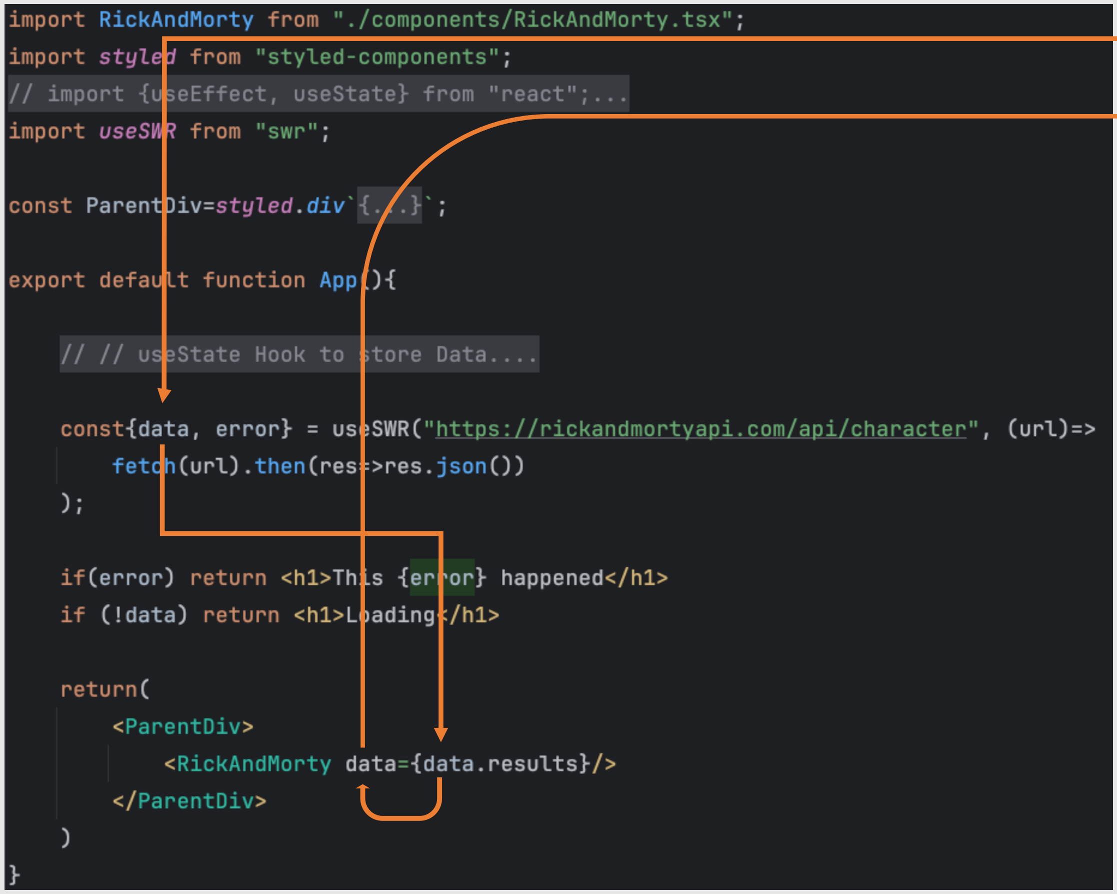
Props/Context

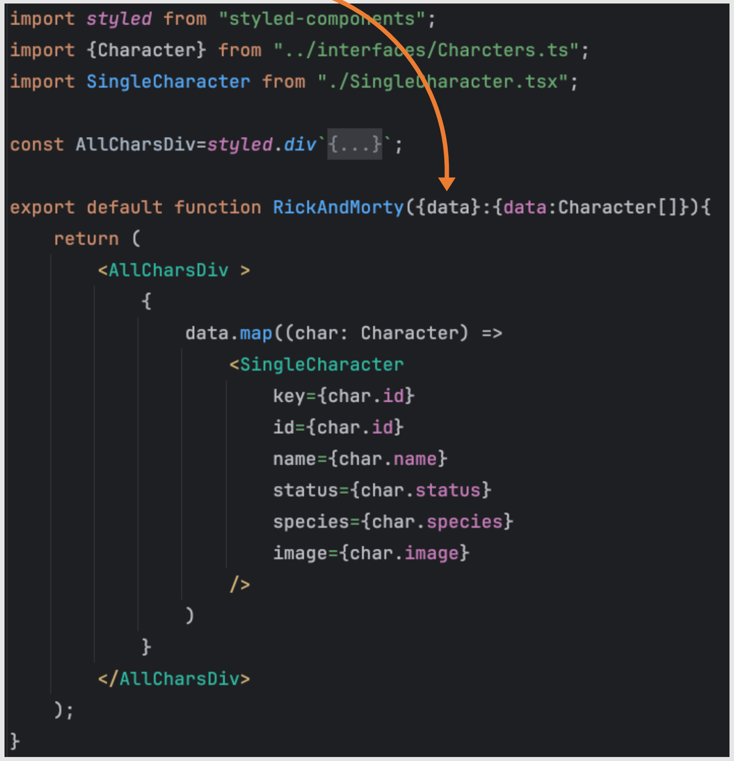
EX:

Props are simultaneously function parameters and HTML attributes.



```
import RickAndMorty from "./components/RickAndMorty.tsx";
import styled from "styled-components";
// import {useEffect, useState} from "react";...
import useSWR from "swr";

const ParentDiv=styled.div`{...}`;

export default function App(){

    // // useState Hook to store Data....

    const{data, error} = useSWR("https://rickandmortyapi.com/api/character", (url)=>
        fetch(url).then(res=>res.json())
    );

    if(error) return <h1>This {error} happened</h1>
    if (!data) return <h1>Loading</h1>

    return(
        <ParentDiv>
            <RickAndMorty data={data.results}/>
        </ParentDiv>
    )
}
```

Here data is an object, we passed it as a prop to <RickAndMorty/> Component

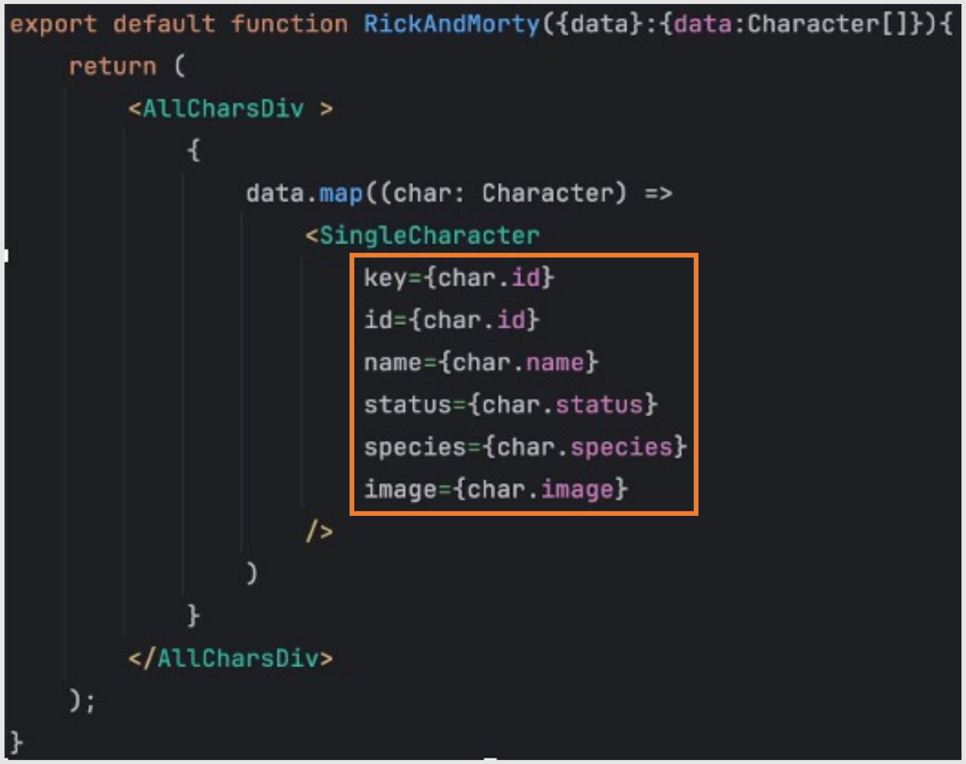


```
import styled from "styled-components";
import {Character} from "../interfaces/Charcters.ts";
import SingleCharacter from "./SingleCharacter.tsx";

const AllCharsDiv=styled.div`{...}`;

export default function RickAndMorty({data}:{data:Character[]}){
    return (
        <AllCharsDiv >
            {
                data.map((char: Character) =>
                    <SingleCharacter
                        key={char.id}
                        id={char.id}
                        name={char.name}
                        status={char.status}
                        species={char.species}
                        image={char.image}
                    />
                )
            }
        </AllCharsDiv>
    );
}
```

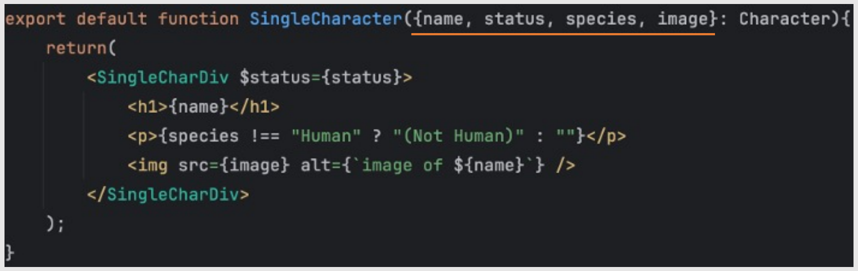
Props/Context

Using the key-word **props**

Then From the **<RickAndMorty/>** Component, we **map()** over the key : value pairs inside **data** and pass selected fields as **props** to **<SingleCharacter/>** Component

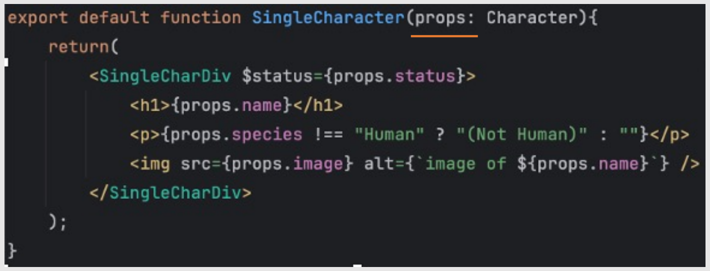


```
export default function RickAndMorty({data}:{data:Character[]}){
    return (
        <AllCharsDiv >
            {
                data.map((char: Character) =>
                    <SingleCharacter
                        key={char.id}
                        id={char.id}
                        name={char.name}
                        status={char.status}
                        species={char.species}
                        image={char.image}
                    />
                )
            }
        </AllCharsDiv>
    );
}
```



```
export default function SingleCharacter({name, status, species, image}: Character){
    return(
        <SingleCharDiv $status={status}>
            <h1>{name}</h1>
            <p>{species !== "Human" ? "(Not Human)" : ""}</p>
            <img src={image} alt={`image of ${name}`} />
        </SingleCharDiv>
    );
}
```

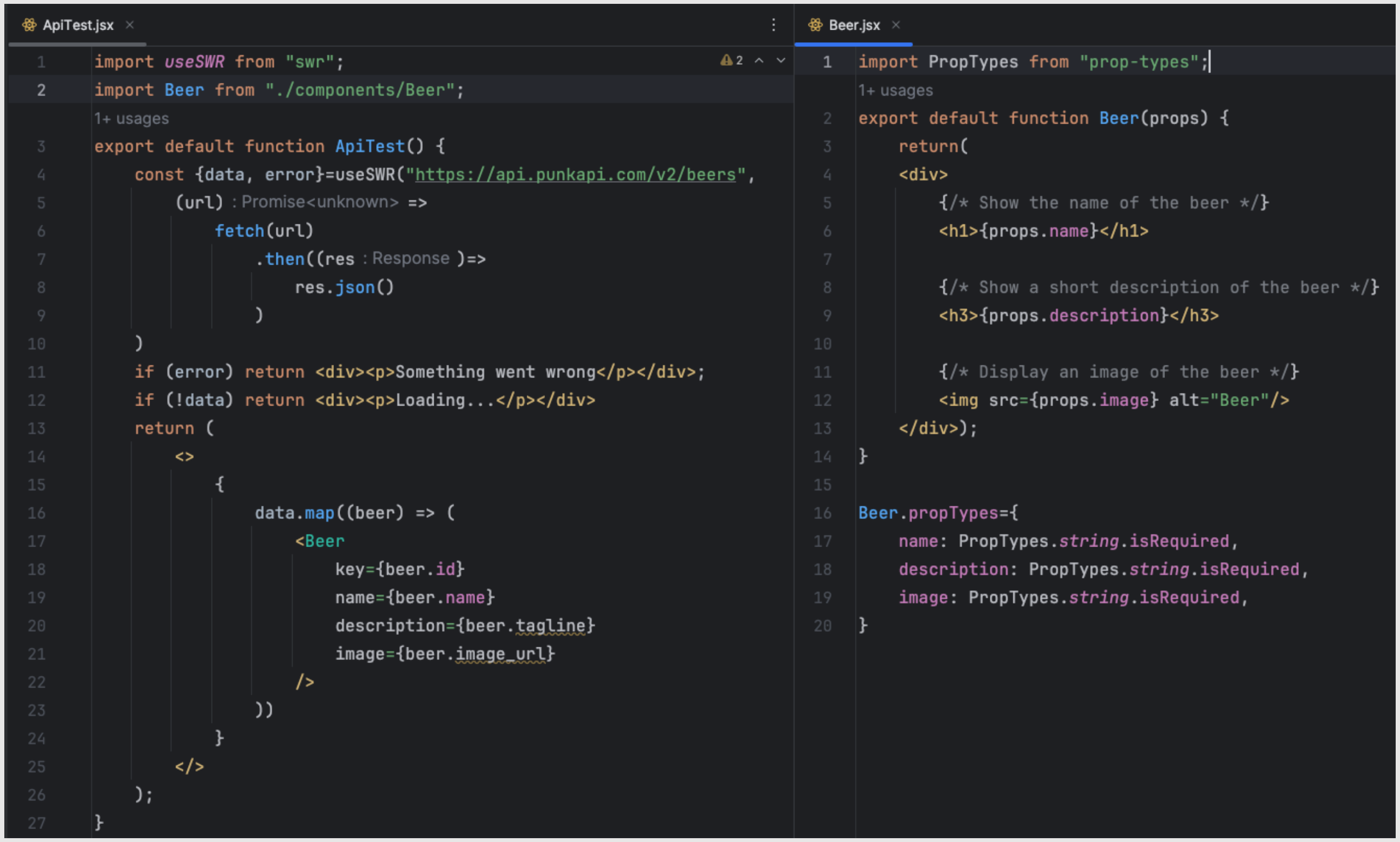
In the **SingleCharacter()** component we could either explicitly reference those props or we could refer to all of them as **props** (this is the conventional method).



```
export default function SingleCharacter(props: Character){
    return(
        <SingleCharDiv $status={props.status}>
            <h1>{props.name}</h1>
            <p>{props.species !== "Human" ? "(Not Human)" : ""}</p>
            <img src={props.image} alt={`image of ${props.name}`} />
        </SingleCharDiv>
    );
}
```

Props/Context

So far, you have learned that we can pass information to components via **props**



ApiTest.jsx

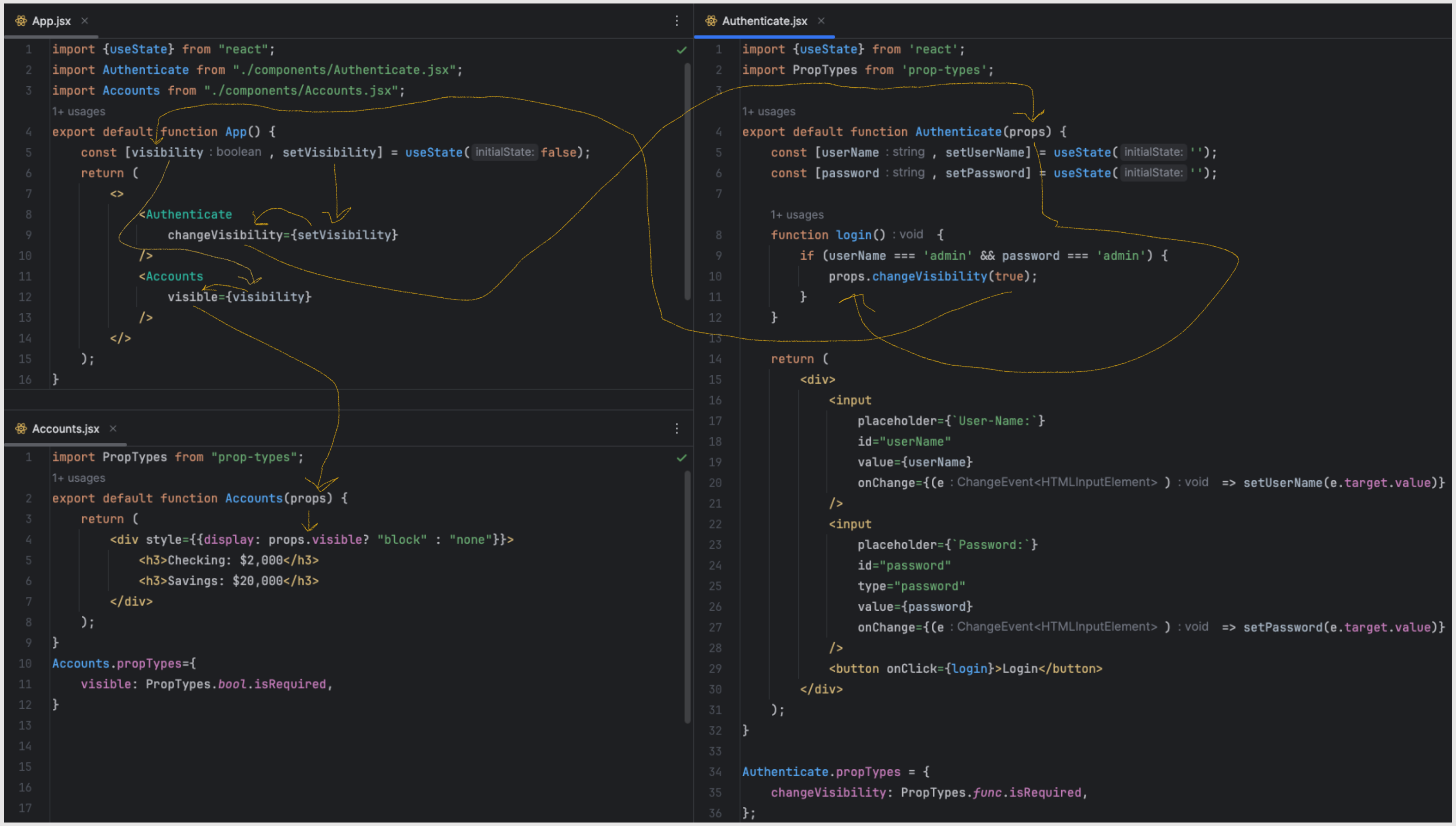
```jsx
import useSWR from "swr";
import Beer from "./components/Beer";
1+ usages
export default function ApiTest() {
    const {data, error}=useSWR("https://api.punkapi.com/v2/beers",
        (url) : Promise<unknown> =>
            fetch(url)
                .then((res : Response )=>
                    res.json()
                )
    )
    if (error) return <div><p>Something went wrong</p></div>;
    if (!data) return <div><p>Loading...</p></div>
    return (
        <>
            {
                data.map((beer) => (
                    <Beer
                        key={beer.id}
                        name={beer.name}
                        description={beer.tagline}
                        image={beer.image_url}
                    />
                ))
            }
        </>
    );
}
```

Beer.jsx

```jsx
import PropTypes from "prop-types";
1+ usages
export default function Beer(props) {
    return(
    <div>
        {/* Show the name of the beer */}
        <h1>{props.name}</h1>

        {/* Show a short description of the beer */}
        <h3>{props.description}</h3>

        {/* Display an image of the beer */}
        <img src={props.image} alt="Beer"/>
    </div>);
}

Beer.propTypes={
    name: PropTypes.string.isRequired,
    description: PropTypes.string.isRequired,
    image: PropTypes.string.isRequired,
}
```

Props/Context

Props can also travel back



App.jsx

```jsx
import {useState} from "react";
import Authenticate from "./components/Authenticate.jsx";
import Accounts from "./components/Accounts.jsx";
1+ usages
export default function App() {
    const [visibility : boolean , setVisibility] = useState( initialState: false);
    return (
        <>
            <Authenticate
                changeVisibility={setVisibility}
            />
            <Accounts
                visible={visibility}
            />
        </>
    );
}
```

Accounts.jsx

```jsx
import PropTypes from "prop-types";
1+ usages
export default function Accounts(props) {
    return (
        <div style={{display: props.visible? "block" : "none"}}>
            <h3>Checking: $2,000</h3>
            <h3>Savings: $20,000</h3>
        </div>
    );
}
Accounts.propTypes={
    visible: PropTypes.bool.isRequired,
}
```

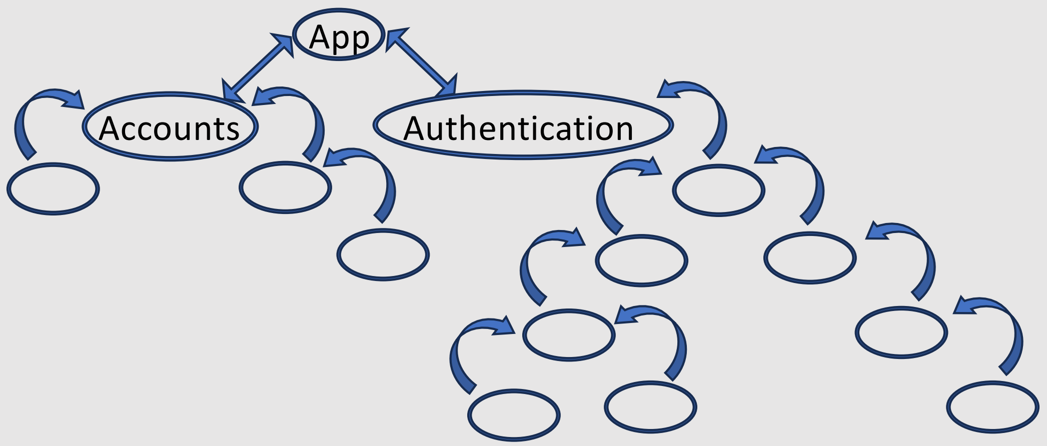
Authenticate.jsx

```jsx
import {useState} from 'react';
import PropTypes from 'prop-types';

1+ usages
export default function Authenticate(props) {
    const [userName : string , setUserName] = useState( initialState: '');
    const [password : string , setPassword] = useState( initialState: '');

    1+ usages
    function login() : void  {
        if (userName === 'admin' && password === 'admin') {
            props.changeVisibility(true);
        }
    }

    return (
        <div>
            <input
                placeholder={`User-Name:`}
                id="userName"
                value={userName}
                onChange={(e : ChangeEvent<HTMLInputElement> ) : void  => setUserName(e.target.value)}
            />
            <input
                placeholder={`Password:`}
                id="password"
                type="password"
                value={password}
                onChange={(e : ChangeEvent<HTMLInputElement> ) : void  => setPassword(e.target.value)}
            />
            <button onClick={login}>Login</button>
        </div>
    );
}

Authenticate.propTypes = {
    changeVisibility: PropTypes.func.isRequired,
};
```

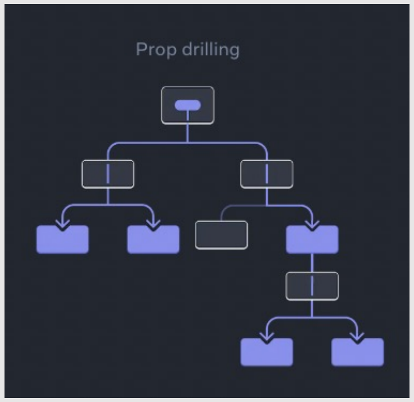
Props/Context

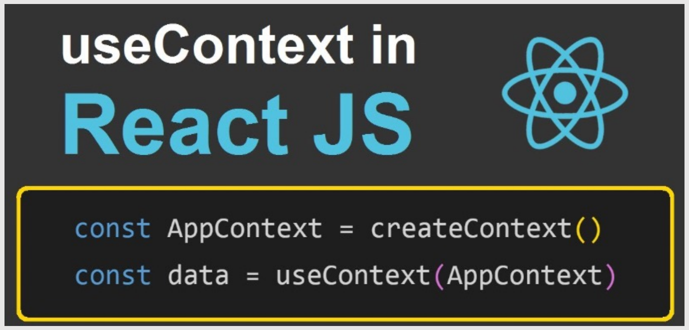
Components can communicate with each other via a parent component, but what if there are many child components?

- You could use tree traversal algorithms, such as **BFS**, **DFS**.
- You could also balance the tree with with algorithms such as **AVL.**
- You may also try to move more complex components to the "root" (inversion technique).

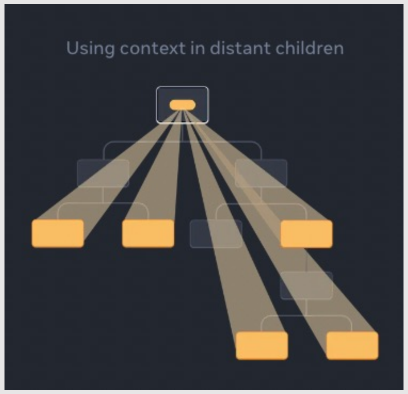


But in most cases, using **React Context** would be the most optimized solution in React



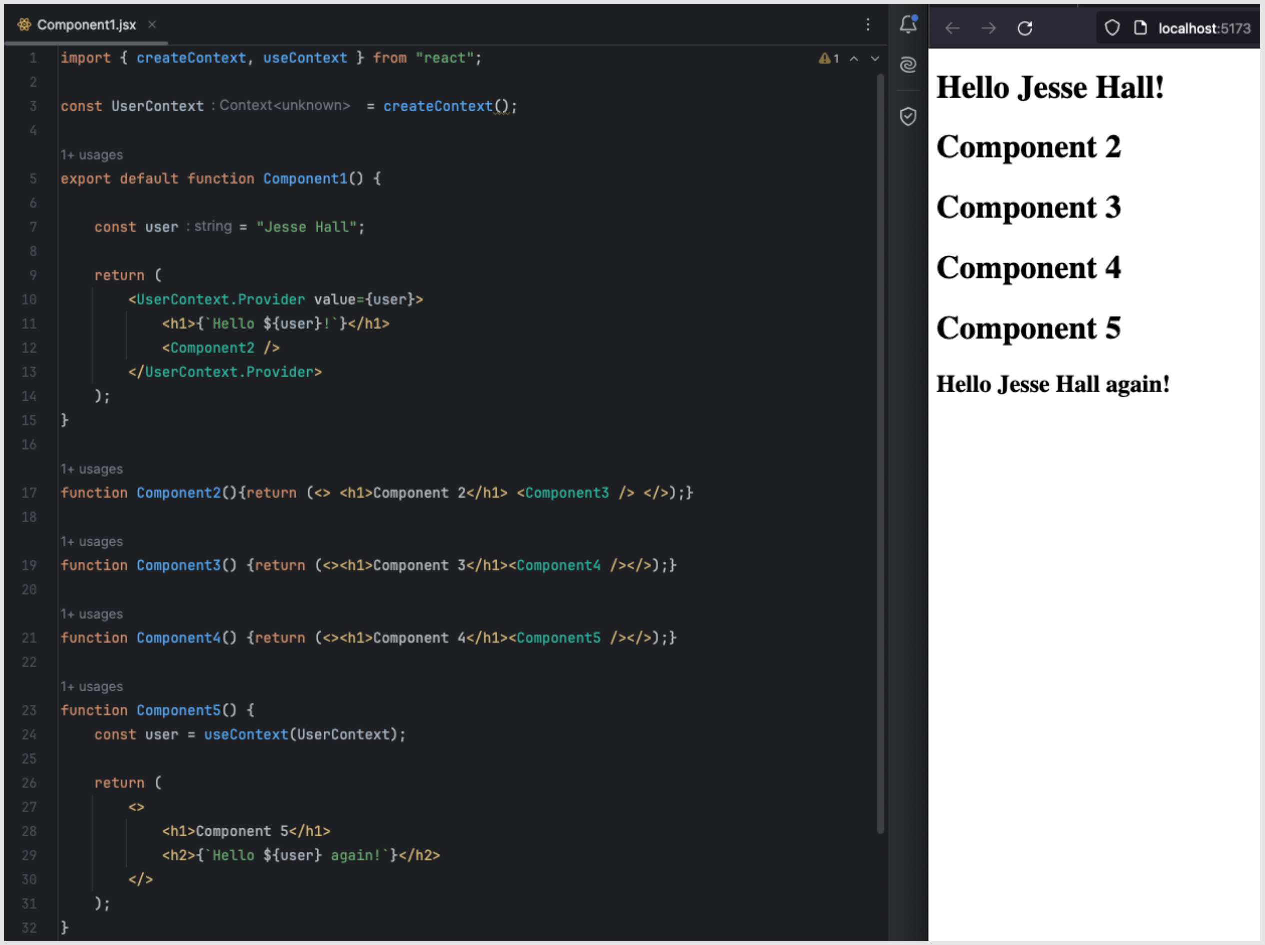


```
const AppContext = createContext()
const data = useContext(AppContext)
```



Props/Context

W3-School Example


Component1.jsx

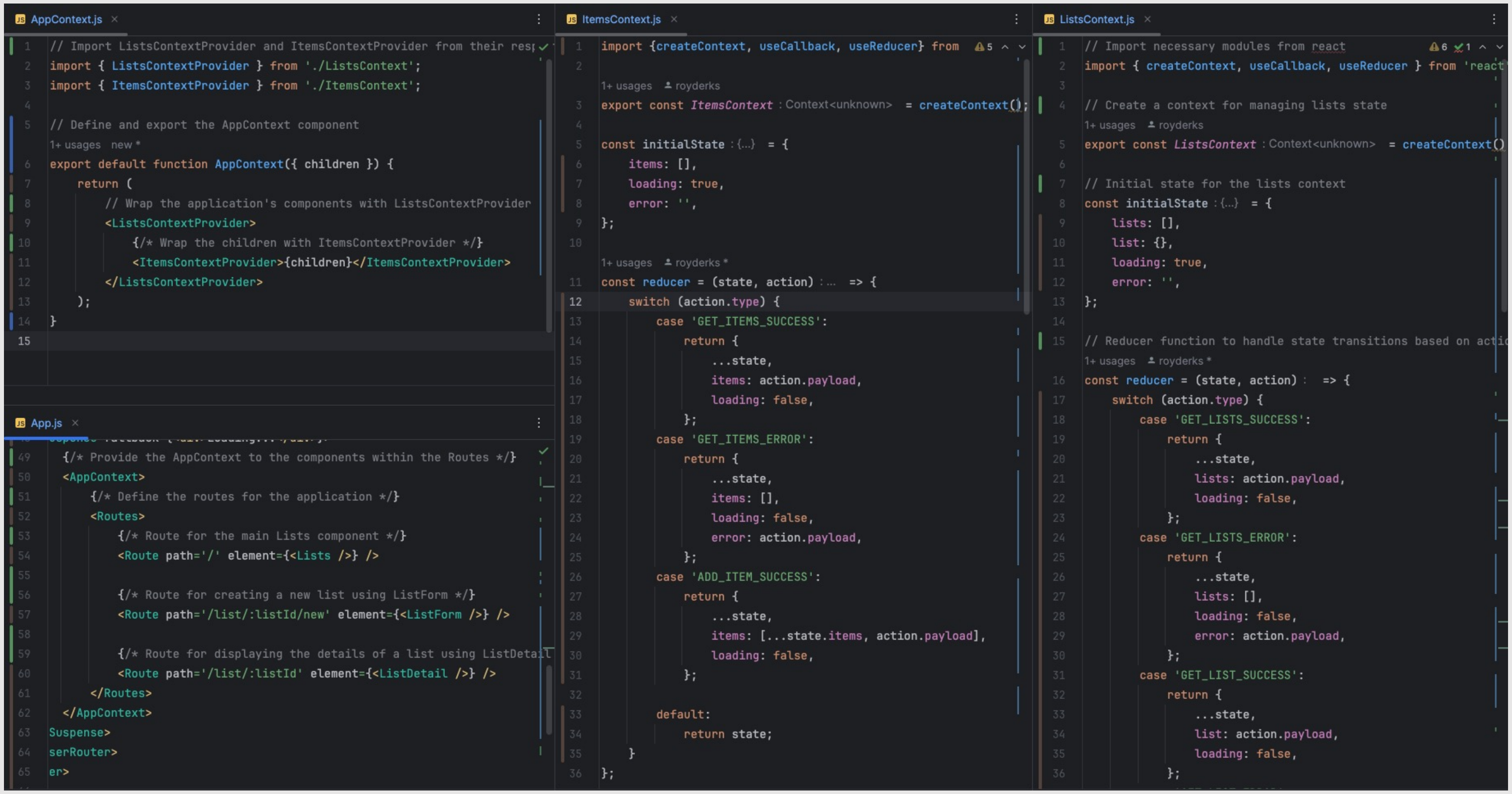
```jsx
import { createContext, useContext } from "react";

const UserContext : Context<unknown>  = createContext();

1+ usages
export default function Component1() {

    const user : string = "Jesse Hall";

    return (
        <UserContext.Provider value={user}>
            <h1>{`Hello ${user}!`}</h1>
            <Component2 />
        </UserContext.Provider>
    );
}

1+ usages
function Component2(){return (<> <h1>Component 2</h1> <Component3 /> </>);}

1+ usages
function Component3() {return (<><h1>Component 3</h1><Component4 /></>);}

1+ usages
function Component4() {return (<><h1>Component 4</h1><Component5 /></>);}

1+ usages
function Component5() {
    const user = useContext(UserContext);

    return (
        <>
            <h1>Component 5</h1>
            <h2>{`Hello ${user} again!`}</h2>
        </>
    );
}
```

localhost:5173

```
Hello Jesse Hall!

Component 2

Component 3

Component 4

Component 5

Hello Jesse Hall again!
```

Props/Context

Another Example



AppContext.js

```
// Import ListsContextProvider and ItemsContextProvider from their resp
import { ListsContextProvider } from './ListsContext';
import { ItemsContextProvider } from './ItemsContext';

// Define and export the AppContext component
1+ usages  new *
export default function AppContext({ children }) {
    return (
        // Wrap the application's components with ListsContextProvider
        <ListsContextProvider>
            {/* Wrap the children with ItemsContextProvider */}
            <ItemsContextProvider>{children}</ItemsContextProvider>
        </ListsContextProvider>
    );
}
```

App.js

```
  {/* Provide the AppContext to the components within the Routes */}
  <AppContext>
      {/* Define the routes for the application */}
      <Routes>
          {/* Route for the main Lists component */}
          <Route path='/' element={<Lists />} />

          {/* Route for creating a new list using ListForm */}
          <Route path='/list/:listId/new' element={<ListForm />} />

          {/* Route for displaying the details of a list using ListDetail
          <Route path='/list/:listId' element={<ListDetail />} />
      </Routes>
  </AppContext>
Suspense>
serRouter>
er>
```

ItemsContext.js

```
import {createContext, useCallback, useReducer} from

1+ usages  royderks
export const ItemsContext : Context<unknown>  = createContext();

const initialState : {...}  = {
    items: [],
    loading: true,
    error: '',
};

1+ usages  royderks *
const reducer = (state, action) : ...  => {
    switch (action.type) {
        case 'GET_ITEMS_SUCCESS':
            return {
                ...state,
                items: action.payload,
                loading: false,
            };
        case 'GET_ITEMS_ERROR':
            return {
                ...state,
                items: [],
                loading: false,
                error: action.payload,
            };
        case 'ADD_ITEM_SUCCESS':
            return {
                ...state,
                items: [...state.items, action.payload],
                loading: false,
            };

        default:
            return state;
    }
};
```

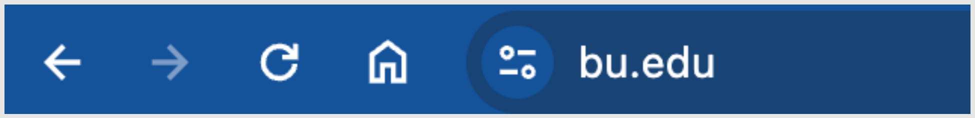
ListsContext.js

```
// Import necessary modules from react
import { createContext, useCallback, useReducer } from 'react'

// Create a context for managing lists state
1+ usages  royderks
export const ListsContext : Context<unknown>  = createContext()

// Initial state for the lists context
const initialState : {...}  = {
    lists: [],
    list: {},
    loading: true,
    error: '',
};

// Reducer function to handle state transitions based on actio
1+ usages  royderks *
const reducer = (state, action) :  => {
    switch (action.type) {
        case 'GET_LISTS_SUCCESS':
            return {
                ...state,
                lists: action.payload,
                loading: false,
            };
        case 'GET_LISTS_ERROR':
            return {
                ...state,
                lists: [],
                loading: false,
                error: action.payload,
            };
        case 'GET_LIST_SUCCESS':
            return {
                ...state,
                list: action.payload,
                loading: false,
            };
```

Routing

Navigation in React

In HTML, the **HTML-DOM** kept track of your rendered HTML pages when ever you clicked on a link. So all you had to do was to create your navigation menu, and your anchor tags were managed by the DOM.



But in React you are creating **Apps**, and apps may run on variety of different devices which means the end-user may not necessary use a web-browser to access your app.

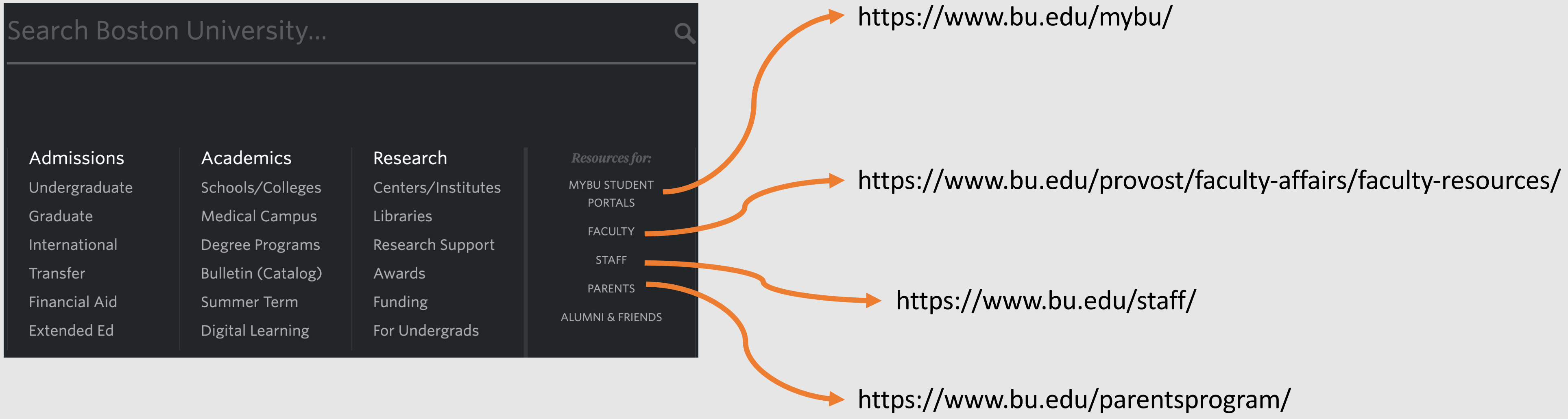
Routing

So far we have created single-page app, now we will focus on multi-page apps

When we say "multi-page" though, we are not referring to having multiple HTML pages, instead we are going to re-render components in one page, but pretend as if we were doing so by switching between multiple pages.

But why do we have to pretend?

Legacy URLs & History (back-button)



https://www.bu.edu/mybu/

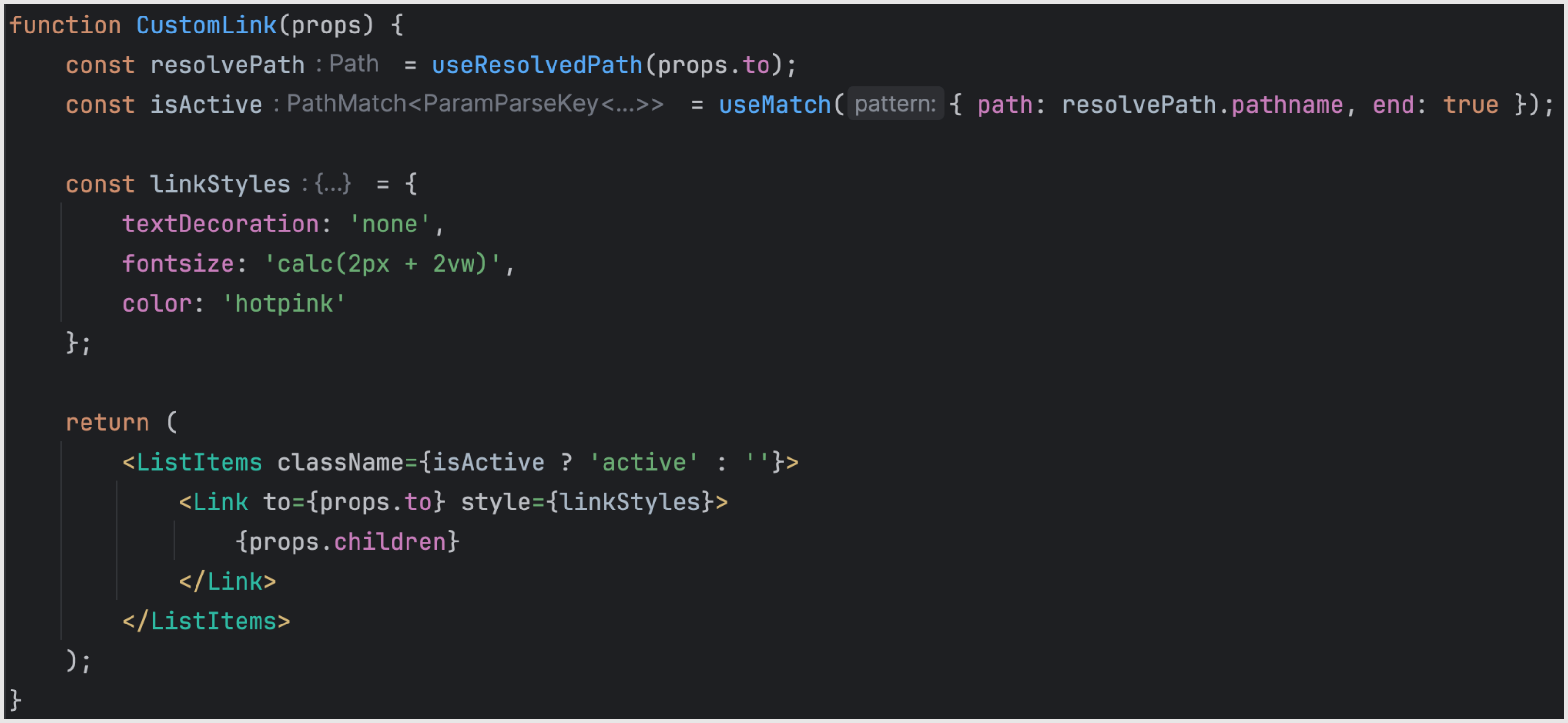
https://www.bu.edu/provost/faculty-affairs/faculty-resources/

https://www.bu.edu/staff/

https://www.bu.edu/parentsprogram/

Routing

useResolvePath and **useMatch,** often used together, are React-Router Hooks. **useResolvePath** will simply the process of generating URLs or paths for different routes in your React application, and **useMatch** hook simplifies the process of checking if the current URL matches a specific route in your React application. It abstracts away the URL matching logic, making it easier to conditionally render components or perform actions based on the current route. These two Hooks are often used for legacy URLs.



```
function CustomLink(props) {
    const resolvePath : Path  = useResolvedPath(props.to);
    const isActive : PathMatch<ParamParseKey<...>>  = useMatch( pattern: { path: resolvePath.pathname, end: true });

    const linkStyles : {...}  = {
        textDecoration: 'none',
        fontsize: 'calc(2px + 2vw)',
        color: 'hotpink'
    };

    return (
        <ListItems className={isActive ? 'active' : ''}>
            <Link to={props.to} style={linkStyles}>
                {props.children}
            </Link>
        </ListItems>
    );
}
```

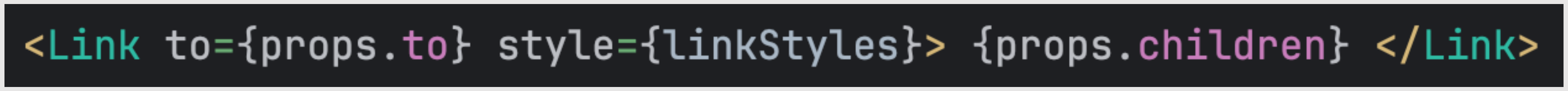
Routing

Navigation in React

In **React** the implementation of the **<nav>** tag, in some ways, is very similar to HTML. We continue to use **<nav>**, **<ul>**, and **<li>**, **but** we won't use an **<a>**

<Link></Link>

Anchor tags were used to render and entire HTML page, but in **React** we have only use one HTML page (index.html). So, we don't need to re-render the entire page when components are being recycled. **Link** will render components in and out of focus, when ever we are done using them.



```
<Link to={props.to} style={linkStyles}> {props.children} </Link>
```

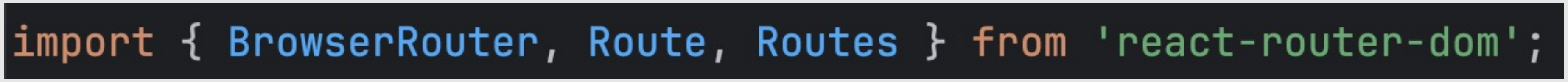
Instead of the **href=""** attribute, we will use **to={}**, in the **Link** component to specify which components should be rendered

Note:
When you want to create an instance-variable, without knowing what/how you will initialize it later, you can use **props.children** as a "place holder".

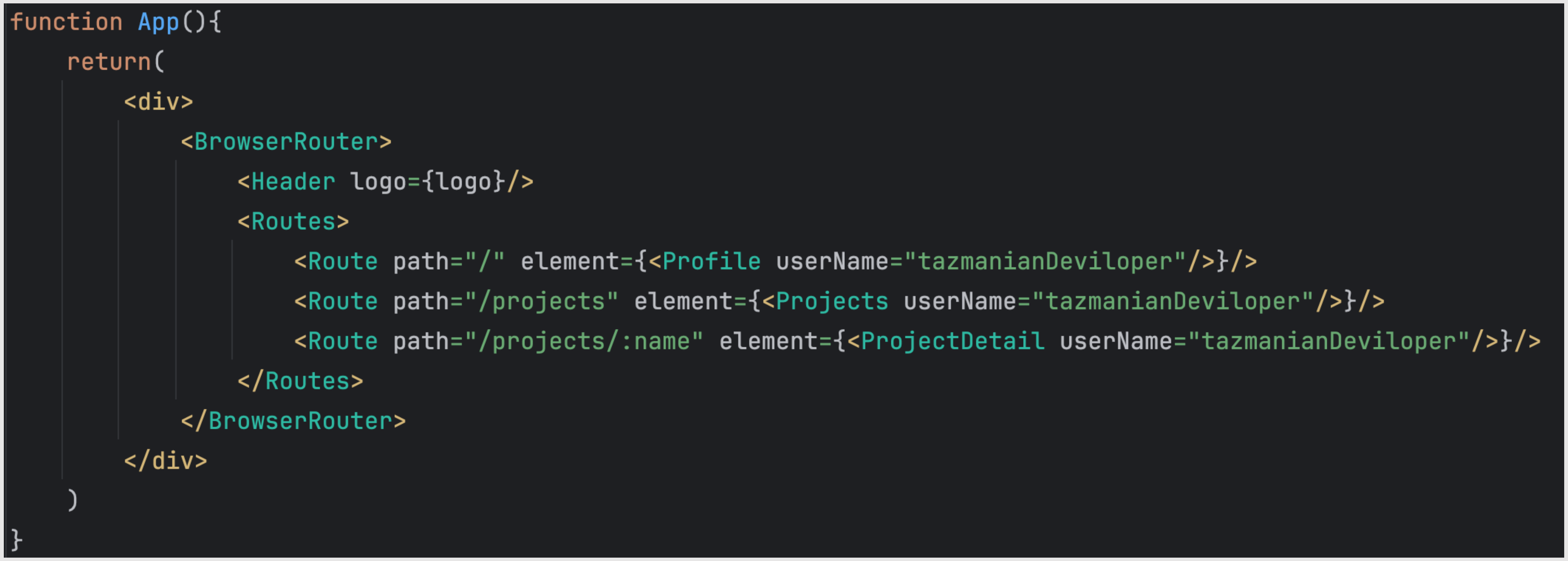
Routing

Navigation in React

So, If you are building a **web-app**, then you have to also allow the end-users to access browser's tool



```
import { BrowserRouter, Route, Routes } from 'react-router-dom';
```



```
function App(){
    return(
        <div>
            <BrowserRouter>
                <Header logo={logo}/>
                <Routes>
                    <Route path="/" element={<Profile userName="tazmanianDeviloper"/>}/>
                    <Route path="/projects" element={<Projects userName="tazmanianDeviloper"/>}/>
                    <Route path="/projects/:name" element={<ProjectDetail userName="tazmanianDeviloper"/>}/>
                </Routes>
            </BrowserRouter>
        </div>
    )
}
```

Routing

Navigation in React

So to have a functional Navigation Menu in a Web-App you need **2 Components**.

Your **Nav** Component & The **Browser-Router**

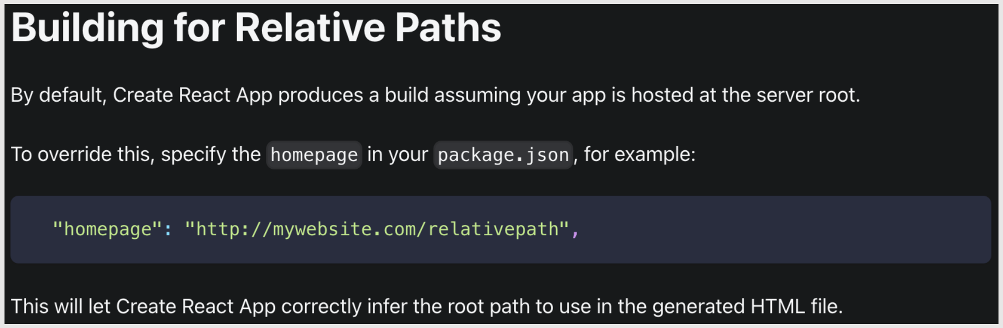
```
function Navigation(){
   return (
      <>
         <nav>
            <ul>
               <li> <Link to="/">Home</Link> </li>
               <li> <Link to="/blogs">Blogs</Link> </li>
               <li> <Link to="/contact">Contact</Link> </li>
            </ul>
         </nav>
      </>
   )
}
```

```
export default function App() {
   return (
      <BrowserRouter>
         <Routes>
            <Route>
               <Route path="/" element={<Home />} />
               <Route path="/blogs" element={<Blogs />} />
               <Route path="/contact" element={<Contact />} />
            </Route>
         </Routes>
      </BrowserRouter>
   );
}
```

Routing

Navigation in React

If your app is not the only app/project that is hosted in a server, then you must specify the **root** directory. Otherwise your **navigation menu** won't work properly.



Building for Relative Paths

By default, Create React App produces a build assuming your app is hosted at the server root.

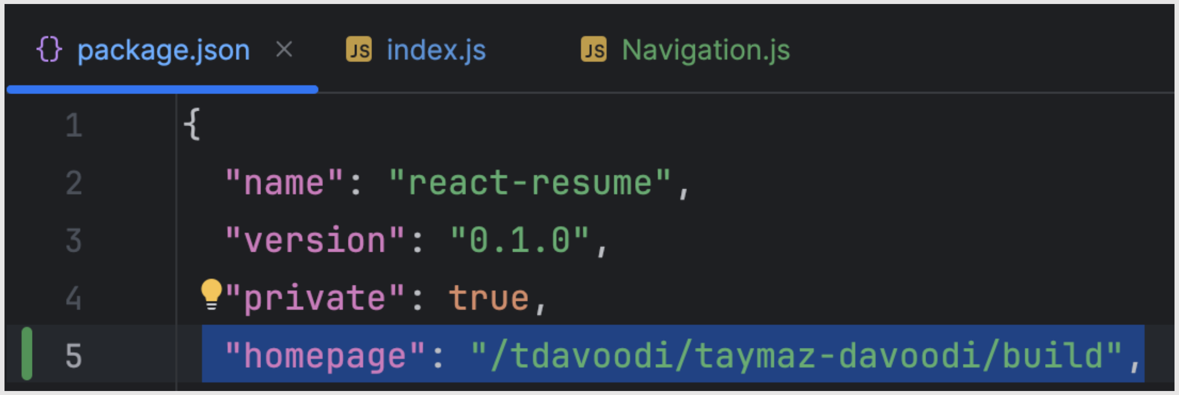
To override this, specify the `homepage` in your `package.json`, for example:

```
"homepage": "http://mywebsite.com/relativepath",
```

This will let Create React App correctly infer the root path to use in the generated HTML file.

Note:

BU's Server, is shared with other CS courses, and instructors, so the conditions mentioned above does apply to your projects. This means, going forward, for every mini-project, you have to specify a "**homepage**" in the **package.json** file.



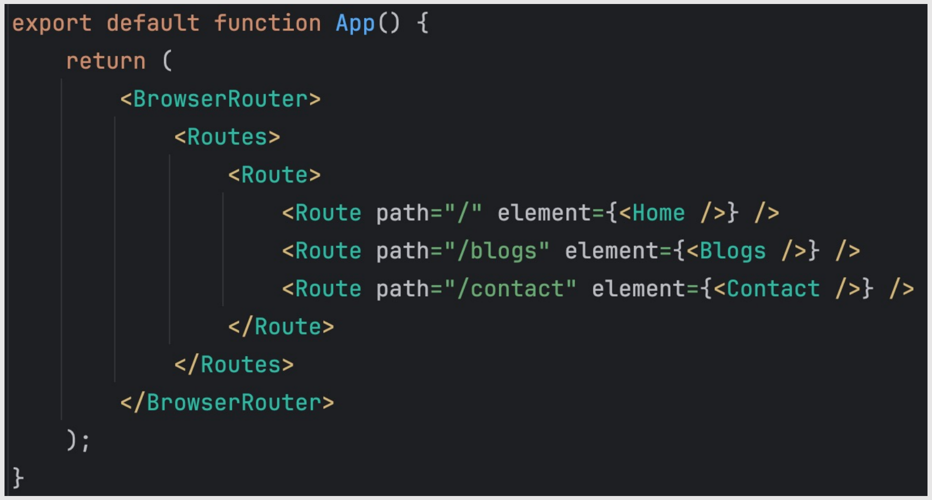
```
{
  "name": "react-resume",
  "version": "0.1.0",
  "private": true,
  "homepage": "/tdavoodi/taymaz-davoodi/build",
```

Routing

Navigation in React

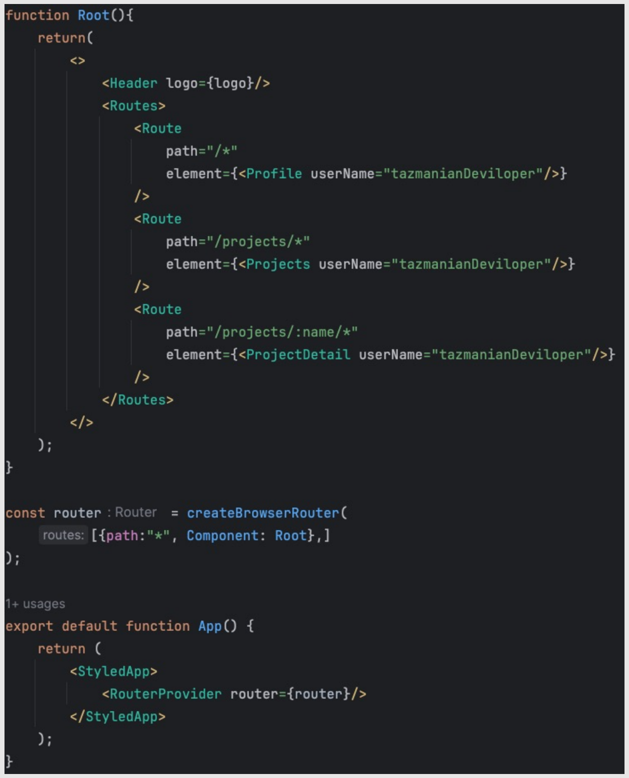
As of version 6.4.0 the **React-Router** has had many updates. These updates has made **routing** much more efficient, but has also changed the syntax a little. The new syntax is referred to as **RouterProvider** and it looks like this:

The **Browser-Router** (old)



```
export default function App() {
    return (
        <BrowserRouter>
            <Routes>
                <Route>
                    <Route path="/" element={<Home />} />
                    <Route path="/blogs" element={<Blogs />} />
                    <Route path="/contact" element={<Contact />} />
                </Route>
            </Routes>
        </BrowserRouter>
    );
}
```

The **RouterProvider** (new)



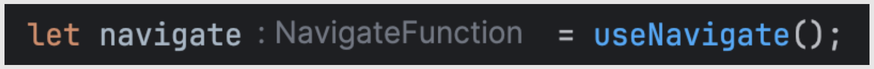
```
function Root(){
    return(
        <>
            <Header logo={logo}/>
            <Routes>
                <Route
                    path="/*"
                    element={<Profile userName="tazmanianDeviloper"/>}
                />
                <Route
                    path="/projects/*"
                    element={<Projects userName="tazmanianDeviloper"/>}
                />
                <Route
                    path="/projects/:name/*"
                    element={<ProjectDetail userName="tazmanianDeviloper"/>}
                />
            </Routes>
        </>
    );
}

const router : Router  = createBrowserRouter(
    routes: [{path:"*", Component: Root},]
);

1+ usages
export default function App() {
    return (
        <StyledApp>
            <RouterProvider router={router}/>
        </StyledApp>
    );
}
```

Additional recourses to help with migration to **RouterProvider**:

https://reactrouter.com/en/main/upgrading/v6-data
https://youtu.be/oTIJunBa6MA?si=Ffbxii0lXnRqXH34

Routing

New Hooks



```
let navigate : NavigateFunction  = useNavigate();
```

useNavigate() is a **React Router** hook that allows you to programmatically navigate to different routes.

Note:

- **useNavigate()** does not replace the <BroswerRouter> <Routes> <Route> formation.
- In **Chapter-5** it is being used as a navigation tool, instead of the conventional <nav>, but that is **not** the right use for it either.

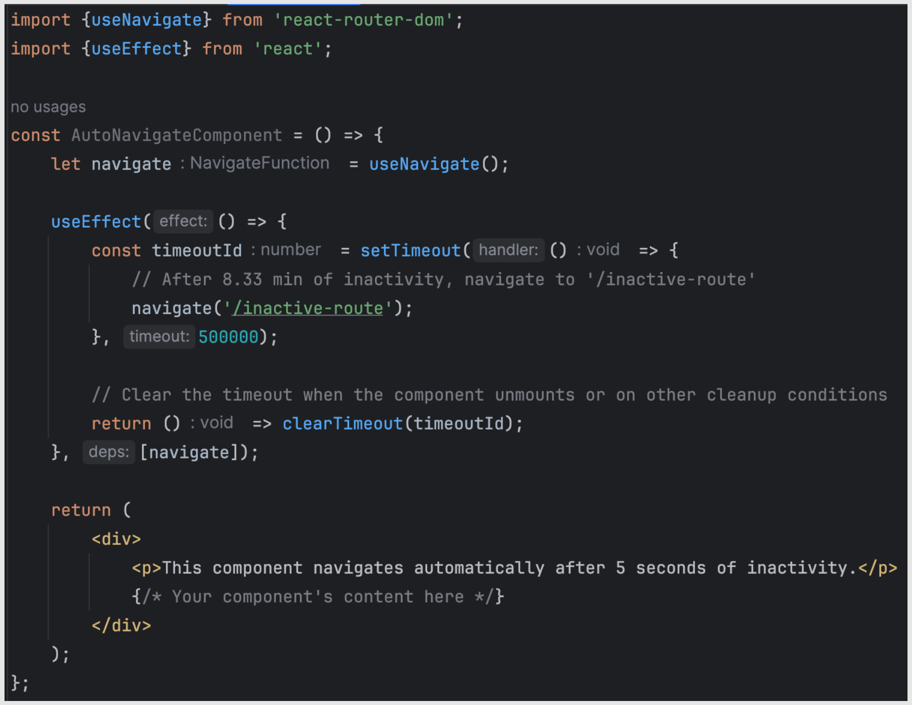
useNavigate() should be used for instances where user interaction is not needed for navigation

Ex:

- You have most likely experienced being redirected from a website/app when you did not interact with it for a while, like the BU's blackboard.

Question:

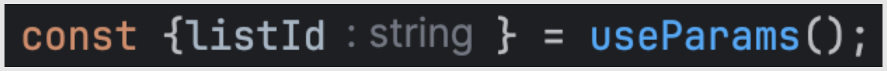
- Why is the book using let instead of const for the navigate variable?



```
import {useNavigate} from 'react-router-dom';
import {useEffect} from 'react';

no usages
const AutoNavigateComponent = () => {
    let navigate : NavigateFunction  = useNavigate();

    useEffect( effect: () => {
        const timeoutId : number  = setTimeout( handler: () : void  => {
            // After 8.33 min of inactivity, navigate to '/inactive-route'
            navigate('/inactive-route');
        },  timeout: 500000);

        // Clear the timeout when the component unmounts or on other cleanup conditions
        return () : void  => clearTimeout(timeoutId);
    },  deps: [navigate]);

    return (
        <div>
            <p>This component navigates automatically after 5 seconds of inactivity.</p>
            {/* Your component's content here */}
        </div>
    );
};
```

Routing

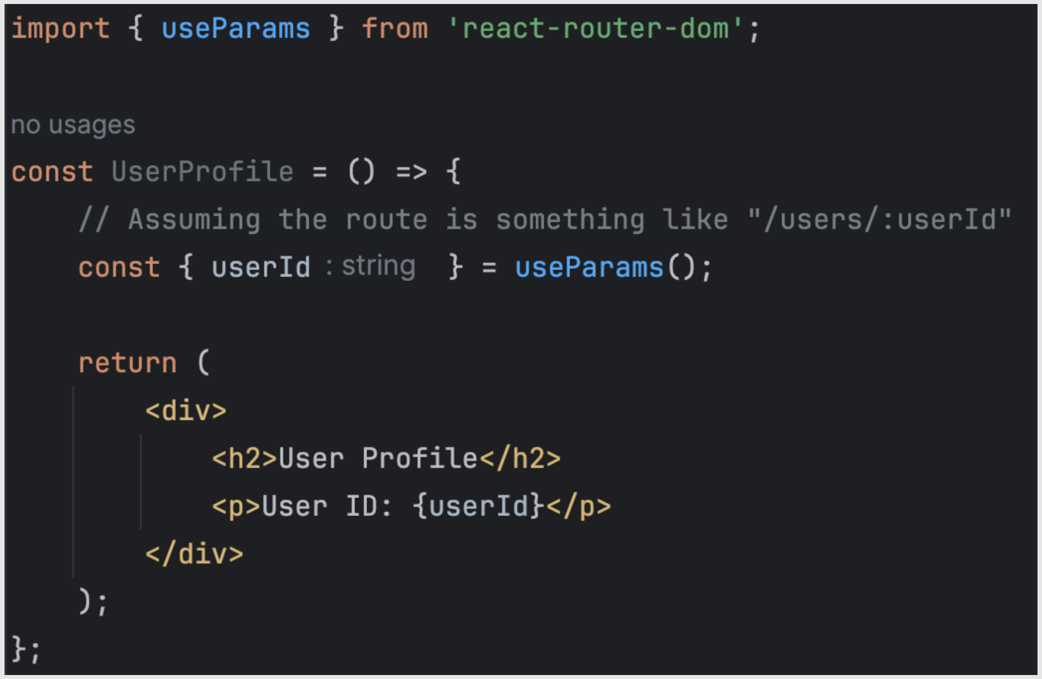
New Hooks



```
const {listId : string } = useParams();
```

useParam() is another **React Router** hook that allows you to access the parameters (variables) of the current route

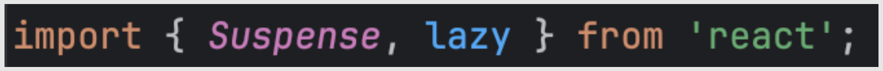
- In a typical web application, you might have URLs with dynamic parts, like **/users/123** where 123 is a user ID.
- With **useParams()** you can access the value of these dynamic parts directly from your component.



```
import { useParams } from 'react-router-dom';

no usages
const UserProfile = () => {
    // Assuming the route is something like "/users/:userId"
    const { userId : string  } = useParams();

    return (
        <div>
            <h2>User Profile</h2>
            <p>User ID: {userId}</p>
        </div>
    );
};
```

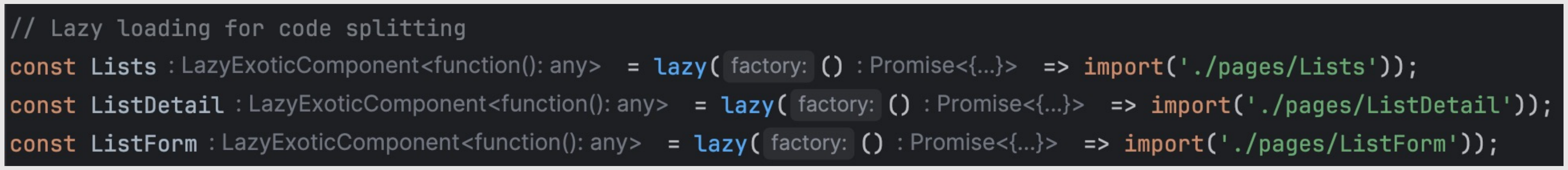
Routing

lazy/suspense



```
import { Suspense, lazy } from 'react';
```

Lazy imports, also known as dynamic imports, allow you to conditionally load a module or component only when it is needed



```
// Lazy loading for code splitting
const Lists : LazyExoticComponent<function(): any>  = lazy( factory: () : Promise<{...}>  => import('./pages/Lists'));
const ListDetail : LazyExoticComponent<function(): any>  = lazy( factory: () : Promise<{...}>  => import('./pages/ListDetail'));
const ListForm : LazyExoticComponent<function(): any>  = lazy( factory: () : Promise<{...}>  => import('./pages/ListForm'));
```

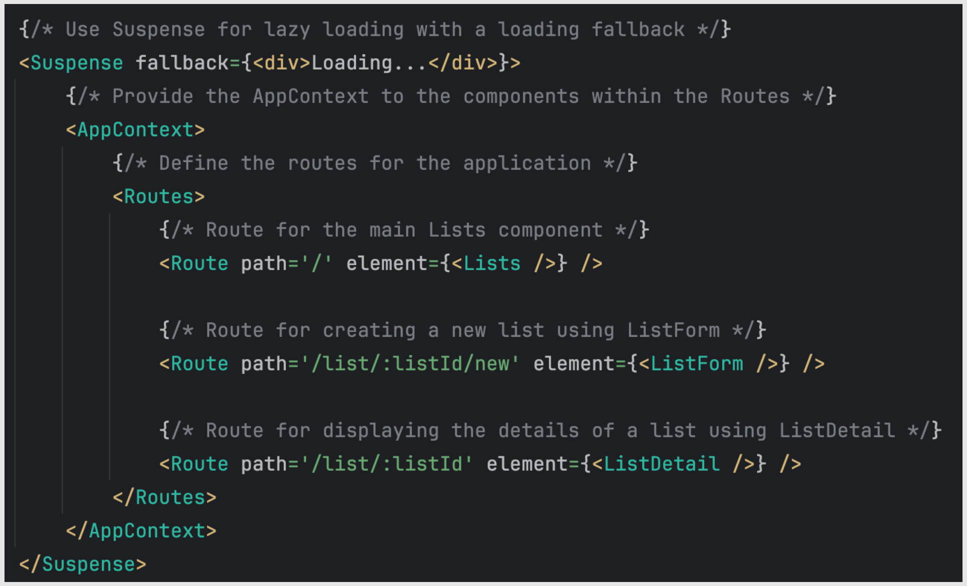
Benefits:

- **Reduced Initial Bundle Size.**
- **Improved Initial Load Time.**
- **Efficient Resource Usage.**

When **lazily** loaded components carry potentially lengthy data or **fetching** schema, a fallback is needed to handle the delay. This is why **lazy imports** are commonly paired with the **Suspense component**.

Suspense acts as a boundary, allowing components to pause rendering until certain tasks, like loading lazy components, are completed.

Suspense is not only for **lazy** loading; it can also be used to handle asynchronous data fetching, by allowing components to wait for the data to be fetched before rendering.



```
{/* Use Suspense for lazy loading with a loading fallback */}
<Suspense fallback={<div>Loading...</div>}>
    {/* Provide the AppContext to the components within the Routes */}
    <AppContext>
        {/* Define the routes for the application */}
        <Routes>
            {/* Route for the main Lists component */}
            <Route path='/' element={<Lists />} />

            {/* Route for creating a new list using ListForm */}
            <Route path='/list/:listId/new' element={<ListForm />} />

            {/* Route for displaying the details of a list using ListDetail */}
            <Route path='/list/:listId' element={<ListDetail />} />
        </Routes>
    </AppContext>
</Suspense>
```