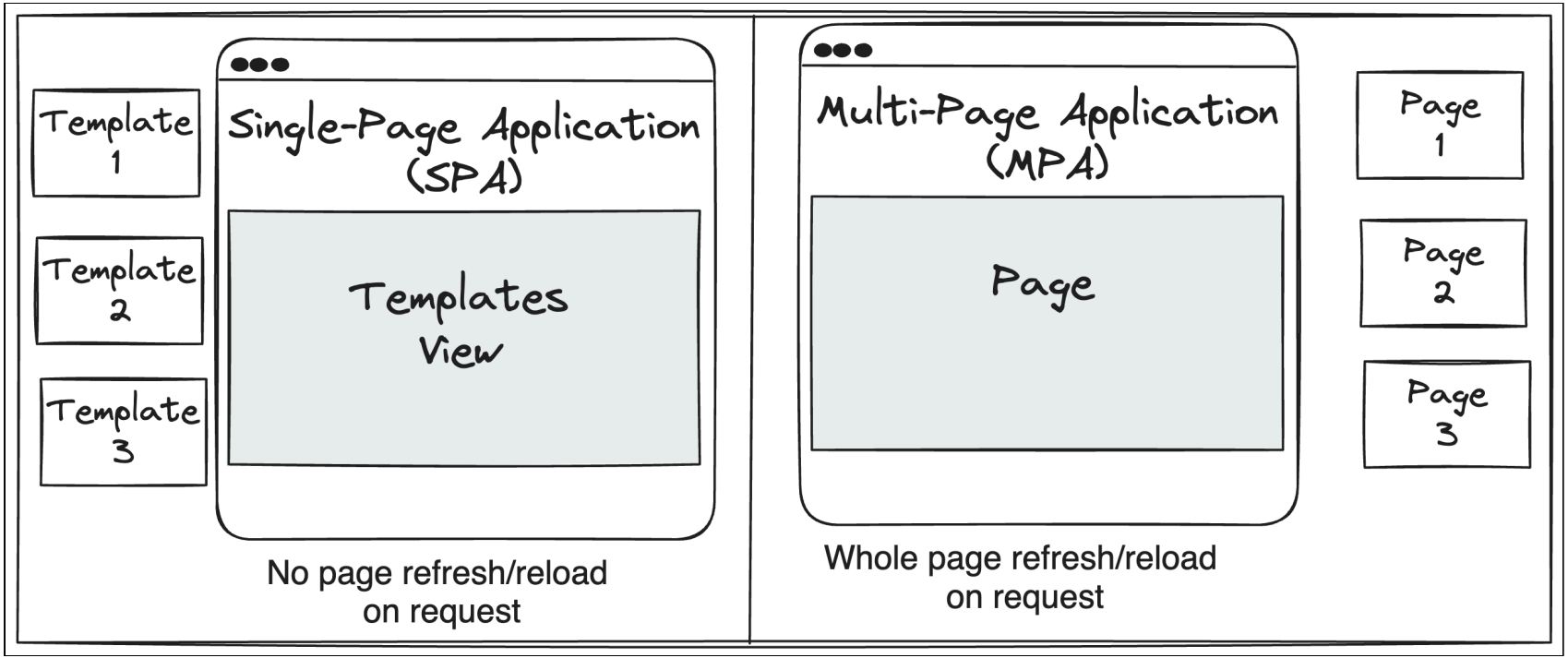
React

Table of Contents

Single Page Applications (SPAs)

- 

- Single-page applications (SPAs) load a single HTML page and dynamically update .
- Traditional multi-page applications (MPAs) load a new HTML page for each new view or interaction.

Why Single Page Applications (SPAs)

- **Faster performance:** SPAs are typically faster than MPAs because they do not have to reload the entire page for each new view or interaction. This is because SPAs use JavaScript to dynamically update the DOM (Document Object Model) of the page.
- **Better user experience:** SPAs can provide a more fluid user experience because there is no need to wait for a new page to load when the user interacts with the application. This can make SPAs feel more like native desktop applications.
- **Search Engine Optimization (SEO) Challenges**: SPAs can present SEO challenges because search engine crawlers traditionally expect full-page loads. However, modern SPAs often include techniques like server-side rendering (SSR) or pre-rendering to address these SEO issues.

Single Page Applications (SPAs) Frameworks

- SPAs are often built using JavaScript frameworks or libraries like React, Angular, or Vue.js, which provide a structured way to manage the application's components, data, and UI updates.

- There are many SPAs that provide a rich user experience without constant page reloads:
  - Gmail, Google Maps, Facebook, Netflix, Airbnb, and almost every modern day app is a SPA.

Single Page Applications (SPAs) Timeline:

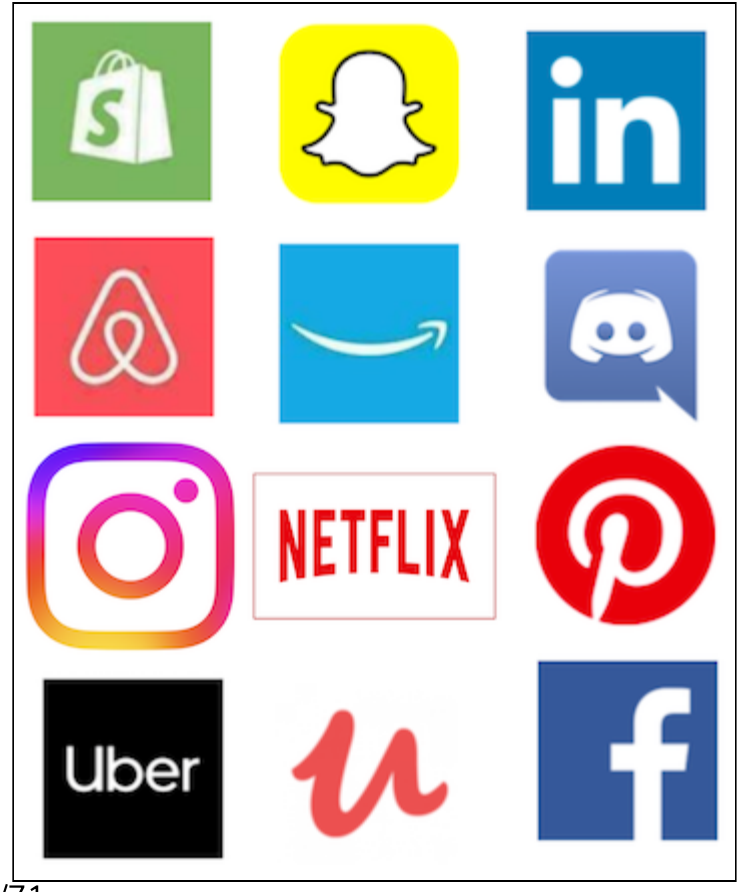
- **2002**: The concept of a single page application is introduced with the development of Outlook on the web by Microsoft, which used AJAX to enable a more desktop-like web application experience.
- **Late 2000s**: Google's Gmail and Google Maps popularize SPAs by providing fast and responsive user experiences through AJAX and dynamic updates.
- **2010**: Backbone.js is released, one of the earliest JavaScript frameworks for building SPAs, offering a basic structure for organizing client-side code.
- **2010**: AngularJS, a comprehensive JavaScript framework for building SPAs, gains significant popularity among developers.
- **2013**: React.js. Jordan Walke, a software engineer at Facebook, released a JavaScript library for building user interfaces called React.
- **2014**: Vue.js was released in February 2014 by Evan You, a former Google employee. It is a progressive JavaScript framework for building user interfaces.
- **2015**: The term "Progressive Web App" (PWA) emerges, combining SPA concepts with a number of features such as offline support, push notifications, installability, and device integration.

React Timeline

- **2011**: React was internally released by Jordan Walke, a software engineer at Facebook. He created React in response to the challenges he faced while developing Facebook's News Feed.
- **2013**: React is open-sourced (v0.3.0) on GitHub.
- **2015**: React v0.15 gains popularity within the developer community for its component-based architecture, virtual DOM, and one-way data flow.
- **2015**: React Native is introduced, extending React's concepts to mobile app development, enabling cross-platform app development.
- **2016**: React switched to major versions and announces the new v15.0 release.
- **2018**: React v16.8 introduced React Hooks, which adds new functions that simplify state management without writing a class component.
- **2022**: React v18 is released with new features such as new hooks and APIs for rendering on the client and server.
- **2024**: React v19 is released with new features such as Actions for async transitions, new hooks, `<form>` Actions and React Server Components.

React Today

- React continues to evolve and remains a dominant force in the development of web and mobile applications, with a large and active community of developers.
- Examples of companies and platforms that use React:
  - Meta (Facebook, Instagram, and WhatsApp)
  - Netflix
  - Airbnb
  - PayPal
  - DropBox
  - Pinterest



Getting Started in React (I)

Installation

- To get started in React, you will need to download and install the following:
  - Node.js because we'll need npm:
    - **npm** is the package manager for the JavaScript.
  - Browser extensions for debugging your React app:
    - React Developer Tools for Chrome OR Firefox
  - A code editor such as Visual Studio Code, which supports React.js out of the box.

Alternatively, we can use a cloud-based IDE

- **Code Sandbox** at https://codesandbox.io: We will use Code Sandbox for most examples and exercises.
- **StackBlitz** at https://vite.new/react
- **Replit** at https://replit.com

Getting Started in React (II): Create a React.js app

- We will use Vite ,pronounced like "veet", a build tool that is considered the modern standard for React development.
- Create a new React project by running the following command:

```
npm create vite@latest
```

- You will be prompted to enter the project name and select a framework (React), and then select a language or variant (JavaScript).
- Navigate to the project directory:

```
cd hello-react
```

- Install the dependencies and start the development server:

```
npm install
npm run dev
```

- Open the app in your web browser
- Open the project in your code editor (e.g., VS Code).

Getting Started in React (II): Create a React.js app using Vite (Demo)

Getting started in React (III)
Demo

main.jsx

```
1   import { StrictMode } from "react";
2   import { createRoot } from "react-dom/cl
3
4   import App from "./App.jsx";
5
6   const rootElement = document.getElementE
7   const root = createRoot(rootElement);
8
9   root.render(
10    <StrictMode>
11      <App />
12    </StrictMode>
13  );
14
```

Hello React

Open Sandbox

Console 0 Problems 0 React DevTools 0

React has three main files:

HTML Entry

`/index.html`

or

`/public/index.html`

Contains the root DOM node where React attaches your entire app

JS Entry

`src/main.jsx`

or

`/src/index.jsx`

Initializes React, renders the root component, and mounts it to the DOM

Root Component

`/src/App.jsx`

or

`/src/App.js`

Main component containing your application layout, routes, and core logic

Modern React projects typically use the `.jsx` extension for files containing React's JSX syntax. More on JSX next.

Main HTML file `index.html`

- React has a single/main HTML file (`index.html`) located in the *root* directory or *public* directory.
- The *public* directory is the location where the main HTML file and other static assets such as images, fonts, and favicon are stored.
- `index.html` is the entry point of the application and contains a root DOM element where the React application is mounted.
- When the React application is built, the contents of the public directory are copied to the build directory, and the `index.html` file is updated to include the necessary links to the built CSS and JavaScript files

index.html

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My React App</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Main script file `main.jsx`

- In a React project, `/src/main.jsx` or `/src/index.jsx` is the entry point of the application.
- This file is responsible for rendering the root component of the application and mounting it to the DOM.

main.jsx

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Main component file `App.jsx`

- In a React project, `/src/App.jsx` is a JavaScript file that contains the main component of the application.
- This component is usually called App and is responsible for rendering the main content of the application.
- The App component is typically composed of other components that are responsible for rendering specific parts of the UI.

App.jsx

```
import "./styles.css";

function App() {
  return (
    <div className="App">
      <h1>Hello React</h1>
    </div>
  );
}

export default App;
```

JSX

- The syntax you have seen in the previous slide is called **JSX** (JavaScript Syntax Extension).

1. JSX looks like HTML where elements are wrapped in one single parent element.
2. Some HTML attributes need to be named differently:

- The HTML `class` attributed is called `className` in JSX.
- The HTML `for` attribute is called `htmlFor` in JSX.

3. JavaScript code must be wrapped between two curly braces `{}`.

App.jsx

```
import { useState } from 'react';
import './App.css';

function App() {
  let d = new Date().toLocaleDateString('en-SA');
  return (
    <div className="App">
      <p>Today: {d} </p>
      <label htmlFor="name">Name:</label>
      <input id="name" type="text" />
    </div>
  );
}

export default App;
```

Terminal

```
VITE v6.3.1  ready in 2012 ms

→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

Fork on StackBlitz Editor Preview Both

React Components

- React components are the building blocks of a React application.
- They are reusable pieces of code that can be combined to create user interfaces.
- A component is a self-contained piece of code that can be reused throughout the application.
- Components can be composed together to create more complex UIs.
- Composition allows for better code reuse and makes it easier to reason about the application.
- React components follow a unidirectional data flow, where data flows from parent to child components. This makes it easier to debug and maintain the application.

React Components Example

Button.jsx

```
const Button = () => {
  return (
    <button>Click me!</button>
  );
};

export default Button;
```

React Components: Class Components

- A class component in React is a JavaScript class that extends the `React.Component` class.
- Class components are more verbose and complex to write than functional components.
- To implement a class component, we need to create a class that extends the React.Component class.

Button.jsx

```
import { Component } from 'react';

class Button extends Component{
  render() {
    return (
      <button>Click me!</button>
    );
  }
}

export default Button;
```

React Components: Functional Components

- A functional component in React is a JavaScript function that returns JSX.
- Functional components are the preferred way to write React components, as they are more concise and easier to write than class components.

Button.jsx

```
const Button = () => {

  const handleClick = () => {
    console.log("I have been clicked")
  };

  return (
    <button onClick={handleClick}>
    Click
    </button>
  );
};

export default Button;
```

React Components: Class vs Functional

Class components	Functional components
Define a state object and lifecycle hooks to manage the component's state and behavior	Define a function that returns JSX
Can use the *this* keyword to access the component's state and methods	Cannot use the *this* keyword
Are more verbose and complex to write	Are more concise and easier to write
There are some cases where class components may be necessary, such as when you need to use lifecycle hooks or manage complex state.	It is recommended to use functional components whenever possible, as they are more concise and easier to write

Props

- Components can receive data from their parent component through props (short for "properties").
- Props are how we pass data from one React component to another.
- Props are immutable, which means that they are read-only and cannot be changed by the child component.
- To pass props to a component, you simply add them as attributes to the component element.

Button.jsx

```
const Button = ({title}) => {
  return (
    <button>{title}</button>
  );
};

export default Button;
```

App.jsx

```
import Button from './Button.js'
const App = () => {
  return (
    <Button title="Submit" />
  );
};
export default App;
```

Receiving Props

There are two ways a component receives multiple props from a parent component:

1. Receiving props as a list of variables.

Image.jsx

```
const Image = ({url, text, buttonText}) => {
  return (
    <>
      <img src={url} alt={text} />
      <button>{buttonText}</button>
    </>
  );
};
export default Image;
```

App.jsx

```
import Image from 'Image.jsx'
const App = () => {
  return (
    <Image url="./logo.png" text="KAU logo"
           buttonTest="Click me" />
  );
};
export default App;
```

2. Receiving props as an object.

Image.jsx

```
const Image = (props) => {
  return (
    <>
      <img src={props.url} alt={props.text}>
      <button>{props.buttonText}</button>
    </>
  );
};
export default Image;
```

App.jsx

```
import Image from 'Image.jsx'
const App = () => {
  return (
    <Image url="./logo.png" text="KAU logo"
           buttonTest="Click me" />
  );
};
export default App
```

Props Demo

- Please see
  - `src/Button.jsx`, `src/Header.jsx`, and `src/Footer.jsx` for receiving props as an object.
  - `src/Image.jsx` for receiving props as a list of variables.

main.js

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/cl

import App from "./App.jsx";

const rootElement = document.getElementE
const root = createRoot(rootElement);

root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

Open Sandbox

Console 0 Problems 0 React DevTools 0

State

- While "props" is immutable/read-only, state is not.
- Components can also manage their own state, which can be modified using the `setState` method.
- State is private to the component and can only be modified by the component itself.
- State is a way to store data that is specific to a component and that can change over time.
- When the state changes, React re-renders the component.
- In React, a **hook** is a special function that lets you use React state and other React features without writing a class.
- `useState` is a React hook that lets you add state to a function component.

`useState`

- In programming "**State**" refers to stored information at a particular point in time.
  - Think of *state* as a fancy name for saying variable 🤔
  - State can change over time, and each change represents a different state.
- `useState` is a React Hook that lets you add a state variable to your component.

```
const [state, setState] = useState(initialState);
```

- `useState` takes one parameter, initialState, which holds the initial value of the state.
- `useState` returns an array with exactly two values:
  - The current value of the state.
  - The set function that lets you change/update the state to a different value, which will cause React to re-render the component.

State Example

App.jsx

```
import "./styles.css";
import Button from "./Button.js";

export default function App() {
  return <Button name="click me" />;
}
```

Button.jsx

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

const Button = ({ name }) => {
  const [buttonText, setButtonText] = useState(name);

  function handleClick() {
    setButtonText("I have been clicked!");
  }
  return <button onClick={handleClick}>{buttonText}</button>;
};

export default Button;
```

State Demo (I)

Edit in CodeSandbox

Button.js

```
1   import React, { useState } from "react";
2
3   const Button = ({ name }) => {
4     // set the intial value for buttonTex
5     let [buttonText, setButtonText] = use
6
7     function handleClick() {
8       setButtonText("I have been clicked!'
9       //buttonText = "Oooops I have been c
10      console.log(buttonText);
11    }
12    return <button onClick={handleClick}>{
13  };
14
```

Open Sandbox

Console 0 Problems 0 React DevTools 0

State Demo (II)

Edit in CodeSandbox

Feedback.js

```
1   import { useState } from "react";
2
3   const Feedback = ({ initLikes, initDisl
4     const [likesCount, setLikesCount] = us
5     const [dislikesCount, setDislikesCoun
6
7     function handleLike() {
8       setLikesCount(likesCount + 1);
9     }
10    function handleDislike() {
11      setDislikesCount(dislikesCount + 1);
12    }
13
14    return (
15      <div className="feedback">
16        <h1>Like or Dislike</h1>
17        <button onClick={handleLike}>
```

Like or Dislike

👍 5 👎 2

Open Sandbox

Console 0 Problems 0 React DevTools 0

Props vs State

Props	State
Props are immutable (read-only)	State is mutable (can be changed)
Props get passed to the component from the parent	State can be changed within the component itself
Props are received as functional parameter	State is received via hooks inside the component

Button.jsx *(Props Example)*

```
function handleClick() {
    console.log("I have been clicked!");
}

const Button = ({ name }) => {
    return <button onClick={handleClick}> {name}</button>
}
export default Button;
// Usage in App.jsx
<Button name="Sign up" />;
```

Button.jsx *(State Example)*

```
import { useState } from 'react';
const Button = ({ name }) => {
    const [buttonText, setButtonText] = useState(name);
    function handleClick() {
        setButtonText("I have been clicked!");
    }
    return <button onClick={handleClick}>{buttonText}</button>
}
export default Button;
// Usage in App.jsx
<Button name="Sign up" />;
```

AJAX and APIs

- Data is essential for any application to function.
- Fetching and sending data provides the information that our components need to work.
- There are two ways to work with APIs and use fetch in React apps.
  - Using regular `fetch` and `setState` hook
  - Using the `useEffect` hook

Using `fetch` and `setState`

- This example shows how to fetch data in response to a user click event.
- We send the request in the event handler and `setData` is the function that updates the *data* state.
- The component renders the fetched data once the data becomes available.
- Pros:
  - Easy to implement
  - Useful when the fetch request is triggered by a user interaction, such as a click, typing, or scrolling
- Cons:
  - Not recommended for fetching data on mount
  - Can lead to unnecessary re-renders

Edit in JSFiddle

- React
- Result

```
const useState = React.useState;

const Article = ({ id }) => {
    const [data, setData] = useState(null);

    const fetchData = async () => {
        const response = await fetch(`https://jsonplaceholde
        const respObj = await response.json();
        setData(respObj);
    }
    return (
        <div className="App">
            <button onClick={fetchData}>Fetch Data</button>
            {data && <div>{JSON.stringify(data)}</div>}
        </div>
    );
}

const root = ReactDOM.createRoot(document.getElementById("rc
root.render(<Article id="1" />);
```

Using `fetch` and `setState` Demo

Edit in CodeSandbox

GitHubUser.js

```
1   import { useState } from "react";
2
3   const GitHubUser = () => {
4     const [userName, setUserName] = useSta
5     const [userData, setUserData] = useSta
6
7     async function handleClick() {
8       const response = await fetch(`https:
9       const data = await response.json();
10      setUserData(data);
11    }
12
13    function handleChange(e) {
```

Enter any GitHub username

Search

Open Sandbox

Console 0 Problems 0 React DevTools 0

Using `fetch` and `useEffect`

- We first import `useState`, and `useEffect` from the `react` library.
- We define a functional component called *Article* that takes an id as *props*.
- Inside this component, we declare a state variable `data` and a function `setData` to update this state. The initial value of data is `null`
- We then declare the `useEffect` hook and perform a `fetch` request inside it. More on this hook next.
- We convert the response to JSON and use `setData` to update our state variable *data*.

```
import {useState, useEffect} from "react";

const Article = ({ id }) => {
  const [data, setData] = useState(null);

  useEffect(() => {
      const fetchData = async () => {
          const response = await fetch(
            `https://jsonplaceholder.typicode.com/posts/${id}`);
          const respObj = await response.json();
          setData(respObj);
      }
      fetchData();
  }, [id]);

  return (
      <div className="App">
          {data && <div>{JSON.stringify(data)}</div>}
      </div>
  );
}
```

`useEffect`

- `useEffect` is a built-in hook in React that allows you to perform side effects in your function components.
- A side effect could be data fetching, subscribing to a service, manually changing the DOM, etc.

```
useEffect(() => {
    ............
}, []);
```

This is the effect function
that should perform a side effect.
It will run when the component mounts
and when the dependency list changes.

It may return a clean up function
to unregister events or clean up
resources.

Optional dependency list.
If not passed, the side effect function will run on every mount.
[] If an empty array is passed, it will run the side effect function once.
[id,name] if a list is passed, it will run the effect function when
any variable in the list changes.

Using `fetch` and `useEffect`

- `useEffect` is a React hook hook for performing side effects in components.
- When to use `fetch` and `useEffect`?
  - When the data being fetched is essential to the component's initial render
  - When the data being fetched is updated frequently
- Fetching data from within a React component requires us to orchestrate both the `useEffect` and `useState` hooks.
  - The `useEffect` hook is used to make the fetch request.
  - The `useState` hook is used to store the response in state
- Pros
  - Can fetch data on mount or whenever other dependencies change
  - Avoids unnecessary re-renders
- Cons
  - More complex to use

Using `fetch` and `useEffect` (Demo)

Edit in CodeSandbox

GitHubUser.js

```
import { useState, useEffect } from "rea

const GitHubUser = () => {
  const [input, setInput] = useState("g
  const [userName, setUserName] = useSta
  const [data, setData] = useState(null)

  useEffect(() => {
    const fetchData = async () => {
      console.log("Sent a request!")
      const response = await fetch(`http
      const data = await response.json()
      setData(data);
    };
```

Enter GitHub username

Submit

GitHub

How people build software.

Open Sandbox

Console 2 Problems 0 React DevTools 0

React Router

Routing

- Routing helps direct users to different pages based on the URL they have entered or clicked.
- In traditional multi-page applications (MPAs), the browser requests a document from a web server and renders the HTML sent from the server.
- In Single Page Applications (SPAs), routing refers to the ability to navigate between different parts of the application without a full page refresh.
- In single-page applications (SPAs), routing is done on the client side without making another request for another document from the server.
- This enables faster user experiences and allows you to define routes for the SPA application and render different components based on the current route.

Routing in React

- Recall that React is often considered the "V" in MVC, which stands for Model-View-Controller.
  - React does not have a built-in concept of a "Controller" or routing.
  - Routing functionality can be added to a React application using third-party libraries
- React Router is a third-party library used for managing routing in React applications.
  - It adds routing to Single Page Applications (SPA) and navigation without page reload.
  - It enables component-based routing, where different routes can render different components.
  - It provides tools to easily retrieve and use URL parameters and query strings.
  - It supports programmatic navigation, allowing you to trigger navigation from functions or hooks.

React Router

- React Router can be installed via npm: `npm install react-router` and use it as follows:

```
import React from 'react';
import {BrowserRouter, Routes, Route, Link } from "react-router";
import Home from './Home';
import About from './About';
import NotFound from './NotFound';

const App = () => {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/about">About</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </BrowserRouter>
  );
}
export default App;
```

React Router: Main Concepts

- **Routes:** Routes group the different route for pages or views in your application.
- **Route:** An object or Route Element typically with a shape of `{ path, element }` or `<Route path element>`
- **Navigation:** Any change to the URL. There are two ways to navigate in React Router:
  - **Declarative navigation:** means that you define your routes upfront and React Router will take care of rendering the appropriate components based on the current URL.
  - **Imperative navigation** means that you explicitly trigger navigation events, such as clicking a button or calling a function.

React Router: Declarative Navigation

- We define routes explicitly and link to them using the `<Link>` component.
  - The `<Link>` component is a declarative way to navigate between routes. It renders an HTML `<a>` tag with a `href` attribute that points to the desired route.

Usage:

1. Declare routes in the main/root component (`App.js`)
2. Link to routes using the `<Link>` component in any component.

```
import {BrowserRouter, Routes, Route, Link } from "react-router";
const App = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/signup" element={<SignUp />} />
      </Routes>
    </BrowserRouter>
  );
}
export default App;
```

```
<Link to="/">Home</Link>
<Link to="/signup">Sign up</Link>
```

React Router: Imperative navigation

- To navigate in code (programmatically), we use the `useNavigate` hook in React Router.

Usage:

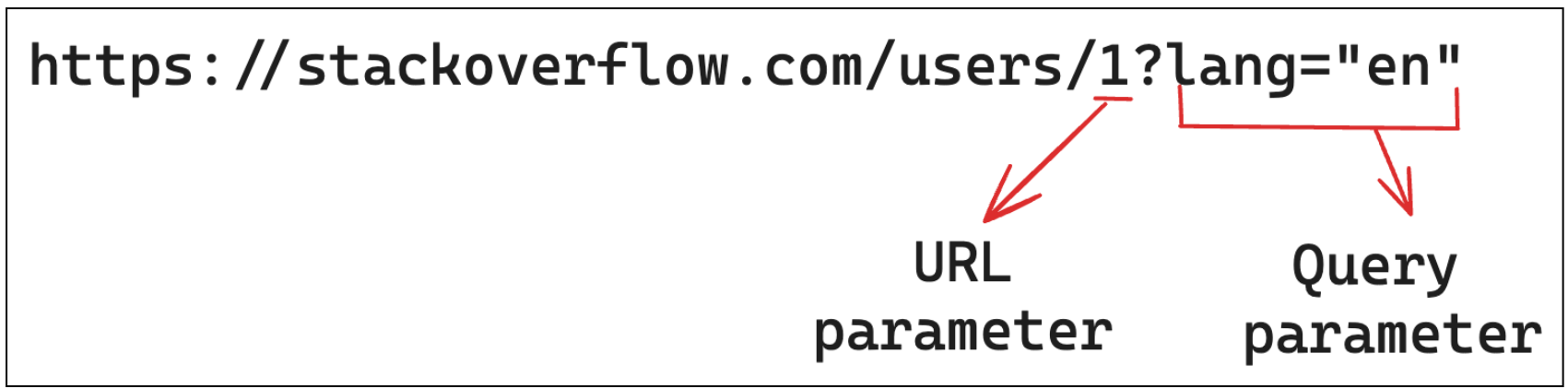
1. Import `BrowserRouter`, `Routes`, and `Route` from `react-router` library.
2. Declare routes in the main component (`App.js`).
3. Use the `useNavigate` hook anywhere you want to navigate programmatically to a route.

```
import {BrowserRouter, Routes, Route} from "react-router";
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/signup" element={<SignUp />} />
      </Routes>
    </BrowserRouter>
```

```
import {useNavigate} from 'react-router';
const Header = () => {
    const navigate = useNavigate();
    function handleClick() {
        navigate('/sign-up');
    }
    return (
        <div>
            <button onClick={handleClick}>Sign up</button>
        </div>
    )
};
export default Header;
```

React Router: Nested Routes, URL parameters, and Query parameters

- Nested routing allows you to organize your routes into hierarchies, making it easier to manage complex navigation structures.
- We can use nested routes to access **URL parameters** and **query parameters** using the `useParams` and `useSearchParams` hooks respectively.



React Router: Nested Routes Example

App.js

```
import {BrowserRouter, Routes, Route} from "react-router";
    <BrowserRouter>
      <Routes>
        <Route path="/users" element={<Users />}>
          <Route path=":id" element={<User />} />
        </Route>
      </Routes>
    </BrowserRouter>
```

Users.js

```
const Users = () => {
  return (
    <div>
      <Link to="1">User 1</Link> |
      <Link to="2">User 2</Link>
      <Routes>
        <Route path=":id" element={<User />} />
      </Routes>
    </div>
```

User.js

```
import {
  useParams,
  useSearchParams
  } from 'react-router';

const User = () => {
 const params = useParams();
 const [searchParams] = useSearchParams();
 const id = params.id;
 const query = searchParams.get('query');

 return (
   <div>
     <p>ID: {id}</p>
     <p>Query: {query}</p>
   </div>
 );
}
export default User;
```

React Router Demo

Edit in CodeSandbox

index.js

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/cl

import App from "./App.jsx";

const rootElement = document.getElementE
const root = createRoot(rootElement);

root.render(
  <StrictMode>
    <App />
  </StrictMode>
);

```

Open Sandbox

Console 0 Problems 0 React DevTools 0

Authentication and Protected Routes

- Web applications contain public and private pages that are protected behind user authentication.
- Authentication can be implemented by rolling your own authentication system or using a third-party provider
- Rolling your own authentication system on the server:
  - Requires robust security measures
  - Adds significant security and maintenance challenges
- Using third-party providers:
  - Provides robust and scalable solutions
  - Reduces the overhead of maintenance and updates
  - Cuts development effort
  - Examples: Google Firebase, Amazon Cognito, and Microsoft Azure Active Directory.

Firebase Authentication in React

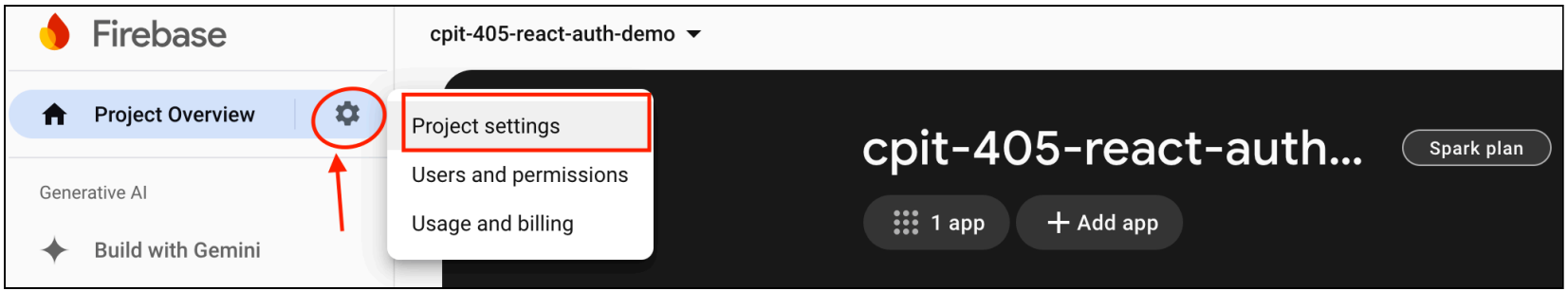
- We will use Firebase for its popularity and ease of integration with React.
- Firebase comes with no cost for the first 50,000 monthly active users
- Create a new react app:

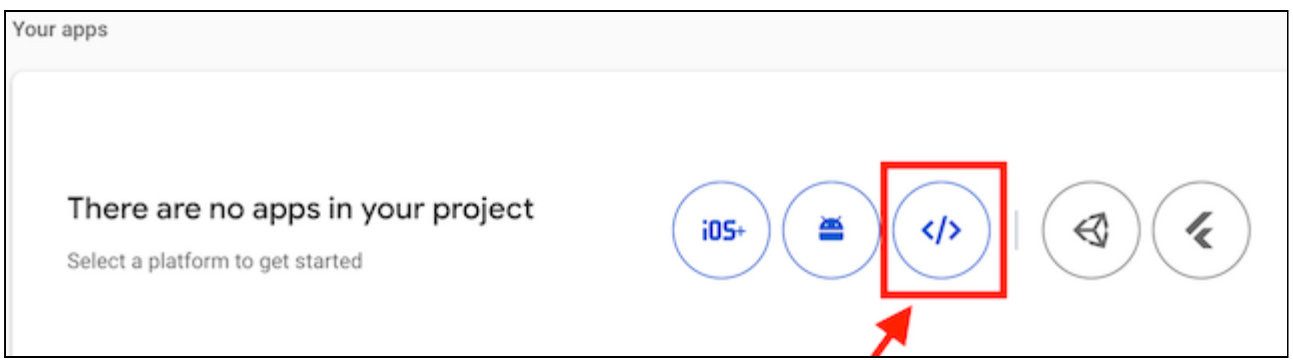
```
npm create vite@latest
```

- Enter a name for the app (e.e., react-router-demo) and select React + JavaScript.
- Go to the project directory `cd react-router-demo` and install the dependencies using `npm install`
- Install Firebase and React-Router libraries:

```
npm install firebase react-router
```

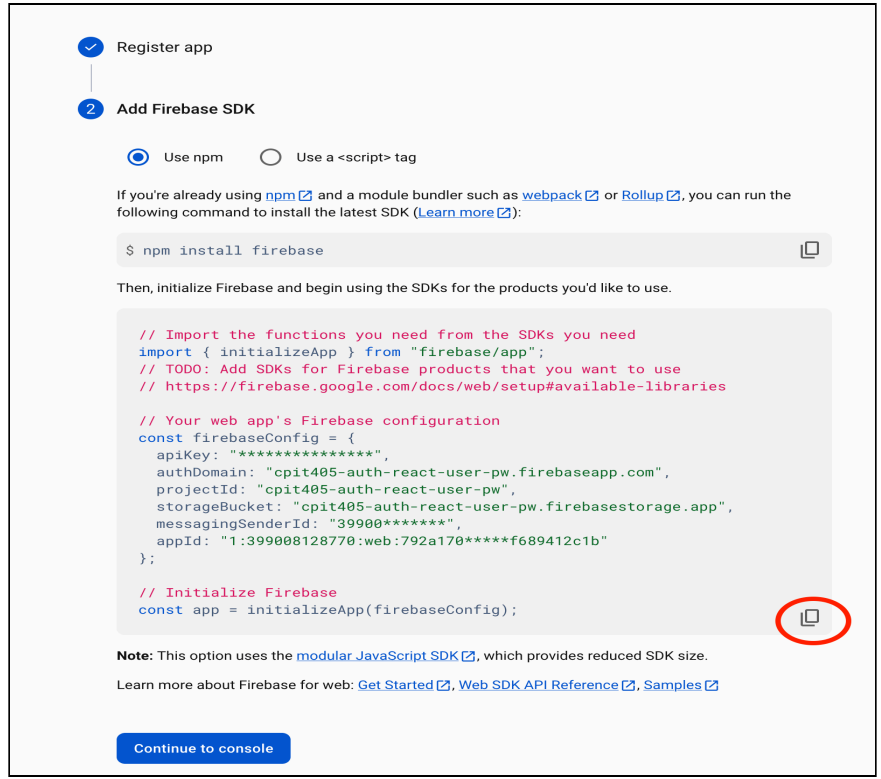
Setting up Firebase (I)

- Create a new Firebase project at console.firebase.google.com
- Click on *Project Settings*
  - 

- Scroll down to *Your Apps* and register your web app
  - 

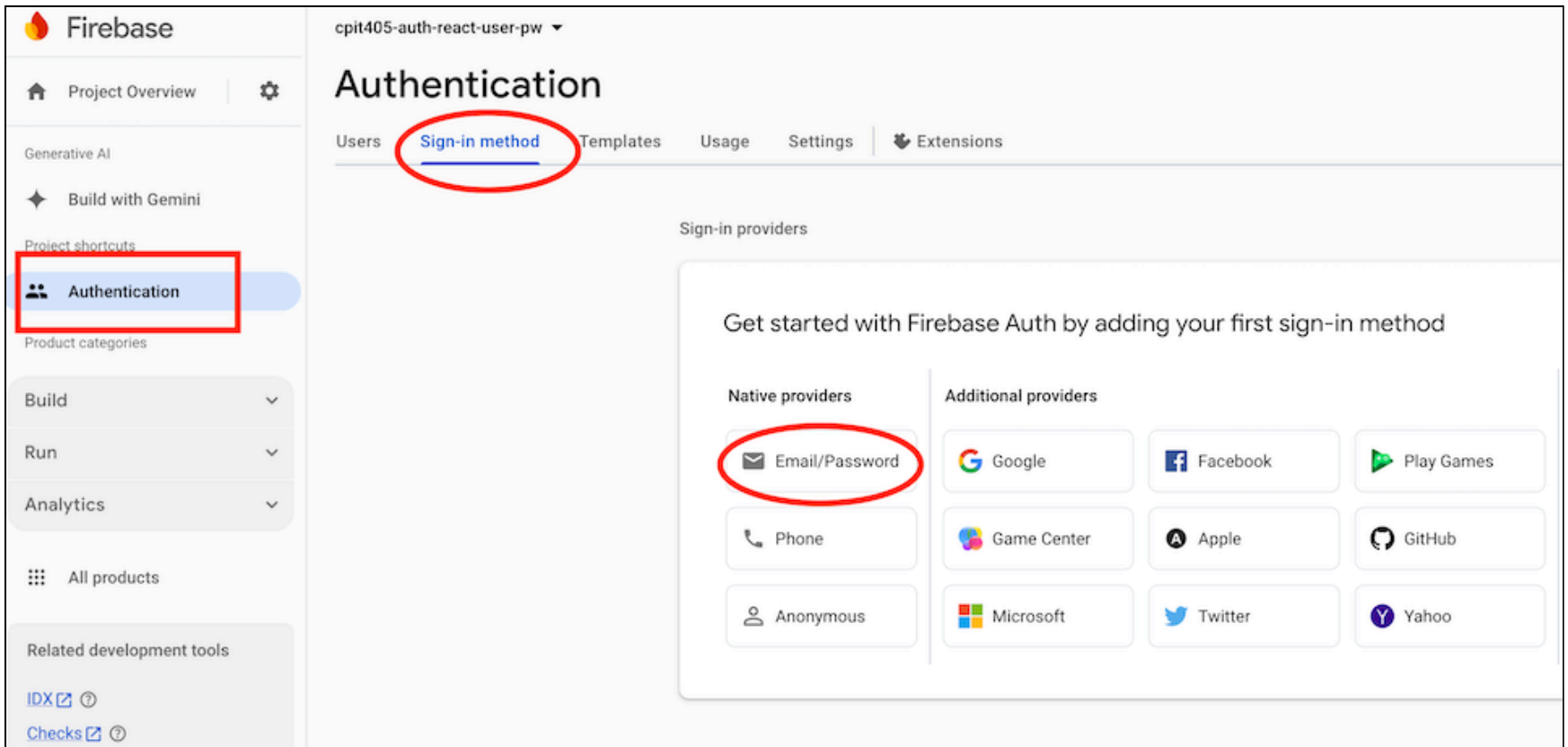
Setting up Firebase (II)

- Enable Authentication (Email/Password) in Firebase Console
- Copy your Firebase config (only `firebaseConfig` object)



Setting up Firebase (III)

- Enable Authentication (Email/Password) in Firebase Console



Setting up Firebase (IV)

- create a new file `firebase.js` under `/src/` and add your Firebase config
- Add the following code and paste the `firebaseConfig` object from the previous step.

```
import { initializeApp } from 'firebase/app';
import { getAuth } from 'firebase/auth';

const firebaseConfig = {
  // Your Firebase config object from
  // the firebase dashboard
};

const app = initializeApp(firebaseConfig);
export const auth = getAuth(app);
```

Create Authentication Context

- Create a new file `src/contexts/AuthContext.jsx`

```
import { createContext, useContext, useState, useEffect } from 'react';
import { auth } from '../firebase';
import {
  createUserWithEmailAndPassword,
  signInWithEmailAndPassword,
  signOut,
  onAuthStateChanged
} from 'firebase/auth';

const AuthContext = createContext();

export const useAuth = () => useContext(AuthContext);

export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
```

Protected Route Component

- Create a custom component that wraps up protected routes
- Crate a new file `src/components/ProtectedRoute.jsx`

```
import { Navigate } from 'react-router';
import { useAuth } from '../contexts/AuthContext';

export default function ProtectedRoute({ children }) {
  const { user } = useAuth();

  if (!user) {
    return <Navigate to="/login" />;
  }

  return children;
}
```

Create a Login Component

- Create a new Login component at `src/components/Login.jsx`

```
import { useState } from 'react';
import { useNavigate } from 'react-router';
import { useAuth } from '../contexts/AuthContext';

export default function Login() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');
  const { login } = useAuth();
  const navigate = useNavigate();

  async function handleSubmit(e) {
    e.preventDefault();
    try {
      await login(email, password);
      navigate('/dashboard');
    } catch (error) {
      setError('Failed to login');
    }
  }
```

Create a Private/Protected Component

- Create a new Dashboard component at `src/components/Dashboard.jsx`
- This component will be restricted to authenticated users

```
import { useAuth } from '../contexts/AuthContext';
import { useNavigate } from 'react-router';

export default function Dashboard() {
  const { user, logout } = useAuth();
  const navigate = useNavigate();

  async function handleLogout() {
    try {
      await logout();
      navigate('/login');
    } catch {
      console.error('Failed to logout');
    }
  }

  return (
    <div>
```

Setting up the main App Component with Routes

- Now we can set up the main App component `App.js`

```
import { BrowserRouter as Router, Routes, Route } from 'react-router';
import { AuthProvider } from './contexts/AuthContext';
import ProtectedRoute from './components/ProtectedRoute';
import Home from './components/Home';
import Login from './components/Login';
import Dashboard from './components/Dashboard';

function App() {
  return (
    <Router>
      <AuthProvider>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/login" element={<Login />} />
          <Route
            path="/dashboard"
            element={
              <ProtectedRoute>
```

- Run and test the Implementation

```
npm run dev
```

Wrapping up Authentication

- Always wrap your app with `AuthProvider`
- Use `useAuth` hook to access authentication state
- Protect routes using `ProtectedRoute` component
- Handle loading states appropriately
- Use `navigate` for programmatic navigation in react-router

Deploying React Apps into GitHub pages

1. Create a repo on GitHub, commit, and push to main.
2. Edit your `package.json` file and add a `homepage` field:

```
{
  "name": "react-gh-pages",
  "homepage": "https://username.github.io/repo-name",
}
```

- Replace `username` and `repo-name` with your username and repo name on GitHub.

3. Install gh-pages

```
npm install gh-pages
```

4. Add deploy to scripts in package.json

```
  "scripts": {
    "predeploy": "npm run build",
    "deploy": "gh-pages -d build",
```

5. Deploy the web app by running

```
npm run deploy
```

Deploying React Apps into GitHub pages (Cont.)

- The web app should be live on GitHub pages
  - Example: https://cpit405.github.io/react-gh-pages/
  - Source code (package.json): https://github.com/cpit405/react-gh-pages/

Practical Projects

Integrating React, React-Router, and APIs

Project 1: Link Shrinker - Shorten and Share Your Links

Project 2: Recipe Finder - search for recipes

Project 1: Link Shrinker (I)

Shorten and Share Your Links

Develop a React application that enables users to shorten long URLs into concise and shareable links. The application should have the following features:

- URL Shortening: Takes a long URL and returns a shortened link.
- Custom short URLs: Allow users to create custom shortened URLs, making them more customizable.

Project 1: Link Shrinker (II)

Edit in CodeSandbox

App.js

```
import { BrowserRouter, Routes, Route, l
import URLShrinker from "./URLShrinker.j
import About from "./About.js";

import "./styles.css";

export default function App() {
  return (
    <BrowserRouter>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
```

Home About

Link Shrinker

Long URL:

Enter short code:

Open Sandbox

Console 0 Problems 0 React DevTools 0

Project 2: Recipe Finder (I)

search for recipes

Create a React application that allows users to search for recipes and view recipe details including ingredients, instructions, and images.

- We'll use a free API, Spoonacular API, to fetch recipes.
  - Sign up for an account and generate a new api key.
  - Once you're logged in, navigate to the "Profile" section and view or generate an API key.
  - Read the docs on how to use the API end point api.spoonacular.com/recipes/complexSearch
  - Below is an example of using this end point. You may copy this into Postman and see the JSON response.

```
https://api.spoonacular.com/recipes/complexSearch?query=pasta&apiKey=PASTE_YOUR_API_KEY_HERE
```

Project 2: Recipe Finder (II)

Edit in CodeSandbox

Repository cpit405/react-recipe-finder main Open Editor

EXPLORER

REACT-RECIPE-FINDER

- .codesandbox
- .devcontainer
- node_modules
- public
- src
  - components
- App.css
- App.js
- App.test.js

OUTLINE

TIMELINE

App.js

srcApp.js...

```
1  import { BrowserRouter, Routes, Route, Link } from 'react-router-dom'
2  import Home from './components/Home';
3  import About from './components/About';
4  import './App.css';
5  import Recipe from './components/Recipe';
6
7  function App() {
```

PROBLEMS OUTPUT TERMINAL PORTS 1 start

```
webpack compiled with 1 warning
```

CodeSandbox - Devbox (Web) 0 0 1 1 Port: 3000 Ln 1, Col 1 Spaces: 2 UTF-8 LF JavaScript JSX Layout: US

Putting all things together

It's time to put your React skills into test and build a React app. In a group of two students, create one of the following apps:

---

1) Schedule Mate: Find the Perfect Meeting Time

Create a React application that simplifies scheduling meetings by finding the most suitable time slot that works for all participants. The app should offer basic scheduling features akin to when2meet.com/.

2) Weather Wonder: Find current weather conditions

Create a React application that displays the current weather conditions for a specific location with routing options to see more details.

Wrapping up

- **React.js**: is a JavaScript library for building user interfaces (UIs) based on reusable components.
- **Props**: immutable (read-only) data passed from parent components to child components to control their behavior and content.
- **State**: mutable (changeable) data managed by a component, determining its appearance and value.
- **JSX**: an extension to JavaScript syntax that allows embedding HTML-like code for UI description.
- **React hooks**: special functions that let you "hook into" React state and lifecycle features in functional components.
  - The `useState` hook allows you to manage local state in functional components.
  - The `useEffect` hook allows you to perform side effects in functional components, such as data fetching, subscriptions, etc.
- **React Router**: a powerful routing library that defines routes and handles navigation between different components.

The most effective way to learn React is to practice by building small projects and reading the official React documentation and tutorials at react.dev.

END