# React.js Notes – Part 3

Contents

[React.js Notes – Part 3 1](#_Toc126683610)

[Section 9: Diving Deeper: Working with Fragments, Portals, & "Refs" 1](#_Toc126683611)

[99. Module Introduction 1](#_Toc126683612)

[100. JSX Limitations & Workarounds 1](#_Toc126683613)

[101. Creating a Wrapper Component 8](#_Toc126683614)

[102. React Fragments 9](#_Toc126683615)

## Section 9: Diving Deeper: Working with Fragments, Portals, & "Refs"

### 99. Module Introduction

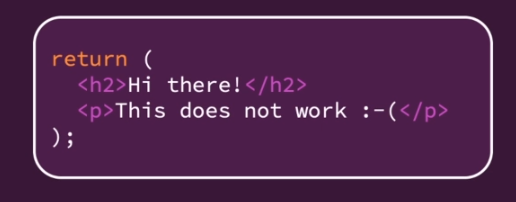
We will look at limitations faced with JSX code and how Fragments can help us overcome those limitations. We'll look at getting a cleaner DOM, better code structure with Portals. We will also look at working with Refs.

### 100. JSX Limitations & Workarounds

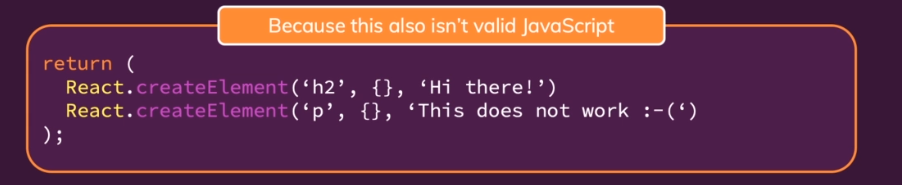
JSX is the code that you return from your components that will in the end be rendered by the real DOM in React.

JSX has limitations.

One limitation is that if we have adjacent root level JSX elements like below, we'll get an error. In React you can't return more than one "root" JSX element (you also can't store more than one "root" JSX element in a variable.

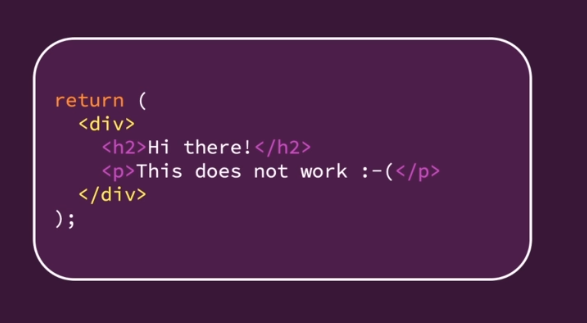


Children elements can be adjacent, but the topmost element can only be one element. This makes sense because in JavaScript we can't return more than one thing.



In the AddUser component we have a wrapping div, and that is a workaround.

|  |
| --- |
| src/components/Users/AddUser.js |
| import React, { useState } from 'react';  import Card from '../UI/Card';  import Button from '../UI/Button';  import ErrorModal from '../UI/ErrorModal';  import classes from './AddUser.module.css';  const AddUser = (props) => {  const [enteredUsername, setEnteredUsername] = useState('');  const [enteredAge, setEnteredAge] = useState('');  const [error, setError] = useState();  const addUserHandler = (event) => {  event.preventDefault();  if (enteredUsername.trim().length === 0 || enteredAge.trim().length === 0) {  setError({  title: 'Invalid input',  message: 'Please enter a valid name and age (non-empty values).',  });  return;  }  if (+enteredAge < 1) {  setError({  title: 'Invalid age',  message: 'Please enter a valid age (> 0).',  });  return;  }  props.onAddUser(enteredUsername, enteredAge);  setEnteredUsername('');  setEnteredAge('');  };  const usernameChangeHandler = (event) => {  setEnteredUsername(event.target.value);  };  const ageChangeHandler = (event) => {  setEnteredAge(event.target.value);  };  const errorHandler = () => {  setError(null);  };  return (  <div>  {error && (  <ErrorModal  title={error.title}  message={error.message}  onConfirm={errorHandler}  />  )}  <Card className={classes.input}>  <form onSubmit={addUserHandler}>  <label htmlFor="username">Username</label>  <input  id="username"  type="text"  value={enteredUsername}  onChange={usernameChangeHandler}  />  <label htmlFor="age">Age (Years)</label>  <input  id="age"  type="number"  value={enteredAge}  onChange={ageChangeHandler}  />  <Button type="submit">Add User</Button>  </form>  </Card>  </div>  );  };  export default AddUser; |

You can solve the problem of returning more than one root element by wrapping them in a div.   


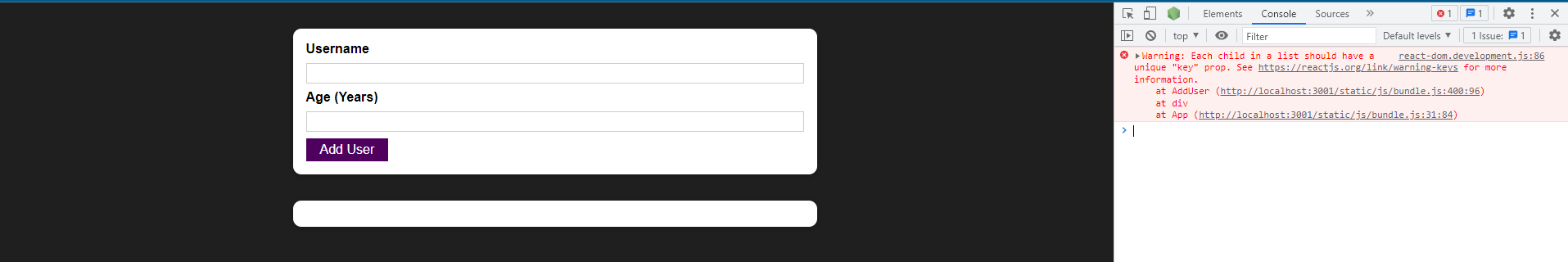
If you wrap the elements in a div, then you are only returning one element. It's similar to returning three numbers by putting them into an array.

**Important:** wrapping element does not have to be a div – ANY element will do the trip.

You could also use a JS array. We could remove our opening and closing div tags and replace with opening and closing square brackets. We would also need to separate the elements with commas and remove the curly braces around the error and ErrorModal because we are no longer inside of JSX here; we are inside of an array.

|  |
| --- |
| src/components/Users/AddUser.js |
| import React, { useState } from 'react';  import Card from '../UI/Card';  import Button from '../UI/Button';  import ErrorModal from '../UI/ErrorModal';  import classes from './AddUser.module.css';  const AddUser = (props) => {  const [enteredUsername, setEnteredUsername] = useState('');  const [enteredAge, setEnteredAge] = useState('');  const [error, setError] = useState();  const addUserHandler = (event) => {  event.preventDefault();  if (enteredUsername.trim().length === 0 || enteredAge.trim().length === 0) {  setError({  title: 'Invalid input',  message: 'Please enter a valid name and age (non-empty values).',  });  return;  }  if (+enteredAge < 1) {  setError({  title: 'Invalid age',  message: 'Please enter a valid age (> 0).',  });  return;  }  props.onAddUser(enteredUsername, enteredAge);  setEnteredUsername('');  setEnteredAge('');  };  const usernameChangeHandler = (event) => {  setEnteredUsername(event.target.value);  };  const ageChangeHandler = (event) => {  setEnteredAge(event.target.value);  };  const errorHandler = () => {  setError(null);  };  return (  **[**  error && (  <ErrorModal  title={error.title}  message={error.message}  onConfirm={errorHandler}  />  )**,**  <Card className={classes.input}>  <form onSubmit={addUserHandler}>  <label htmlFor="username">Username</label>  <input  id="username"  type="text"  value={enteredUsername}  onChange={usernameChangeHandler}  />  <label htmlFor="age">Age (Years)</label>  <input  id="age"  type="number"  value={enteredAge}  onChange={ageChangeHandler}  />  <Button type="submit">Add User</Button>  </form>  </Card>  **]**  );  };  export default AddUser; |

We can return an array because React is able to work with arrays of JSX elements. However, we do get a warning in Chrome's developer tools console:



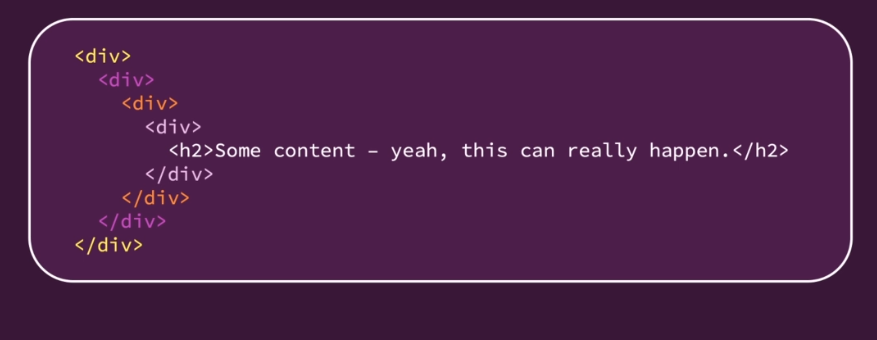
Because we're working with an array of JSX elements, React wants a key on every element. We can simply add a key prop with a custom value to each of the elements.

|  |
| --- |
| src/components/Users/AddUser.js |
| import React, { useState } from 'react';  import Card from '../UI/Card';  import Button from '../UI/Button';  import ErrorModal from '../UI/ErrorModal';  import classes from './AddUser.module.css';  const AddUser = (props) => {  const [enteredUsername, setEnteredUsername] = useState('');  const [enteredAge, setEnteredAge] = useState('');  const [error, setError] = useState();  const addUserHandler = (event) => {  event.preventDefault();  if (enteredUsername.trim().length === 0 || enteredAge.trim().length === 0) {  setError({  title: 'Invalid input',  message: 'Please enter a valid name and age (non-empty values).',  });  return;  }  if (+enteredAge < 1) {  setError({  title: 'Invalid age',  message: 'Please enter a valid age (> 0).',  });  return;  }  props.onAddUser(enteredUsername, enteredAge);  setEnteredUsername('');  setEnteredAge('');  };  const usernameChangeHandler = (event) => {  setEnteredUsername(event.target.value);  };  const ageChangeHandler = (event) => {  setEnteredAge(event.target.value);  };  const errorHandler = () => {  setError(null);  };  return (  [  error && (  <ErrorModal  **key="error-modal"**  title={error.title}  message={error.message}  onConfirm={errorHandler}  />  ),  <Card **key="add-user-card"** className={classes.input}>  <form onSubmit={addUserHandler}>  <label htmlFor="username">Username</label>  <input  id="username"  type="text"  value={enteredUsername}  onChange={usernameChangeHandler}  />  <label htmlFor="age">Age (Years)</label>  <input  id="age"  type="number"  value={enteredAge}  onChange={ageChangeHandler}  />  <Button type="submit">Add User</Button>  </form>  </Card>  ]  );  };  export default AddUser; |

You typically don't use this solution because adding those keys and wrapping it in an array is a little bit cumbersome. It's way easier to simply wrap it in a div like before.

|  |
| --- |
| src/components/Users/AddUser.js |
| import React, { useState } from 'react';  import Card from '../UI/Card';  import Button from '../UI/Button';  import ErrorModal from '../UI/ErrorModal';  import classes from './AddUser.module.css';  const AddUser = (props) => {  const [enteredUsername, setEnteredUsername] = useState('');  const [enteredAge, setEnteredAge] = useState('');  const [error, setError] = useState();  const addUserHandler = (event) => {  event.preventDefault();  if (enteredUsername.trim().length === 0 || enteredAge.trim().length === 0) {  setError({  title: 'Invalid input',  message: 'Please enter a valid name and age (non-empty values).',  });  return;  }  if (+enteredAge < 1) {  setError({  title: 'Invalid age',  message: 'Please enter a valid age (> 0).',  });  return;  }  props.onAddUser(enteredUsername, enteredAge);  setEnteredUsername('');  setEnteredAge('');  };  const usernameChangeHandler = (event) => {  setEnteredUsername(event.target.value);  };  const ageChangeHandler = (event) => {  setEnteredAge(event.target.value);  };  const errorHandler = () => {  setError(null);  };  return (  **<div>**  {error && (  <ErrorModal  title={error.title}  message={error.message}  onConfirm={errorHandler}  />  )}  <Card className={classes.input}>  <form onSubmit={addUserHandler}>  <label htmlFor="username">Username</label>  <input  id="username"  type="text"  value={enteredUsername}  onChange={usernameChangeHandler}  />  <label htmlFor="age">Age (Years)</label>  <input  id="age"  type="number"  value={enteredAge}  onChange={ageChangeHandler}  />  <Button type="submit">Add User</Button>  </form>  </Card>  **</div>**  );  };  export default AddUser; |

With the wrapping div or any other wrapping element, a new problem arises. Now we can end up with "<div> Soup" – where you have a real DOM rendered with many nested components and all of those components for various reasons need wrapping divs or other wrapping components, and you have all of these unnecessary divs being rendered in the real DOM even though they're only there because of this limitation in JSX.



In bigger apps, you can easily end up with **tons of unnecessary <div>**s or other elements which add **no semantic meaning or structure** to the page but **are only there because of React's JSX' requirement**.

### 101. Creating a Wrapper Component

We will add a new subfolder in our components folder called "Helpers". We add a Wrapper component. In there we won't import React because we are not going to write any JSX code. Instead we will do this:

|  |
| --- |
| src/components/Helpers/Wrapper.js |
| const Wrapper = props => {  return props.children;  };  export default Wrapper; |

props.children holds all of the content that you are passing between the opening and closing tag of your custom component.

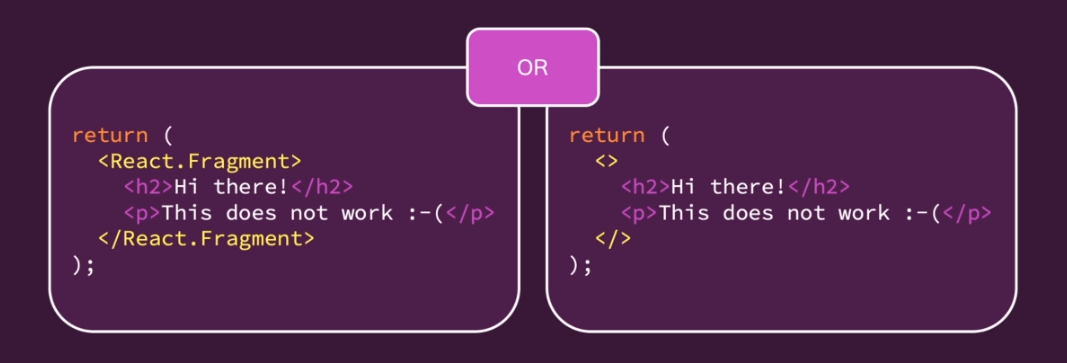
In AddUser we then use our Wrapper by replacing our divs with Wrapper.

|  |
| --- |
| src/components/Users/AddUser.js |
| import React, { useState } from 'react';  import Card from '../UI/Card';  import Button from '../UI/Button';  import ErrorModal from '../UI/ErrorModal';  import Wrapper from '../Helpers/Wrapper';  import classes from './AddUser.module.css';  const AddUser = (props) => {  const [enteredUsername, setEnteredUsername] = useState('');  const [enteredAge, setEnteredAge] = useState('');  const [error, setError] = useState();  const addUserHandler = (event) => {  event.preventDefault();  if (enteredUsername.trim().length === 0 || enteredAge.trim().length === 0) {  setError({  title: 'Invalid input',  message: 'Please enter a valid name and age (non-empty values).',  });  return;  }  if (+enteredAge < 1) {  setError({  title: 'Invalid age',  message: 'Please enter a valid age (> 0).',  });  return;  }  props.onAddUser(enteredUsername, enteredAge);  setEnteredUsername('');  setEnteredAge('');  };  const usernameChangeHandler = (event) => {  setEnteredUsername(event.target.value);  };  const ageChangeHandler = (event) => {  setEnteredAge(event.target.value);  };  const errorHandler = () => {  setError(null);  };  return (  **<Wrapper>**  {error && (  <ErrorModal  title={error.title}  message={error.message}  onConfirm={errorHandler}  />  )}  <Card className={classes.input}>  <form onSubmit={addUserHandler}>  <label htmlFor="username">Username</label>  <input  id="username"  type="text"  value={enteredUsername}  onChange={usernameChangeHandler}  />  <label htmlFor="age">Age (Years)</label>  <input  id="age"  type="number"  value={enteredAge}  onChange={ageChangeHandler}  />  <Button type="submit">Add User</Button>  </form>  </Card>  **</Wrapper>**  );  };  export default AddUser; |

The Wrapper is basically an empty component. Everything it does is that it returns props.children.

If we save AddUser, our application works just fine because in AddUser we have a wrapping element. It's a component that won't render anything to the DOM.

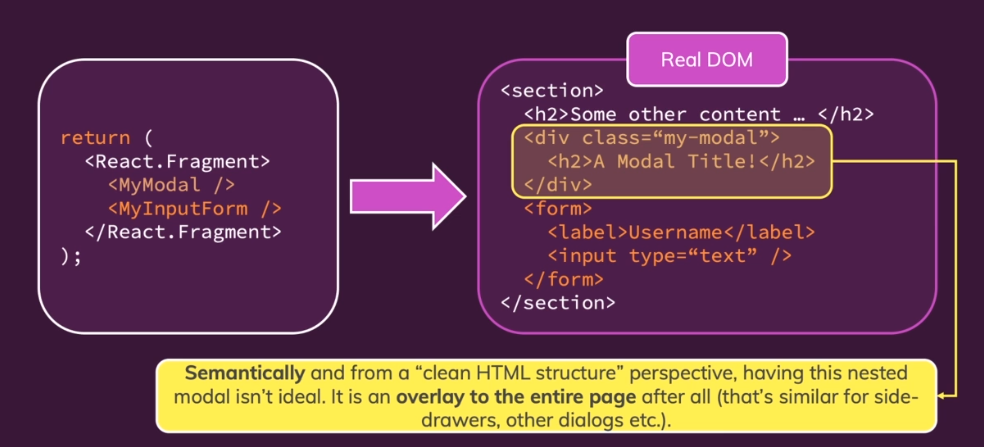
### 102. React Fragments

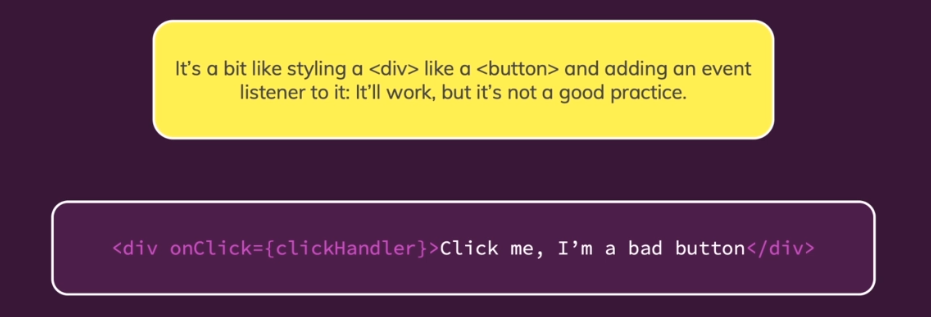
The Wrapper component that we built is not one we need to build because it comes with React. In React it is called the fragment component. You can always use the Fragmenet setup on the left in the screenshot but using the Fragment setup on the right depends on your project setup. 

A fragment is an empty wrapper component that doesn't render any real HTML element to the DOM but fulfills React's JSX requirement. We can use the built in wraper like so in the App component:

|  |
| --- |
| src/App.js |
| import React, { useState } from 'react';  import AddUser from './components/Users/AddUser';  import UsersList from './components/Users/UsersList';  function App() {  const [usersList, setUsersList] = useState([]);  const addUserHandler = (uName, uAge) => {  setUsersList((prevUsersList) => {  return [  ...prevUsersList,  { name: uName, age: uAge, id: Math.random().toString() },  ];  });  };  return (  **<>**  <AddUser onAddUser={addUserHandler} />  <UsersList users={usersList} />  **</>**  );  }  export default App; |

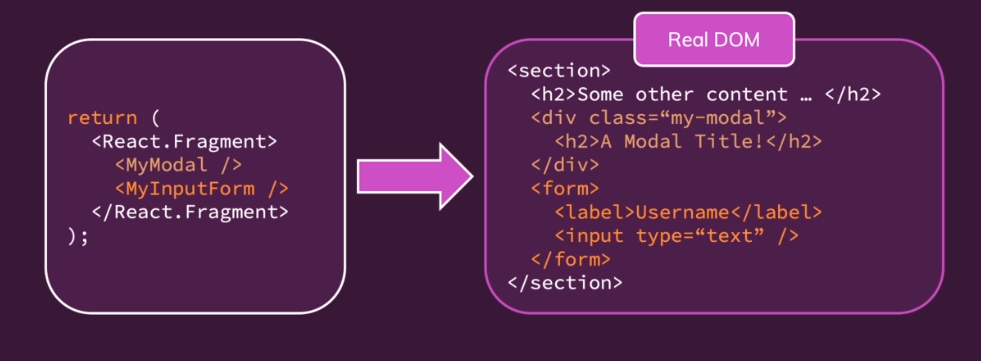
### 103. Introducing React Portals

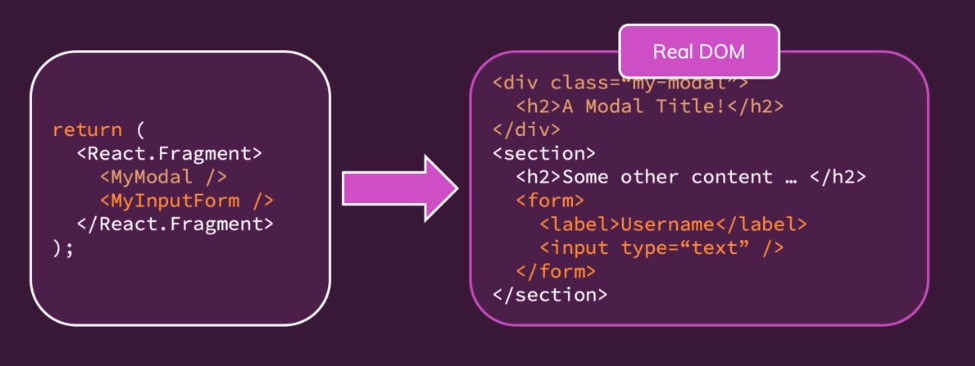




Styling a div like a button is a bad practice for accessibility, bad if a fellow developer has to work on it, and it's just a bad practice in general.

We can use a portal to keep the structure we have on the left in the screenshot below. We want to still be able to write our components the way we want to write them, so that we have no friction when we want to pass data around.



But you still render this differently in the real DOM.

For example, to render the modal html content somewhere else than it would normally go to so that on the left, JSX code re-write hasn't changed, but the rendered HTML code is a bit different from our JSX code. So, the Modal there is not next to the form. You can achieve this with React portals.

### 104. Working with Portals

We of course have an ErrorModal in our project. We can use a fragment in it.

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const ErrorModal = (props) => {  return (  **<React.Fragment>**  <div className={classes.backdrop} onClick={props.onConfirm} />  <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  **</React.Fragment>**  );  };  export default ErrorModal; |

We now should use a portal because the entire ErrorModal component should not be rendered in the place it is being rendered.

Portals need two things

1. You need a place where you want to port the component to.
2. You need to let the component know that it should have a portal to that place.

To mark that place, we go into the "public" folder and then into "index.html". In this file it is common to add a div with an id, which you will then use to identify this place later. You could have ids called "backdrop-root" and "modal-root" and you could create multiple such roots for different kinds of Components that should be portaled there.

|  |
| --- |
| public/index.html |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="utf-8" />  <link rel="icon" href="%PUBLIC\_URL%/favicon.ico" />  <meta name="viewport" content="width=device-width, initial-scale=1" />  <meta name="theme-color" content="#000000" />  <meta  name="description"  content="Web site created using create-react-app"  />  <link rel="apple-touch-icon" href="%PUBLIC\_URL%/logo192.png" />  <!--  manifest.json provides metadata used when your web app is installed on a  user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/  -->  <link rel="manifest" href="%PUBLIC\_URL%/manifest.json" />  <!--  Notice the use of %PUBLIC\_URL% in the tags above.  It will be replaced with the URL of the `public` folder during the build.  Only files inside the `public` folder can be referenced from the HTML.  Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC\_URL%/favicon.ico" will  work correctly both with client-side routing and a non-root public URL.  Learn how to configure a non-root public URL by running `npm run build`.  -->  <title>React App</title>  </head>  <body>  <noscript>You need to enable JavaScript to run this app.</noscript>  **<div id="backdrop-root"></div>**  **<div id="modal-root"></div>**  <div id="root"></div>  <!--  This HTML file is a template.  If you open it directly in the browser, you will see an empty page.  You can add webfonts, meta tags, or analytics to this file.  The build step will place the bundled scripts into the <body> tag.  To begin the development, run `npm start` or `yarn start`.  To create a production bundle, use `npm run build` or `yarn build`.  -->  </body>  </html> |

You could simplify this a bit and have an id of "overlay-root" rather than "modal-root", which will then hold all kinds of overlays – modals, side drawers, and so on. That's what we'll go with.

|  |
| --- |
| public/index.html |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="utf-8" />  <link rel="icon" href="%PUBLIC\_URL%/favicon.ico" />  <meta name="viewport" content="width=device-width, initial-scale=1" />  <meta name="theme-color" content="#000000" />  <meta  name="description"  content="Web site created using create-react-app"  />  <link rel="apple-touch-icon" href="%PUBLIC\_URL%/logo192.png" />  <!--  manifest.json provides metadata used when your web app is installed on a  user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/  -->  <link rel="manifest" href="%PUBLIC\_URL%/manifest.json" />  <!--  Notice the use of %PUBLIC\_URL% in the tags above.  It will be replaced with the URL of the `public` folder during the build.  Only files inside the `public` folder can be referenced from the HTML.  Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC\_URL%/favicon.ico" will  work correctly both with client-side routing and a non-root public URL.  Learn how to configure a non-root public URL by running `npm run build`.  -->  <title>React App</title>  </head>  <body>  <noscript>You need to enable JavaScript to run this app.</noscript>  <div id="backdrop-root"></div>  <div id="**overlay-root**"></div>  <div id="root"></div>  <!--  This HTML file is a template.  If you open it directly in the browser, you will see an empty page.  You can add webfonts, meta tags, or analytics to this file.  The build step will place the bundled scripts into the <body> tag.  To begin the development, run `npm start` or `yarn start`.  To create a production bundle, use `npm run build` or `yarn build`.  -->  </body>  </html> |

We save index.html and then go back to our components.

We now work on the ErrorModal and tell React that it should be portaled somewhere. We can do this by creating a constant, and calling it Backdrop, which will actually be a new Component. We'll add it in this same file because in this app, I only use this Backdrop component in conjunction with my ErrorModal. So we will store both the ErrorModal and Backdrop components in one big file but we could also split it into multiple component files, especially if you would use the backdrop in conjunction with other component files as well. We pass in props and then we return that div that gets my backdrop class: <div className={classes.backdrop} onClick={props.onConfirm} />.

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  **const Backdrop = (props) => {**  **return <div className={classes.backdrop} onClick={props.onConfirm} />**  **};**  const ErrorModal = (props) => {  return (  <React.Fragment>  <div className={classes.backdrop} onClick={props.onConfirm} />  <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  </React.Fragment>  );  };  export default ErrorModal; |

We will also add our ModalOverlay component, which gets props, and there I will return the Card and remove Card from where we are currently using it inside of the ErrorModal component.

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const Backdrop = (props) => {  return <div className={classes.backdrop} onClick={props.onConfirm} />  };  **const ModalOverlay = (props) => {**  **return <Card className={classes.modal}>**  **<header className={classes.header}>**  **<h2>{props.title}</h2>**  **</header>**  **<div className={classes.content}>**  **<p>{props.message}</p>**  **</div>**  **<footer className={classes.actions}>**  **<Button onClick={props.onConfirm}>Okay</Button>**  **</footer>**  **</Card>**  **};**  const ErrorModal = (props) => {  return (  <React.Fragment>  <div className={classes.backdrop} onClick={props.onConfirm} />  </React.Fragment>  );  };  export default ErrorModal; |

I have basically split my modal into two separate Components now – Backdrop and ModalOverlay – because that will make working with portals much easier.

We can also remove the div with the class of backdrop from the ErrorModal component.

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const Backdrop = (props) => {  return <div className={classes.backdrop} onClick={props.onConfirm} />  };  const ModalOverlay = (props) => {  return <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  };  const ErrorModal = (props) => {  return (  <React.Fragment>  </React.Fragment>  );  };  export default ErrorModal; |

What do we do inside of the fragment in the ErrorModal? We can add an expression because we're still inside of JSX code, and we want to call a method that is not actually defined on React but on another library that comes together with React, the react-dom library. You can imagine React as being the library that has all the React features, state management, and so on baked in. And react-dom uses React to bring that logic and these features into the web browser, so making them compatible with working with the DOM.

Put in other words, the react library doesn't care whether you run it in an environment that has a DOM or if you would use it to build a native app. So react-dom is kind of the adapter for React to the browser. And, therefore, since now we're going to portal something into a different place in the real DOM we need to import react-dom.

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  **import ReactDOM from 'react-dom';**  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const Backdrop = (props) => {  return <div className={classes.backdrop} onClick={props.onConfirm} />  };  const ModalOverlay = (props) => {  return <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  };  const ErrorModal = (props) => {  return (  <React.Fragment>  </React.Fragment>  );  };  export default ErrorModal; |

Inside of the React.Fragment and on ReactDOM you can now call a createPortal method.

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import ReactDOM from 'react-dom';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const Backdrop = (props) => {  return <div className={classes.backdrop} onClick={props.onConfirm} />  };  const ModalOverlay = (props) => {  return <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  };  const ErrorModal = (props) => {  return (  <React.Fragment>  **{ReactDOM.createPortal()}**  </React.Fragment>  );  };  export default ErrorModal; |

The createPortal method takes two arguments. The first one is your React node that should be rendered and here we can render our Backdrop lets say. It's important that we pass JSX, so pass <Backdrop /> rather than Backdrop because your really render it like this: <Backdrop />. We can then forward our onConfirm prop and get access to props.onConfirm, which I need to pass here to make sure that everything still works. I can now set onConfirm in my ErrorModal and forward the function I get on the onConfirm prop to the onClick prop inside of the Backdrop component.

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import ReactDOM from 'react-dom';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const Backdrop = (props) => {  return <div className={classes.backdrop} onClick={props.onConfirm} />  };  const ModalOverlay = (props) => {  return <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  };  const ErrorModal = (props) => {  return (  <React.Fragment>  {ReactDOM.createPortal(**<Backdrop onConfirm={props.onConfirm} />**, )}  </React.Fragment>  );  };  export default ErrorModal; |

The second argument of createPortal is a pointer to that container in the real DOM where this element should be rendered in. And in my case, we of course want to render our backdrop here in that "backdrop-root" component (the div inside of "public/index.html" that has an id of "backdrop-root"). We select the element where it should be rendered to, and for this, we use a DOM API. We use document.getElementById('backdrop-root') for example and get access to the "backdrop-root".

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import ReactDOM from 'react-dom';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const Backdrop = (props) => {  return <div className={classes.backdrop} onClick={props.onConfirm} />  };  const ModalOverlay = (props) => {  return <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  };  const ErrorModal = (props) => {  return (  <React.Fragment>  {ReactDOM.createPortal(<Backdrop onConfirm={props.onConfirm} />, **document.getElementById('backdrop-root')** )}  </React.Fragment>  );  };  export default ErrorModal; |

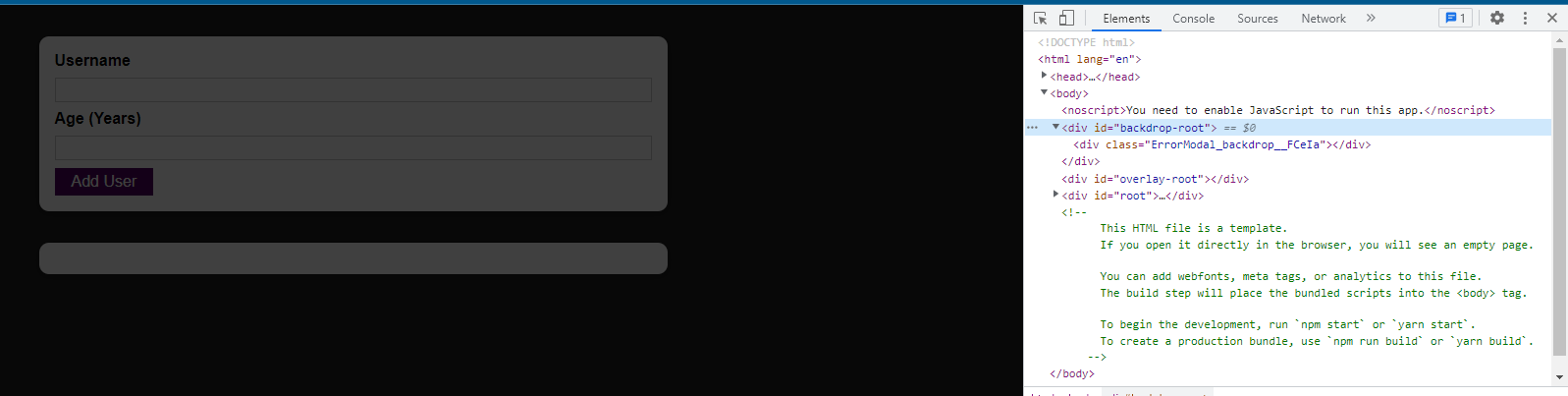
We really get access to a real HTML DOM element, a DOM node here. And we do this with the API that is provided by the browser. document.getElementById has nothing to do with React. WE really get access to a real DOM element with this API.

This is similar how we selected an element in "src/index.js". In "src/index.js" we also rendered the root component with the render method into a place selected with getElementById.

|  |
| --- |
| src/index.js |
| 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(<App />); |

In the ErrorModal we're not rendering an element but inside of an existing application which is already being rendered by React, we portal. We move the HTML content that is about to be rendered into a different place.

If we reload the browser at this point, if we click the backdrop appears, but the modal does not since we haven't added the logic for this yet.



In Chrome's Dev Tools in the Elements tab, we see that in the "backdrop-root", we now have the modal backdrop. And now it will always be there no matter where you would use your error modal in your JSX code. No matter how deeply nested it is in other elements. It will always be here which is of course very close to the body.

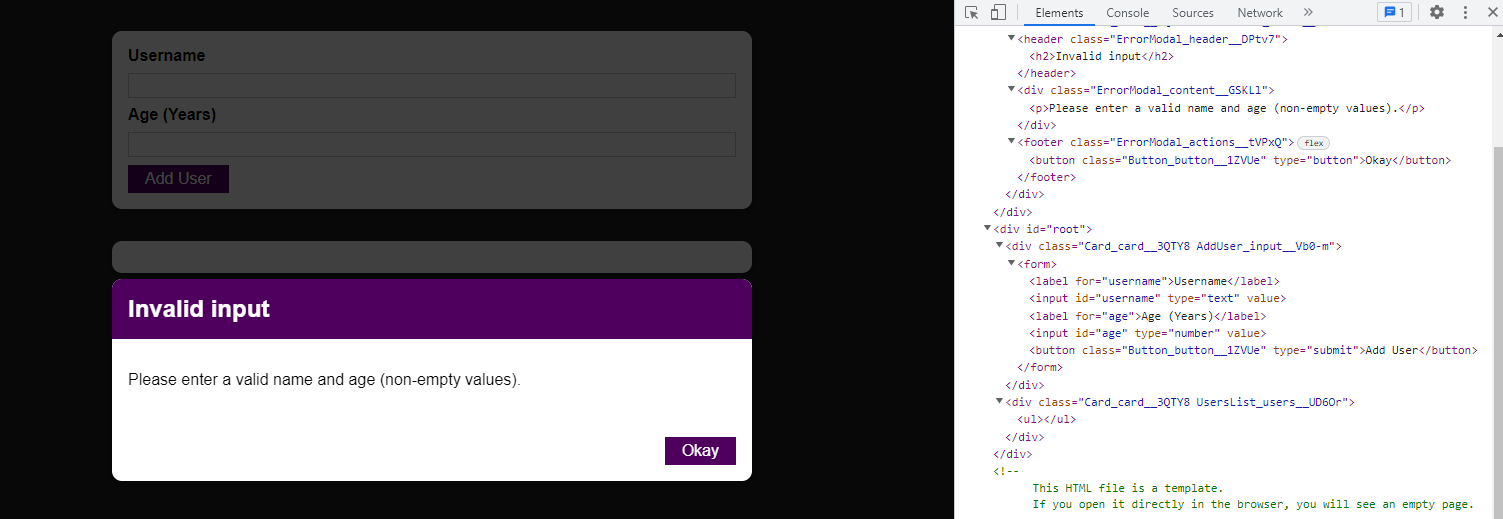
Now we will repeat this for the actual overlay. For that we will add a new expression inside of the ErrorModal next to the expression we just added. Here we use createPortal, but this time we want to render our ModalOverlay component, and my ModalOverlay component needs a bunch of props, which I now of course want to forward. It needs a title, message, and onConfirm.

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import ReactDOM from 'react-dom';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const Backdrop = (props) => {  return <div className={classes.backdrop} onClick={props.onConfirm} />  };  const ModalOverlay = (props) => {  return <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  };  const ErrorModal = (props) => {  return (  <React.Fragment>  {ReactDOM.createPortal(  <Backdrop onConfirm={props.onConfirm} />,  document.getElementById('backdrop-root')  )}  {ReactDOM.createPortal(  **<ModalOverlay**  **title={props.title}**  **message={props.message}**  **onConfirm={props.onConfirm}**  **/>,**  )}  </React.Fragment>  );  };  export default ErrorModal; |

Now just as with the backdrop, we need to let React DOM know where it should render this HTML content where it needs to be rendered. For that, I'll use document.getElementById and select my overlay-root, which was the other div that I created in "public/index.html."

|  |
| --- |
| src/components/UI/ErrorModal.js |
| import React from 'react';  import ReactDOM from 'react-dom';  import Card from './Card';  import Button from './Button';  import classes from './ErrorModal.module.css';  const Backdrop = (props) => {  return <div className={classes.backdrop} onClick={props.onConfirm} />  };  const ModalOverlay = (props) => {  return <Card className={classes.modal}>  <header className={classes.header}>  <h2>{props.title}</h2>  </header>  <div className={classes.content}>  <p>{props.message}</p>  </div>  <footer className={classes.actions}>  <Button onClick={props.onConfirm}>Okay</Button>  </footer>  </Card>  };  const ErrorModal = (props) => {  return (  <React.Fragment>  {ReactDOM.createPortal(  <Backdrop onConfirm={props.onConfirm} />,  document.getElementById('backdrop-root')  )}  {ReactDOM.createPortal(  <ModalOverlay  title={props.title}  message={props.message}  onConfirm={props.onConfirm}  />,  **document.getElementById('overlay-root')**  )}  </React.Fragment>  );  };  export default ErrorModal; |

And with that saved, if we reload and click "Add User," we get the modal popup. With the modal it works and everything is now rendered in the divs where it should be rendered and no longer nested in our other HTML code.



The idea behind a portal really just is that the rendered HTML content is moved somewhere else. ReactDOM.createPortal can be used anywhere where you would otherwise use JSX code. Of course wrap in curly braces because we're using JavaScript code. So wherever you would normally just use a component, you can use createPortal to move that Component's HTML content somewhere else, only in the actual DOM that is being rendered.