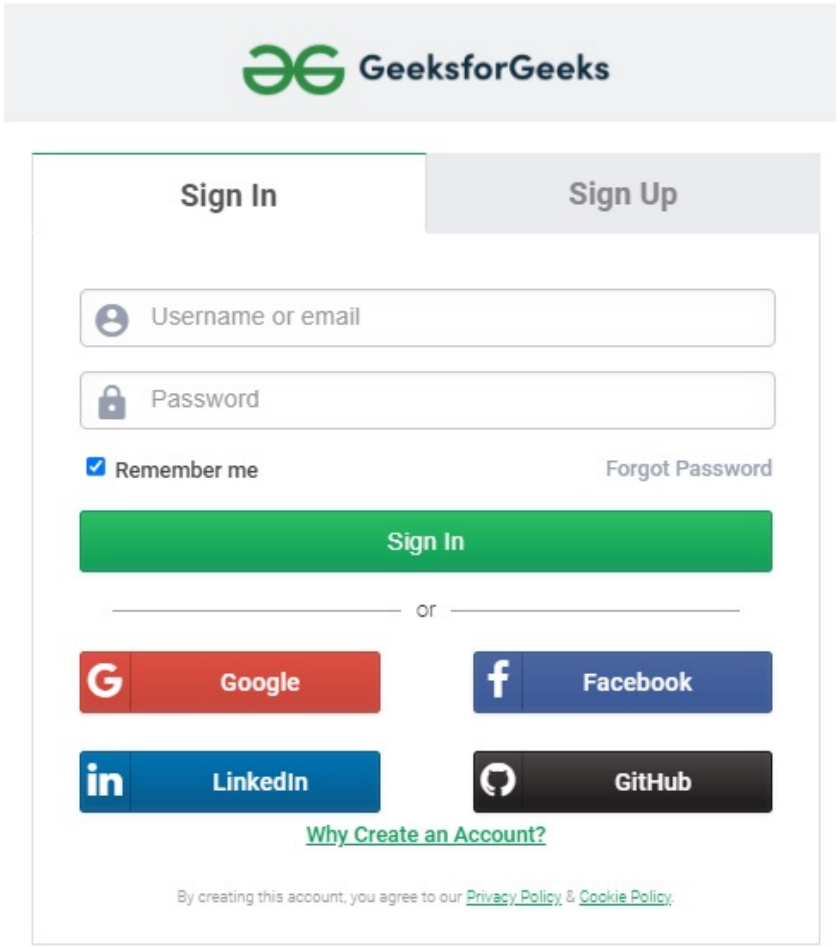
CORS



Source: https://developers.exlibrisgroup.com/blog/using-a-simple-proxy-to-add-cors-support-to-ex-libris-apis/

OAuth

- Sign in using Googler, Facebook, etc.
- Redirects client to Google sign in, if successful, redirects back with auth code
- Server contacts Google with auth code and API key to receive user info (name, email, files, …)
- Server creates an account and generate access token for client



Next session

- Navigation with Next.js
- Global state and context
- Type safety with TypeScript
- Advanced CSS
  - Tailwind classes