**React – Introduction**

**Importing React and ReactDOM modules**

We can import **React** and **ReactDOM** modules as below:

import React from "react";

import ReactDOM from "react-DOM";

This allows us to use the functions in this module to, for example, use HTML embedded in JS or **render** our code (i.e. show it in the screen). For example, with these modules, we can render elements in a much simpler way:

**With React and ReactDOM**:

ReactDOM.render(<h1>Hello World</h1>>, document.getElementById("root"));

**Without React and ReactDOM**:

var h1 = document.createElement("h1");

h1.innerHTML = "Hello World";

document.getElementById("root.appendChild")(h1)

The **root** is simply a **div** in HTML code where all our JS code goes:

**HTML Code:**

<!DOCTYPE html>

<html lang="en">

  <head>

    <title>Document</title>

    <link rel="stylesheet" href="styles.css" />

  </head>

  <body>

    <div id="root"></div>

    <script src="../src/index.js" type="text/jsx"></script>

  </body>

</html>

**Note**: React allows us to use ES6 without worrying with the browser. Inside the React module there is something called Babel which is a JS compiler. So, it is able to get next generation JS like ES6,7 or 8 and compile it down to a version of JS every browser can understand (even Internet Explorer).

**JSX Rules**

**Rendering can only take one HTML parent element**

Rendering can only take one HTML parent element, but if we put everything inside a **div**, whatever is inside that div doesn’t count. For example:

|  |  |
| --- | --- |
| **Possible** | **NOT possible** |
| ReactDOM.render(    <div>      <h1>Hello</h1>      <h1>World</h1>    </div>, document.getElementById("root")); | ReactDOM.render(  <h1>Hello</h1>  <h1>World</h1>, document.getElementById("root")); |

This syntax that allows us to use React and put HTML embedded in JS code is called JSX.

**Camel Case Syntax**

While in HTML we are used to write classes in lower case, in JSX we use camel notation:

|  |
| --- |
| **HTML** |
| <div tabindex="1">    <button onclick="myFunction()">click me</button>    <label for="name">Name</label>    <input readonly id="name" />  </div> |
| **JSX** |
| return (    <div tabIndex={1}>      <button onClick={myFunction}>click me</button>      <label htmlFor='name'>Name</label>      <input readOnly={true} id='name' />    </div>  ) |

**Class Keyword**

The **class** keyword is not used in JSX. Instead, use **className**.

**Self-closing tags in JSX**

HTML5 allowed us the not use **/** on self-closing tags. However, in JSX that is still needed:

|  |  |
| --- | --- |
| **HTML** | **JSX** |
| return <img>; | return <img />; |

**JS Expressions in JSX**

**JSX** stands for JavaScript XML. It is simply a syntax extension of JavaScript. It allows us to **directly write HTML in React** (within JavaScript code). In JSX, **only one single parent element can be returned**.

You can write JS expressions inside HTML using **{}**. For example:

import React from "react";

import ReactDOM from "react-dom";

const name = "Angela"

const num = 7

ReactDOM.render(

    <div>

        <h1>My name is {name}</h1>

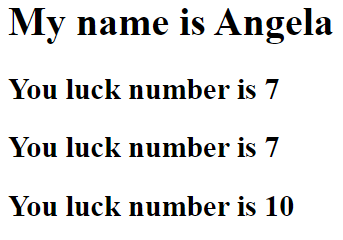
        <h2>You luck number is {num}</h2>

        <h2>You luck number is {3 + 4}</h2>

        <h2>You luck number is {Math.ceil(Math.random()\*10)}</h2>

    </div>

    , document.getElementById("root"));



**JSX Attributes & Styling React Elements**

**External CSS**

In order to keep the styling in the CSS, we should **use JSX to give classes to the HTML elements**, and NOT to style them (with certain exceptions). See the example below, where we use **className** to assign a class to the elements (and then the classes are styles in the CSS file).

|  |
| --- |
| **JSX file** |
| import React from "react";  import ReactDOM from "react-dom";  ReactDOM.render(    <div>      <h1 className="heading">My Favourite Foods</h1>    <img        className="food-img"        alt="bacon"        src="https://hips.hearstapps.com/hmg-prod.s3.amazonaws.com/images/delish-190621-air-fryer-bacon-0035-landscape-pf-1567632709.jpg?crop=0.645xw:0.967xh;0.170xw,0.0204xh&resize=480:\*"      />      <img        className="food-img"        alt="jamon"        src="https://images-na.ssl-images-amazon.com/images/I/71lNrnbMXsL.\_SL1200\_.jpg"      />      <img        className="food-img"        alt="noodles"        src="https://www.errenskitchen.com/wp-content/uploads/2014/04/quick-and-easy-chinese-noodle-soup3-1.jpg"      />    </div>,    document.getElementById("root")  ); |
| **CSS file** |
| .heading {    color: red;  }  .food-img {    height: 100px;    width: 100px;  } |
| **Result** |
|  |

**Inline Styling for React Elements**

As mentioned, styling should always be in the CSS file, and we should use JSX to give classes to the elements only. But there are some exceptions. You may want the **style to update on the fly**, i.e. overwrite the normal style for a certain element. That’s when inline styling becomes useful.

|  |  |
| --- | --- |
| import React from "react";  import ReactDOM from "react-dom";  const customStyle = {      color: "red",      fontSize: "20px"  };  ReactDOM.render(    <div>      <h1 style={customStyle}>Hello World</h1>    </div>,    document.getElementById("root")  ); |  |

You can also change a single property of the object. This this case, the final color is blue because the code was called after. This is useful, for example, if we want to change the style when a button is clicked (for example). In that case, we would add an event listener to the **h1** element and change the property within a function (more on event listeners later).

|  |  |
| --- | --- |
| import React from "react";  import ReactDOM from "react-dom";  const customStyle = {      color: "red",      fontSize: "20px"  };  customStyle.color = "blue";  ReactDOM.render(    <div>      <h1 style={customStyle}>Hello World</h1>    </div>,    document.getElementById("root")  ); |  |

**React Components**

**Components** allow us to **split up a large file** or a complex web structure into **smaller** **components**. We also get the added benefit of **reusing** **each** **of** **these** **components** when we need the same functionality.

All **components** have names which **start with a capital letter** (Pascal case). That’s how React differentiates the HTML elements from the components.

It’s a good practice to keep the components in a separate folder, with a JSX extension (e.g. Heading.jsx).

To create components:

1. Take the **HTML** **elements** you want to set as a **component** and put inside a **separate** **JSX** **file**. This HTML code will be **returned** **inside** **a** **function**, and that function **exported** in the end.
2. Import the function in the parent file (in this case Index.js).
3. Add the components inside the **div** (same name as the component function name).

|  |  |
| --- | --- |
| **Index.js** | **List.jsx** |
| import React from "react";  import ReactDOM from "react-dom";  import Heading from "./components/Heading"  import List from "./components/List"  ReactDOM.render(    <div>  *<Heading />*  *<List/>*  *<List />*  *</div>,*  *document.getElementById("root")*  *);* | import React from "react"  function List () {      return(          <ul>          <li>Bacon</li>          <li>Jamon</li>          <li>Noodles</li>      </ul>      )  }  export default List; |
| **Heading.jsx** | **Result** |
| import React from "react"  function Heading () {      return(          <h1>My Favourite Food</h1>      )  }  export default Heading |  |

As you can see, because we have **called** **the** **component** **twice** **in** **the** **component** **tree** (see index.js), the component **List** was easily **repeated** **by** **rendering** **it** **twice**.

Normally, there are no HTML elements in the index.js of React apps. There is just one custom component called **App.jsx**, so generally, **index.js** will look very simple.

|  |  |
| --- | --- |
| **Index.js** | **List.jsx** |
| import React from "react";  import ReactDOM from "react-dom";  import App from "./components/App"  ReactDOM.render(<App/>,document.getElementById("root")); | import React from "react"  function List () {      return(          <ul>          <li>Bacon</li>          <li>Jamon</li>          <li>Noodles</li>      </ul>      )  }  export default List; |
| **Heading.jsx** | **App.jsx** |
| import React from "react"  function Heading () {      return(          <h1>My Favourite Food</h1>      )  }  export default Heading | import React from "react";  import Heading from "./Heading"  import List from "./List"  function App () {      return(      <div>          <Heading/>          <List/>          <List/>      </div>      );  }  export default App; |

**ES6 Import/Export Modules**

In React, the components are split into separate files, which means we have to export them and then import them in the parent file. Have a look at the example below.

|  |  |  |
| --- | --- | --- |
| **Index.js** | **Math.jsx** | **Result** |
| import React from "react";  import ReactDOM from "react-dom";  import pi, {doublePi,triplePi} from "./components/Math"  ReactDOM.render(      <ul>          <li>{pi}</li>          <li>{doublePi()}</li>          <li>{triplePi()}</li>      </ul>      ,document.getElementById("root")); | const pi = 3.14;  function doublePi() {      return pi \* 2;  }  function triplePi() {      return pi \* 3;  }  export default pi;  export {doublePi, triplePi}; |  |

**Exporting files**

There are types of export: **named** and **default**.

In the example above, we export more than one element from a component. When this happens, **one of the elements is exported as default**. All the other ones are exported and imported inside **{}** – named export.

In **named exports**, the import name at the parent file needs to match exactly the export name. Also, we can do named exports in two ways:

|  |  |
| --- | --- |
| **In-line individually** | **All at once in the bottom** |
| export const name = "Jesse"  export const age = 40 | const name = "Jesse"  const age = 40  export { name, age } |

You can only do **one default export per component**. Also, when importing it in the parent file, you don’t need to import it with the same name as it was exported. You can import it with any name, and then when you use it in your code, you need to use the imported name.

**Importing files**

You can import modules into a file in two ways, based on if they are **named** **exports** or **default** **exports**.

|  |  |
| --- | --- |
| **Importing a named export** | **Importing a default export** |
| import { name, age } from "./person.js"; | import message from "./message.js"; |

**Importing Local Images**

When importing local images, you can use the following the setup below:

import img1 from "./images/book-1.jpg";

import img2 from "./images/book-2.jpg";

import img3 from "./images/book-3.jpg";

export const books = [

  {

    author: "Jordan Moore",

    title: "Interesting Facts For Curious Minds",

    img: img1,

    id: 1,

  },

  {

    author: "James Clear",

    title: "Atomic Habits",

    img: img2,

    id: 2,

  },

  {

    author: "Stephen King",

    title: "Fairy Tale",

    img: img3,

    id: 3,

  },

];

**File paths**

A file path describes the location of a file in a web site's folder structure.

|  |  |
| --- | --- |
| **Path** | **Description** |
| <img src="picture.jpg"/> | The "**picture.jpg**" file is located in the **same** **folder** as the current page |
| <img src="images/picture.jpg"/> | The "**picture.jpg**" file is located in the **images** folder in the current folder |
| <img src="/images/picture.jpg"/> | The "**picture.jpg**" file is located in the **images** folder at the **root** of the current web |
| <img src="../picture.jpg"/> | The "**picture.jpg**" file is located in the folder **one level up** from the current folder |

**React Props**

**Props** stands for **properties**. **Props** are **arguments** passed into React components when they are created or setup. They are then defined when the component is rendered.

If the property value sent is not a string, this need to be inside curly braces **{}**.

We can assign different values to the properties each time we call a component, as it is shown in the image below. This is useful when we want to **reuse** **a** **block** that has the **same** **structure**, but **different** **data**.

|  |  |
| --- | --- |
| **Index.js** | **Card.jsx** |
| import React from "react";  import ReactDOM from "react-dom";  import App from "./components/App"  ReactDOM.render(      <div>          <h1>My Contacts</h1>          <App/>      </div>,  document.getElementById("root")  ); | import React from "react";  function Card(props) {    return (      <div>        <h2>{props.name}</h2>        <p>{props.tel}</p>        <p>{props.email}</p>      </div>    );  }  export default Card; |
| **App.jsx** | **Result** |
| import React from "react"  import Card from "./Card"  function App () {      return(          <div>              <Card                  name="Beyonce"                  tel="+123 456 789"                  email="b@beyonce.com"              />              <Card                  name="Jack Bauer"                  tel="+7387384587"                  email="jack@nowhere.com"              />          </div>      )  }  export default App; |  |

A more organized way to give properties to a component is by **creating a separate JS file with an array of objects**, and then access the properties of each object it using the index number and the name of the property.

We can also use **destructuring** when setting up the object, so instead of repeating **props** each time we want to access it, we destruct it once (see **Card** file).

|  |  |
| --- | --- |
| **Index.js** | **Card.jsx** |
| import React from "react";  import ReactDOM from "react-dom";  import App from "./components/App"  ReactDOM.render(      <div>          <h1>My Contacts</h1>          <App/>      </div>,  document.getElementById("root")  ); | import React from "react";  function Card(props) {      const {name, tel, email} = props;    return (      <div>        <h2>{ name }</h2>        <p>{ tel }</p>        <p>{ email }</p>      </div>    );  }  export default Card; |
| **App.jsx** | **Contact.js** |
| import React from "react"  import Card from "./Card"  import contacts from "../Contacts"  function App () {    return(       <div>         <Card          name={contacts[0].contactName}           tel={contacts[0].contactTel}           email={contacts[0].contactEmail}         />         <Card          name={contacts[1].contactName}          tel={contacts[1].contactTel}           email={contacts[1].contactEmail}         />       </div>     )  }  export default App; | const contacts = [      {        contactName: "Beyonce",        contactTel: "+123 456 789",        contactEmail: "b@beyonce.com"      },      {        contactName: "Jack Bauer",        contactTel: "+7387384587",        contactEmail: "jack@nowhere.com"      }  ]  export default contacts; |

**Children Prop**

If we want to render something that is between the component tags, we need to use a special prop – and the name is **children**.

In the example below, you use **children** prop to target and render everything else from the **Book.jsx** component that is passed as prop and comes after the properties of the books array object, which is the **<p>** and the **<button>**.

|  |  |
| --- | --- |
| **Index.js** | **Book.jsx** |
| import React from "react";  import ReactDOM from "react-dom";  import App from "./components/App"  ReactDOM.render(      <div>          <App/>      </div>,  document.getElementById("root")  ); | import React from "react";  function Book ({img, title, author, children}) {    return (      <article className="book">        <img src={img} alt={title} />        <h2>{title}</h2>        <h4>{author}</h4>        {children}      </article>    );  };  export default Book; |
| **App.jsx** | **Booklist.js** |
| import React from "react"  import Book from "./Book"  import books from "../Booklist"  function App () {    return(       <div>         <Book          img={books[0].img}           title={books[0].title}           author={books[0].author}          <p>This is extra text</p>          <button>click me</button>         />        </Book>         <Book          img={books[1].img}          title={books[1].title}           author={books[1].author}         />       </div>     )  }  export default App; | const books = [      {    author: "Lloyd Devereux Richards",    title: "Stone Maidens",    img: "https://images-na.ssl-images-amazon.com/images/I/51NcOGy0y3L.\_AC\_UL600\_SR600,400\_.jpg",      },      {    author: "B. Dylan Hollis",    title: "Baking Yesteryear: The Best Recipes from the 1900s to the 1980s",    img: "https://images-na.ssl-images-amazon.com/images/I/81Oa54UCQoL.\_AC\_UL600\_SR600,400\_.jpg",      }  ]  export default contacts; |
| **Result** | |
|  | |

**Mapping Data to Components**

**Revision**

**map()** creates a new array from calling a function for every array element. For example:

const numbers = [65, 44, 12, 4];

const newArr = numbers.map(myFunction)

function myFunction(num) {

  return num \* 10;

}

console.log(newArr)

//expected output [650, 440, 120, 40]

**Mapping data in React**

Have a look at the example below. The **map()** function loops through the array of contacts and calls the function **createCard()** on every item of the array.

So, on the first loop, it gets hold of the properties of each object in the contacts array (**contactName**, **contactTel** and **contactEmail**). Then, when we call the component, we assign those values we took and assign it to the Props values (**name**, **tel** and **email**) of the first object (Beyonce).

The **key prop is mandatory when mapping through arrays**. Each object of that array should have a key property associated with a unique **id**.

|  |  |
| --- | --- |
| **Index.js** | **Card.jsx** |
| import React from "react";  import ReactDOM from "react-dom";  import App from "./components/App"  ReactDOM.render(      <div>          <h1>My Contacts</h1>          <App/>      </div>,  document.getElementById("root")  ); | import React from "react";  function Card(props) {    return (      <div>        <h2>{props.name}</h2>        <p>{props.tel}</p>        <p>{props.email}</p>      </div>    );  }  export default Card; |
| **App.jsx** | **Contact.js** |
| import React from "react"  import Card from "./Card"  import contacts from "../Contacts"  function createCard(contact) {      return (        <Card          key={contact.id}          name={contact.contactName}          tel={contact.contactTel}          email={contact.contactEmail}        />      );    }  function App () {      return(          <div>              {contacts.map(createCard)}          </div>      )  }  export default App;  } | const contacts = [      {          id = 1,          contactName: "Beyonce",          contactTel: "+123 456 789",          contactEmail: "b@beyonce.com"      },      {          id = 2,          contactName: "Jack Bauer",          contactTel: "+7387384587",          contactEmail: "jack@nowhere.com"      }  ]  export default contacts; |
| **Result** | |
|  | |

Another way is by using the **spread operator**. So, each time the **map** function returns a **Book** component, it will spread out all the properties (**author**, **title** and **img**) and then also pass in the **id** as a **key**, once it’s mandatory when mapping through component properties in React.

|  |  |
| --- | --- |
| **Index.js** | **Book.jsx** |
| import React from "react";  import ReactDOM from "react-dom";  import App from "./components/App"  ReactDOM.render(      <div>          <App/>      </div>,  document.getElementById("root")  ); | import React from "react";  function Book ({img, title, author}) {    return (      <article className="book">        <img src={img} alt={title} />        <h2>{title}</h2>        <h4>{author}</h4>      </article>    );  };  export default Book; |
| **App.jsx** | **Booklist.js** |
| import React from "react"  import Book from "./Book"  import books from "../Booklist"  function App () {    return (      <section className="booklist">        {books.map((book) => {          return <Book {...book} key={book.id} />;        })}      </section>    );  }  export default App; | import img1 from './images/book-1.jpg';  import img2 from './images/book-2.jpg';  import img3 from './images/book-3.jpg';  export const books = [    {      author: 'Jordan Moore',      title: 'Interesting Facts For Curious Minds',      img: img1,      id: 1,    },    {      author: 'James Clear',      title: 'Atomic Habits',      img: img2,      id: 2,    },    {      author: 'Stephen King',      title: 'Fairy Tale',      img: img3,      id: 3,    },  ]; |

**Conditional Rendering**

Sometimes we may want to show a different component depending on some condition – and that’s when we use **conditional** **rendering**.

In the example below, it will show “Hello” if the user is Logged in, or the Login screen if the user hasn’t logged in yet. For this we use the **Ternary** **Operator**, which follow the syntax below:

**Condition ? Do it if True : Do it if False**

So, when we call the following line:

{isLoggedIn === true ? <h1>Hello</h1> : <Login />}

It means that, depending on the value of the **isLoggedIn** variable (**true** or **false**), we will display different things. If it is **true**, it will display “Hello”. If it is **false**, it will display the **Login** component.

**Note**: In this case, for simplification, we are just changing the value of **isLoggedIn** manually.

|  |  |  |
| --- | --- | --- |
| **Index.js** | **App.jsx** | |
| import React from "react";  import ReactDOM from "react-dom";  import App from "./components/App";  ReactDOM.render(<App />, document.getElementById("root")); | import React from "react";  import Login from "./Login";  const isLoggedIn = false;  function App() {    return (      <div className="container">        {isLoggedIn === true ? <h1>Hello</h1> : <Login />}      </div>    );  }  export default App; | |
| **Input.jsx** | **Login.js** | |
| import React from "react";  function Input(props) {    return <input type={props.type} placeholder={props.placeholder} />;  }  export default Input; | import React from "react";  import Input from "./Input";  function Login(props) {    return (      <form className="form">        <Input type="text" placeholder="Username" />        <Input type="password" placeholder="Password" />        <button type="submit">Login</button>      </form>    );  }  export default Login; | |
| **Result** | | |
|  | | var isLoggedIn = false; |
|  | | var isLoggedIn = true; |

Now imagine we want to add a “confirm password” field in case the user was not registered. In this case, we want to show the field “confirm password” **if the user is not registered**. We can put call the component inside a condition statement using the **ternary operator**, so the component is only rendered if the condition applies, i.e. **isRegistered** is **true**.

|  |  |  |
| --- | --- | --- |
| **Index.js** | **App.jsx** | |
| import React from "react";  import ReactDOM from "react-dom";  import App from "./components/App";  ReactDOM.render(<App />, document.getElementById("root")); | import React from "react";  import Form from "./Form";  var userIsRegistered = false;  function App() {    return (      <div className="container">        <Form isRegistered={userIsRegistered} />      </div>    );  }  export default App; | |
| **Input.jsx** | | |
| import React from "react";  function Input(props) {    return <input type={props.type} placeholder={props.placeholder} />;  }  export default Input; | | |
| **Form.jsx** | | |
| import React from "react";  import Input from "./Input";  function Form(props) {    return (      <form className="form">        <Input type="text" placeholder="Username" />        <Input type="password" placeholder="Password" />  **{props.isRegistered === false && (**  **<Input type="password" placeholder="Confirm Password" />**  **)}**        <button type="submit">          {props.isRegistered === true ? "Login" : "Register"}        </button>      </form>    );  }  export default Form; | | |
| **Result** | | |
|  | | var isRegistered = true; |
|  | | var isRegistered = false; |

There are different ways to set up this ternary operator:

|  |
| --- |
| isRegistered === false && (<Input type="password" placeholder="Confirm Password" />) |
| isRegistered === false ? (<Input type="password" placeholder="Confirm Password" />) : null |
| !isRegistered ? (<Input type="password" placeholder="Confirm Password" />) : null |

**React Hooks – Use State**

**Hooks** allow function components to have access to **state** and other React features.

The **state** is a built-in React object that is used to **contain** **data** **or** **information** about the **component**. A component’s state can change over time; **whenever it changes, the component re-renders**. The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.

The **React** **useState** **Hook** allows us to **track** **state** **in** **a** **function** **component**.

**useState** accepts an **initial** **state** and **returns** **two** **values**:

* **The current state.**
* **A function that updates the state.**

For example, the code below takes the variable **color** as the current value, and **setColor** as the function that updates de color. The **useState** function takes an argument, which is the initial value of the state (in this case, it’s just an empty string, but it can be anything).

function FavoriteColor() {

  const [color, setColor] = useState("");

}

The **useState** **Hook** can be used to keep track of **strings**, **numbers**, **booleans**, **arrays**, **objects**, and any **combination** **of** **these.**

For example, below it is tracking the state of 3 variables **brand**, **model** and **year**.

function Car() {

  const [brand, setBrand] = useState("Ford");

  const [model, setModel] = useState("Mustang");

  const [year, setYear] = useState("1964");

  return (

    <>

      <h1>My {brand}</h1>

      <p>

        It is a {model} from {year}.

      </p>

    </>

  )

}

Or, we can just use **one** **state** and include an **object** instead. Both ways would lead to the same result:

|  |  |
| --- | --- |
| **car.jsx** | **Result** |
| function Car() {    const [car, setCar] = useState({      brand: "Ford",      model: "Mustang",      year: "1964",    });    return (      <>        <h1>My {car.brand}</h1>        <p>          It is a {car.model} from {car.year}.        </p>      </>    )  } |  |

**Another Example**

Consider the following code, where we have a counter that goes up every time the button **+** is pressed. The variable **count** increases in value. However, it is not re-rendered to the screen.

|  |  |
| --- | --- |
| import React from "react";  import ReactDOM from "react-dom";  var count = 0;  function increase() {      count++  }  ReactDOM.render(      <div className="container">          <h1>{count}</h1>          <button onClick={increase}>+</button>      </div>  ) |  |

We can copy and paste bit of the code that renders the HTML elements inside the increase function so it is rendered every time the buttom is clicked, but that’s **not** **good** **practice** due to **repetitive** and **long** **code**.

|  |  |
| --- | --- |
| var count = 0;  function increase() {      count++      ReactDOM.render(          <div className="container">              <h1>{count}</h1>              <button onClick={increase}>+</button>          </div>      )  }  ReactDOM.render(      <div className="container">          <h1>{count}</h1>          <button onClick={increase}>+</button>      </div>  ) |  |

The correct way is to use **React.useState()**.

It has **2** **arguments**:

* The **value** that is going to change (**count**)
* A **function** to change that value (**setCount**)

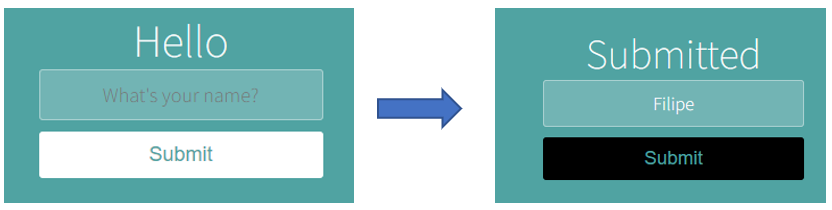
The **setCount** function will be inside another function, which is going to be called when the event is triggered. This is the typical format when using React Hook.

|  |  |
| --- | --- |
| import React, { useState } from "react";  function App() {    const [count, setCount] = React.useState(0);    function increase() {      setCount(count + 1);    }    return (      <div className="container">        <h1>{count}</h1>        <button onClick={increase}>+</button>      </div>    );  }  export default App; |  |

**Event Handling in React**

A lot of the times, we will want **events to change the state of React components**. Imagine you have the following interface and you want to:

* Change the text from “Hello” to “Submitted” as soon as you submit a name.
* Change the background color of the button if you are hovering the mouse over it.



We will want to control two states:

* Heading Text
* If the Mouse is over the “submit” button

1. So, we need to create 2 hooks:

  const [headingText, setHeadingText] = React.useState("Hello");

  const [isMousedOver, setMouseOver] = React.useState(false);

1. To change the state of the heading when the button “submit” is clicked, we need to create a function and call it with an event in the **button** element.

  function handleClick() {

    setHeadingText("Submitted");

  }

1. And we need two **functions** (one to change **isMousedOver** to **true** and other to change it to “**false**”).

|  |  |
| --- | --- |
| function handleMouseOver() {      setMouseOver(true);    } | function handleMouseOut() {      setMouseOver(false);    } |

1. We need to assign **conditional** **rendering** to the background color of the button, depending on the state of **isMousedOver**.

        style={{ backgroundColor: isMousedOver ? "black" : "white" }}

1. Finally, we need to assign **event** **handlers** to the **button** element to call the functions when they are triggered.

      <button

        style={{ backgroundColor: isMousedOver ? "black" : "white" }}

        onClick={handleClick}

        onMouseOver={handleMouseOver}

        onMouseOut={handleMouseOut}

      >Submit

      </button>

See the full code below:

import React from "react";

function App() {

  const [headingText, setHeadingText] = React.useState("Hello");

  const [isMousedOver, setMouseOver] = React.useState(false);

  function handleClick() {

    setHeadingText("Submitted");

  }

  function handleMouseOver() {

    setMouseOver(true);

  }

  function handleMouseOut() {

    setMouseOver(false);

  }

  return (

    <div className="container">

      <h1>{headingText}</h1>

      <input type="text" placeholder="What's your name?" />

      <button

        style={{ backgroundColor: isMousedOver ? "black" : "white" }}

        onClick={handleClick}

        onMouseOver={handleMouseOver}

        onMouseOut={handleMouseOut}

      >Submit

      </button>

    </div>

  );

}

export default App;

**Event Handler and useState Example**

The code below renders a list with buttons.

1. We start be setting the people state value equal to the data array.
2. To **remove an item** (person) from the array, we use the filter function. Then when someone clicks in the “remove” button, it will run the function inside a callback function. **As the state value people changes, the page re-renders**. Note that we have to pass the id, so the function really needs to be inside a callback function, other wise it would run automatically without clicking on it.

const removeItem = (id) => {

  setPeople(people.filter((person) => person.id != id));

};

<button

   type="button"

   onClick={() => removeItem(id)}

>

remove

</button>

1. In order to clear the items, we just need to set the people array equal to an empty array, using the setPeople function. Not that there is no argument to be passed this time so the function doesn’t need to be inside a callback function.

const clearAllItems = () => {

  setPeople([]);

};

<button

 type="button"

 style={{ marginTop: "2rem" }}

 className="btn"

 onClick={clearAllItems}

>

  clear items

</button>

See the full code below:

|  |  |
| --- | --- |
| **03-useState-array.jsx** | |
| import { data } from "../../../data";  import React from "react";  const UseStateArray = () => {    const [people, setPeople] = React.useState(data);    const removeItem = (id) => {      setPeople(people.filter((person) => person.id != id));    };    const clearAllItems = () => {      setPeople([]);    };    return (      <div>        {people.map((person) => {          const { id, name } = person;          return (            <div key={id}>              <h4>{name}</h4>              <button                type="button"                onClick={() => removeItem(id)}              >                remove              </button>            </div>          );        })}        <button          type="button"          style={{ marginTop: "2rem" }}          className="btn"          onClick={clearAllItems}        >          clear items        </button>      </div>    );  };  export default UseStateArray; | |
| **data.js** | **Result** |
| export const data = [    { id: 1, name: 'john' },    { id: 2, name: 'peter' },    { id: 3, name: 'susan' },    { id: 4, name: 'anna' },  ]; |  |

**useState Gotcha**

In the example below, notice that the value that is rendered is more updated than the one logged in the console, i.e. they are not synchronous. There might be a case where in order the functionality to work that value needs to be updated.

|  |  |
| --- | --- |
| import { useState } from "react";  const UseStateGotcha = () => {    const [value, setValue] = useState(0);    const handleClick = () => {      setValue(value + 1);      console.log(value);    };    return (      <div>        <h2>{value}</h2>        <button          type="button"          className="btn"          onClick={handleClick}        >          increase        </button>      </div>    );  };  export default UseStateGotcha; |  |

If you want to update the state immediately and synchronously, you can pass a function to **setState** that receives the previous state as an argument and returns the new state.

The fact that the argument of the function is equal to the previous state value is a functionality provided by React. For example:

  const handleClick = () => {

    setValue((currentState) => {

      const newState = currentState + 1;

      return newState;

    });

**useState using objects**

The following program is written so there are 3 hooks for 3 state values:

import { useState } from "react";

const UseStateObject = () => {

  const [name, setName] = useState("peter");

  const [age, setAge] = useState(24);

  const [hobby, setHobby] = useState("read books");

  const displayPerson = () => {

    setName("john");

    setAge(28);

    setHobby("scream at the computer");

  };

  return (

    <div>

      <h3>{name}</h3>

      <h3>{age}</h3>

      <h4>Enjoys : {hobby}</h4>

      <button className="btn" onClick={displayPerson}>

        show john

      </button>

    </div>

  );

};

export default UseStateObject;

|  |  |  |
| --- | --- | --- |
|  |  |  |

However, wouldn’t it be easier if we could group then together in one object? See the example below.

import { useState } from "react";

const UseStateObject = () => {

  const [person, setPerson] = useState({

    name:"peter",

    age: 24,

    hobby: "read books"

  })

  const displayPerson = () => {

    setPerson({name: "john", age: 28, hobby: "scream at the computer"})

  };

  return (

    <div>

      <h3>{person.name}</h3>

      <h3>{person.age}</h3>

      <h4>Enjoys : {person.hobby}</h4>

      <button className="btn" onClick={displayPerson}>

        show john

      </button>

    </div>

  );

};

export default UseStateObject;

if, in the **displayPerson** function, we only write one of the properties, the other ones will be **undefined**.

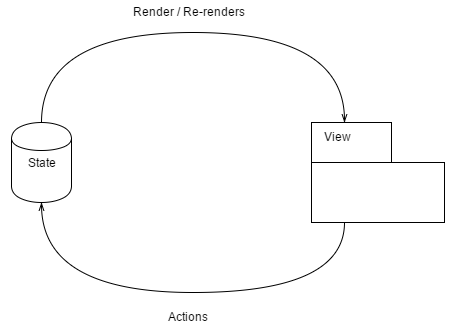
|  |  |
| --- | --- |
| const displayPerson = () => {      setPerson({name: "john"})    }; |  |

We can solve this by using the **spread operator**, which grabs all the previous values and then just replaces the ones specified after.

|  |  |
| --- | --- |
| const displayPerson = () => {      setPerson({...person, name: "john"})    }; |  |

**Mutating States**

React follows Unidirectional Data Flow, meaning that the data flow inside react should and will be expected to be in a circular path.



To make React work like this, developers made React similar to functional programming. The rule of thumb of functional programming is **immutability**.

**How does the unidirectional flow works?**

**States** are a data store which contains the data of a component. The view of a component renders based on the state. When the view needs to change something on the screen, that value should be supplied from the store.

To make this happen, React provides **setState()** function which **takes in the new state** and, **compares and merge over the previous state** and **adds the new state to the state data store**.

**Whenever the data in the state store changes, react will trigger a re-render with the new state which the view consumes and shows it on the screen.** This cycle will continue throughout the component's lifetime.

If you see the above steps, it clearly shows a lot of things are happening behind when you change the state. So, when you mutate the state directly and call **setState()** with an empty object, the previous state will be polluted with your mutation. Due to which, the shallow compare and merge of two states will be disturbed or won't happen, because you'll have only one state now. This will disrupt all the React's Lifecycle Methods. As a result, your app will **behave abnormal** or **even crash**.

And another downside of mutation of Objects and Arrays in JavaScript is, when you assign an object or an array, you're just making a reference of that object or that array. When you mutate them, all the reference to that object or that array will be affected. React handles this in an intelligent way in the background and simply give us an API to make it work.

**Most common errors done when handling states in React**

// original state

this.state = {

  a: [1,2,3,4,5]

}

// changing the state in react

// need to add '6' in the array

// bad approach

const b = this.state.a.push(6)

this.setState({

  a: b

})

In the above example, **this.state.a.push(6)** will mutate the state directly. Assigning it to another variable and calling **setState** is same as what's shown below. As we mutated the state anyway, there's no point assigning it to another variable and calling **setState** with that variable.

// same as

this.state.a.push(6)

this.setState({})

Many people do this. **This is so wrong**. This breaks the beauty of React and is bad programming practice.

So, what's the best way to handle states in React?

When you need to change 'something' in the existing state, first get a copy of that 'something' from the current state.

// original state

this.state = {

  a: [1,2,3,4,5]

}

// changing the state in react

// need to add '6' in the array

// create a copy of this.state.a

// you can use ES6's destructuring

const currentStateCopy = [...this.state.a]

Now, mutating **currentStateCopy** won't mutate the original state. Do operations over **currentStateCopy** and set it as the new state using **setState()**.

currentStateCopy.push(6)

this.setState({

  a: currentStateCopy

})

By doing this, all the references of **this.state.a** won't get affected until we use **setState**. This gives you control over your code and this'll help you write elegant test and make you confident about the performance of the code in production.

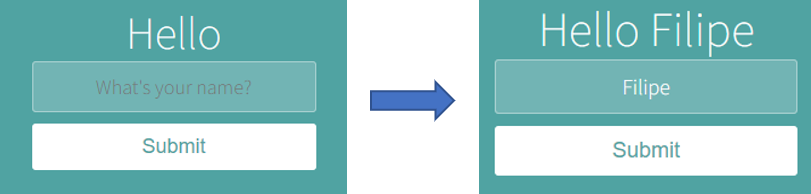
So, why can't I directly modify a component's state?

Well, you can. But, you need to face the following consequences.

1. When you scale, you'll be writing unmanageable code.
2. You'll lose control of state across components.
3. Instead of using React, you'll be writing custom codes over React.

**React Forms – Controlled Input Approach**

Imagine we want to up add the submitted name to the heading like in the image below.



So, we have **2 states**: one to store and set the **text** **input**, and another one to store and set the **heading** **text**.

  const [name, setName] = useState("");

  const [headingText, setHeading] = useState("");

We have a function that is triggered each time the event **onChange** happens. This function will update the **name** **input** **field** **value** and store it in the **name** **variable**. In this case, **event** is pointing to the **HTML element that triggered the event**, in this case the **input field**. So, **event.target.value** will get the value of the input field.

  function handleChange(event) {

    setName(event.target.value);

  }

The input field is **controlled** because React sets the name state value from the state <input value={name}/>. When the user types into in input field, the **onChange** handler updates the state with the input’s value accessed from the event object **event.target.value**. So, name is the only source of truth. Each time you need to access the value entered by the user in the input field, just read the name state variable.

<input

  onChange={handleChange}

  type="text"

  placeholder="What's your name?"

  value={name}

/>

When we click the **submit** button, we want it to **delete the input field** and send the it to the **heading**. When using forms and we have a submit button, the page refreshes, which means it will delete the name that was added to the heading. We can prevent this by using the line **event.preventDefault()**.

function handleClick(event) {

  setHeading(name);

  event.preventDefault();

  setName("");

}

So, the generic code for a controlled input in a Form looks like this:

import React, { useState } from "react";

function App() {

  const [name, setName] = useState("");

  const [headingText, setHeading] = useState("");

  function handleChange(event) {

    setName(event.target.value);

  }

  function handleClick(event) {

    setHeading(name);

    event.preventDefault();

    setName("");

  }

  return (

    <div className="container">

      <h1>Hello {headingText}</h1>

      <form onSubmit={handleClick}>

        <input

          onChange={handleChange}

          type="text"

          placeholder="What's your name?"

          value={name}

        />

        <button type="submit">Submit</button>

      </form>

    </div>

  );

}

export default App;

Basically, whatever we're going to be typing, this is going to be added to the state value.

Then whenever you submit the form, you just grab that state value and do whatever you need to do, whether that is to post some data on a server or to set up some kind of functionality.

**Multiple Returns**

**Introduction**

A function can have multiple returns, depending or if a certain condition is met. Below is a typical example where, while we are waiting for the data to be fetched (which in this case that waiting time is simulated with a setTimeout function), we display a Loading message on the screen.

So, if isLoading is true, then it will return the **Loading** heading. If it’s false, the function will return whatever is in the app.

import { useEffect, useState } from 'react';

const MultipleReturnsBasics = () => {

  // while fetching data

  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {

    setTimeout(() => {

      // done fetching data

      setIsLoading(false);

    }, 3000);

  }, []);

  // can return entire app

  if (isLoading) {

    return <h2>Loading...</h2>;

  }

  return <h2>My App</h2>;

};

export default MultipleReturnsBasics;

**Fetching Data**

The example below is a typical case of fetching data using **useEffect**.

1. We usually want to display a Loading screen (while we are waiting for the data) and an error screen (if the data cannot be fetched).

  const [isLoading, setIsLoading] = useState(true);

  const [isError, setIsError] = useState(false);

  if (isLoading) {

    return <h2>Loading...</h2>;

  }

  if (isError) {

    return <h2>There was an error...</h2>;

  }

1. So, we want to set up a **useEffect** to fetch the data.

If we are NOT successful, we want the Loading message to disappear and the error message to show.

if (!resp.ok) {

setIsError(true);

  setIsLoading(false);

  return;

}

If we are successful, we just want to hide the loading message.

setIsLoading(false);

So, the full code would look like:

import { useEffect, useState } from 'react';

const url = 'https://api.github.com/users/QuincyLarson';

const MultipleReturnsFetchData = () => {

  const [isLoading, setIsLoading] = useState(true);

  const [isError, setIsError] = useState(false);

  const [user, setUser] = useState(null);

  useEffect(() => {

    const fetchUser = async () => {

      try {

        const resp = await fetch(url);

        // console.log(resp);

        if (!resp.ok) {

          setIsError(true);

          setIsLoading(false);

          return;

        }

        const user = await resp.json();

        setUser(user);

      } catch (error) {

        setIsError(true);

        // console.log(error);

      }

      // hide loading

      setIsLoading(false);

    };

    fetchUser();

  }, []);

  // order matters

  // don't place user JSX before loading or error

  if (isLoading) {

    return <h2>Loading...</h2>;

  }

  if (isError) {

    return <h2>There was an error...</h2>;

  }

  const { avatar\_url, name, company, bio } = user;

  return (

    <div>

      <img

        style={{ width: '150px', borderRadius: '25px' }}

        src={avatar\_url}

        alt={name}

      />

      <h2>{name}</h2>

      <h4>works at {company}</h4>

      <p>{bio}</p>

    </div>

  );

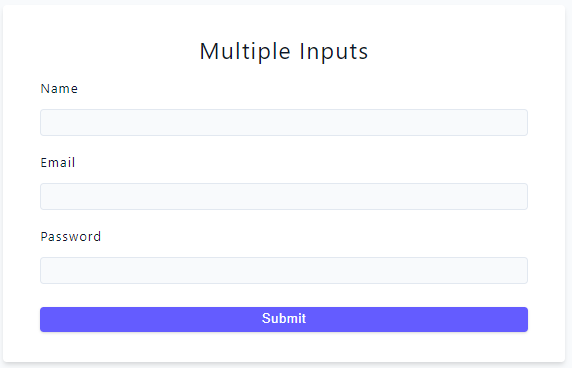
};

export default MultipleReturnsFetchData;

**Multiple Controlled Inputs**

Let's talk about how we can set up one state value for multiple inputs. This is not something you have to do, you can have multiple states for multiple inputs.

Consider the following input field.



You can set one state value that controls the inputs for each input field if that state value is an object:

  const [user, setUser] = useState({

    name: '',

    email: '',

    password: '',

  });

Since we only have one state value, we will only have one function. Now, when we setup the value in the input, we have to use dot notation to refer to that specific property.

   <input

     type='text'

     className='form-input'

     id='name'

**value={user.name}**

**onChange={handleChange}**

    />

But how can we access the actual value? For this, we have to use the name attribute. So, if you have this setup where you use a state as an object, you have to always use the name attribute.

   <input

     type='text'

     className='form-input'

     id='name'

**name='name'**

value={user.name}

     onChange={handleChange}

    />

So how do we now update the value? Through the handleChange function. We cannot use the following code.

  const handleChange = (e) => {

    setUser({ name:"" });

  };

That would just override the object and make it just have one property. We want to spread out again all the existing first values, and after that we want to dynamically update the property in state and set it equal to value. For that, we will use the name attribute, and that is why it is **so important that the name attribute matches the key name.**

  const handleChange = (e) => {

    setUser({ ...user, [e.target.name]: e.target.value });

  };

Now, the state value is equal to whatever is in the input field.

**Checkbox Input**

When working with checkboxes, the value of the checked attribute (true or false) is controlled by a state value. By doing this, the value of the checkbox (true or false) is synchronized with the corresponding state value.

const [shipping, setShipping] = useState(false);

  const handleShipping = (e) => {

    setShipping(e.target.checked);

  };

  return (

    <div>

      <form className='form'>

        <h4>Other Inputs</h4>

        {/\* name \*/}

        <div className='form-row' style={{ textAlign: 'left' }}>

          <input

            type='checkbox'

            checked={shipping}

            id='shipping'

            name='shipping'

            onChange={handleShipping}

          />

          <label htmlFor='shipping'> Free Shipping </label>

        </div>

        <div className='form-row' style={{ textAlign: 'left' }}>

          <label htmlFor='framework' className='form-label'>

            Framework

          </label>

        </div>

        <button type='submit' className='btn btn-block'>

          submit

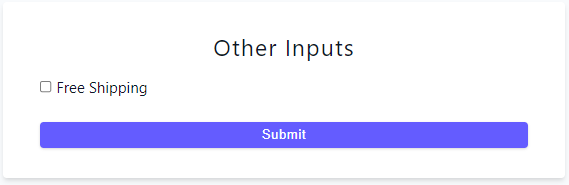
        </button>

      </form>

    </div>

  );

};



**Select Input**

When it comes to select input, we usually get a list of possible selections from the API. For example:

const frameworks = ["react", "angular", "vue", "svelte"];

We then have to iterate over that list to get the elements. Note that in this case we don’t have the parameter, so we are using the framework name itself as a key. The key just needs to be a unique value, which in this case it is (the framework names don’t repeat). In reality, every API should have IDs for each element.

{frameworks.map((framework) => {

    return <option key={framework}>{framework}</option>;

)}

We also need to synchronize the value selected with a state value. We do that by setting the value of the select input equal to the state value, and then using the state function to update the state so it is equal to the value.

import { useState } from "react";

const frameworks = ["react", "angular", "vue", "svelte"];

const OtherInputs = () => {

  const [framework, setFramework] = useState("react");

  const handleFramework = (e) => {

    setFramework(e.target.value);

  };

  return (

    <div>

      <form className="form">

        <h4>Other Inputs</h4>

        {/\* name \*/}

        <div className="form-row" style={{ textAlign: "left" }}>

          <label htmlFor="framework" className="form-label">

            Framework

          </label>

          <select

            name="framework"

            id="framework"

            value={framework}

            onChange={handleFramework}

          >

            {frameworks.map((framework) => {

              return <option key={framework}>{framework}</option>;

            })}

          </select>

        </div>

        <button type="submit" className="btn btn-block">

          submit

        </button>

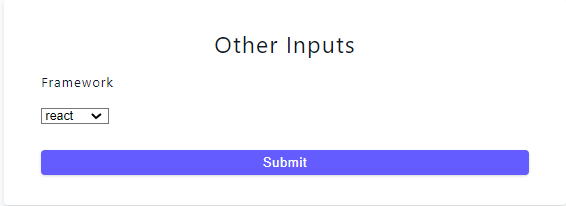
      </form>

    </div>

  );

};

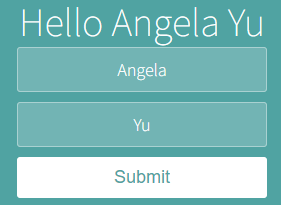
export default OtherInputs;



**Changing Complex State**

**Complex** **State** is a way of **managing** **the** **state** of Javascript objects where you might have to **retrieve** **the** **previous** **value** **of** **the** **object**.

Imagine the example below, where we have 2 input fields: **First** **Name** and **Last** **Name**.



We want to add them to the message in the top “Hello” as we type. Without using complex state, and just the previous knowledge, it would look something like this:

import React from "react";

function App() {

  const [fName, setFName] = React.useState("")

const [lName, setLName] = React.useState("")

  function updateFName(e) {

    setFName (e.target.value)

  }

  function updateLName(e) {

    setLName (e.target.value)

  }

  return (

    <div className="container">

      <h1>Hello {fName} {lName}</h1>

      <form>

        <input onChange={updateFName} name="fName" placeholder="First Name" />

        <input onChange={updateLName} name="lName" placeholder="Last Name" />

        <button>Submit</button>

      </form>

    </div>

  );

}

export default App;

Although it’s good practice, there are a few problems with this approach:

* There are so **many** **functions**.
* There are **two** **separate** **constants**.
* The **inputs** are **not** **associated** with each other.

So how can we improve this code?

1. We can use **useState** but instead of storing a value, we can store an object **fullName**. This object is going to have 2 properties: **fName** and **lName**. So now, we are actually inserting {fullName.fName} and {fullName.lName} to the Hello title.

  const [fullName, setFullName] = React.useState("{

    fName: "" ,

    lName: ""

  })

1. Instead of having two functions, we are going to have one functions that handle both names, starting by **destructuring** the **fullName** object, and hooking the two properties of it: **value** and **name**.

const {name, value} = e.target

Which means:

**name = e.target.name** (identifies which input field was changed)

**value = e.target.value** (identifies what the input field changed into)

1. Now what we want to do is being able to update the full name, but without deleting the previous properties. We have to **store** them somehow.

In the setting function **setFullName**, we are going to **pass an anonymous function as an argument**, and that function is going to have **prevValue** as an argument. **prevValue** contains the previous value of the **fullName** object. So, if we **console.log(prevValue)** right at the start, what we get is:

Object (fName:"", lName:"")

By having access to the previous value, that means we can **set the new value** for a property, but **keep the value of the other preperties** as they are. So, for example, if the **First Name** input field **fname** was changed, it will:

1. call the function **updateName**
2. update the value of the **fName** property with the value that was inserted in the input field that triggered the event (**fName**).
3. The previous value of **fullName** properties is stored by the argument **prevValue**.
4. So, the key that sets the value of the last name **lName** is set to **prevValue.lName**. That way, we keep its value.

  function updateName(e) {

    const { value, name } = e.target;

    setFullName(prevValue => {

      if (name === "fName") {

        return {

          fName: value,

          lName: prevValue.lName

        };

      } else if (name === "lName") {

        return {

          fName: prevValue.fName,

          lname: value

        };

      }

    });

  }

So, the final code looks like this:

import React, { useState } from "react";

function App() {

  const [fullName, setFullName] = useState({

    fName: "",

    lName: ""

  });

  function updateName(e) {

    const { value, name } = e.target;

    setFullName(prevValue => {

      if (name === "fName") {

        return {

          fName: value,

          lName: prevValue.lName

        };

      } else if (name === "lName") {

        return {

          fName: prevValue.fName,

          lname: value

        };

      }

    });

  }

  return (

    <div className="container">

      <h1>

        Hello {fullName.fName} {fullName.lName}

      </h1>

      <form>

        <input

          name="fName"

          onChange={ updateName }

          placeholder="First Name"

          value={fullName.fName}

        />

        <input

          name="lName"

          onChange={ updateName }

          placeholder="Last Name"

          value={fullName.lName}

        />

        <button>Submit</button>

      </form>

    </div>

  );

}

**The Spread Operator**

**Review**

The JavaScript spread operator (**...**) allows us to quickly copy all or part of an existing array or object into another array or object.

const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];

console.log(numbersCombined)

Resulting in [1, 2, 3, 4, 5, 6]

The spread operator is often used in combination with **destructuring**. Assign the first and second items from numbers to variables and put the rest in an array:

const numbers = [1, 2, 3, 4, 5, 6];

const [one, two, ...rest] = numbers;

console.log(numbers)

Resulting in [1, 2, 3, 4, 5, 6]

You can use the spread operator with objects as well.

const fullName = {

    fName: "James",

    lName: "Bond"

};

const user = {

    ...fullName,

    id:1,

    username: "jamesbond007"

};

console.log(user)

Resulting in:

{fName: 'James', lName: 'Bond', id: 1, username: 'jamesbond007'}

**Applying it in React**

We can then apply this principle to the previous example, saving a lot of lines of code:

1. Our **setFullName** function will call an anonymous function with **prevValue** as the argument (which stores the previous value of the objects properties.

2. Our anonymous function will return an object (note the curly braces **{}**), with the previous value, plus add the new **value** that was input in a specific **name** input field. Note that the **name** variable is inside **[]**, so JS doesn’t look at it as a string, but as a array.

|  |  |
| --- | --- |
| **Before Spread Operator:** | **After Spread Operator:** |
| function updateName(e) {  const { name, value } = e.target;      setFullName(prevValue => {        if (name === "fName") {          return {            fName: value,            lName: prevValue.lName          };        } else if (name === "lName") {          return {            fName: prevValue.fName,            lname: value          };        }      });    } | function updateName(e) {   const { name, value } = e.target;           setFullName(prevValue => {           return {             ...prevValue,             [name]: value           };         });       } |

See below an image of the result (on the left) and the hooks created, which we can find out by doing inspect 🡪 components.

|  |  |
| --- | --- |
|  |  |

import React, { useState } from "react";

function App() {

  const [fullName, setFullName] = useState({

    fName: "",

    lName: ""

  });

   function updateName(e) {

       const { name, value } = e.target;

       setFullName(prevValue => {

         return {

           ...prevValue,

           [name]: value

         };

       });

     }

  return (

    <div className="container">

      <h1>

        Hello {fullName.fName} {fullName.lName}

      </h1>

      <form>

        <input

          name="fName"

          onChange={updateName}

          placeholder="First Name"

          value={fullName.fName}

        />

        <input

          name="lName"

          onChange={updateName}

          placeholder="Last Name"

          value={fullName.lName}

        />

        <button>Submit</button>

      </form>

    </div>

  );

}

**Managing a Component Tree**

We know from the **React** **Props** lesson that a **React** **App** will **not end up with only one component App**. In reality, **each component will have its own separated file**.

For example, the code below represents a To Do List, with every component in the same **App.jsx** file. You can add items to this list by typing text in the input field and clicking the **Add** button.

We have got 2 hooks:

* One to store and update the input text (which will be updated each time the input field is changed)
* One to store and update the list of items (which will be updated each time the **Add** button is clicked).

The **addItem** function creates a new array with all the previous items, plus the new item which came from the **inputText** variable.

|  |
| --- |
| **App.jsx** |
| import React, { useState } from "react";  function App() {    const [inputText, setInputText] = useState("");    const [items, setItems] = useState([]);    function handleChange(event) {      setInputText(event.target.value);    }    function addItem() {      setItems(prevItems => {        return [...prevItems, inputText];      });      setInputText("");    }    return (      <div className="container">        <div className="heading">          <h1>To-Do List</h1>        </div>        <div className="form">          <input onChange={handleChange} type="text" value={inputText} />          <button onClick={addItem}>            <span>Add</span>          </button>        </div>        <div>          <ul>            {items.map(todoItem => (              <li>{todoItem}</li>            ))}          </ul>        </div>      </div>    );  }  export default App; |
| **Result** |
|  |

Now, imagine you want to put the list component in a separated file. It would look like:

import React from "react";

function ToDoItem(props) {

  return (

    <li>{props.text}</li>

  );

}

export default ToDoItem;

The **props.text** is the **text** **property** passed from the in the parent file **App.jsx**. Also, **{todoItem}** will now be replaced by the component **<ToDoItem/>**, as we have learned in the React Props lesson.

<ul>

{items.map((todoItem) => (

  <ToDoItem text={todoItem} />

))}

</ul>

And that’s it for stateless components. But now that we have learned about states, we will try to incorporate them into the **ToDoItem** component. So, imagine we want to be able to add a strike-though line when we click the item.

1. We need to create a hook with the initial value of **false** (in the beginning the item is not clicked):

const [clicked, setClicked] = React.useState(false)

1. We need a function that changes the value from **false** to **true** and vice-versa. So, it needs to remember the previous value of clicked and change it to the opposite. The function below returns the opposite value of the current **clicked** variable.

function handleClick() {

  setClicked(prevValue => {

    return !prevValue

  })

}

1. We need to apply **conditional** **rendering** to the **li** style and add an **Event** **Handler** that calls the function **handleClick** if the item is clicked.

|  |
| --- |
| <li    onClick={handleClick}    style={{textDecoration: clicked ? "line-through" : "none" }}>{props.text}  </li> |

So, the final code looks like this:

import React from "react";

function ToDoItem(props) {

const [clicked, setClicked] = React.useState(false)

function handleClick() {

  setClicked(prevValue => {

    return !prevValue

  })

}

return (

    <li

    onClick={handleClick}

    style={{textDecoration: clicked ? "line-through" : "none" }}>{props.text}</li>

  );

}

export default ToDoItem;

So, we now have the **clicked** state being managed inside our **ToDoItem**, and it is localized to this component.

But what if we wanted to delete it from our items array? How do we reach up from our **ToDoItem** into its parent (**App**) and somehow change this array? So now, instead of just changing the style of an item, we want to be able to change the entire array.

When we call a component in the **App** (parent), we cannot only pass variables as **props**, **we can also pass functions**:

<ul>

{items.map((todoItem) => (

  <ToDoItem

    text={todoItem}

    onChecked={deleteItem}

    />

))}

</ul>

And then call this function when the event **onClick** occurs in an item.

import React from "react";

function ToDoItem(props) {

  return (

    <li

    onClick={props.onChecked}>{props.text}</li>

  );

}

export default ToDoItem;

But how can we know which item was clicked and delete it from the array?

Remember, when we **map** through arrays in React, we should always have a **key**, which identifies the object with a unique **id** number.

One way to do this is by setting the **key** value equal to the **index** that the **map** function is looping through. Remember the syntax of a map function is:

**array.map(function(element, index))**

in which the **index** parameter is **optional**. But, it means we can have access to it and use it to assign key values to each item. So, the **ToDoItem** component in the **App.jsx** file will look like this:

<ul>

{items.map((todoItem, index) => (

  <ToDoItem

    key={index}

    id={index}

    text={todoItem}

    onChecked={deleteItem}

  />

))}

</ul>

And then we need to pass the id property as a **prop** to the **ToDoItem** component.

import React from "react";

function ToDoItem(props) {

  return <li onClick={props.onChecked(props.id)}>{props.text}</li>;

}

export default ToDoItem;

However, using the syntax above, because **onChecked** is called with the parenthesis, it means **it will be called as soon as the item is rendered**, which means it will delete the item as soon as it’s added.

So, we need to find a way of **calling the function only when it’s clicked**, i.e. without **()**. So, instead of calling the **onChecked** function, it will call an anonymous function which will call the **onChecked** function, and this anonymous function will only be called once the item is clicked.

import React from "react";

function ToDoItem(props) {

  return (

    <li

      onClick={() => {

        props.onChecked(props.id);

      }}

    >

      {props.text}

    </li>

  );

}

export default ToDoItem;

Now we just need to know how to delete an item from the array with the **deleteItem** function. We can do this using the **filter** method, which, remember, **takes a function as an input which will tell set the condition of the filter**. And that function will take the current value that is being filtered, as well as the optional index parameter, which again returns the index of the value being filtered.

**array.filter(function(currentValue, index))**

So, we need access to store the **prevItems** array to have access to the last array that was stored. We need to filter through it and find the **id** (or index) of the item that was clicked.

It will then return all the items of that array which **id** doesn’t match to the index of the item that was clicked.

function deleteItem(id) {

  setItems((prevItems) => {

    return prevItems.filter( (item, index) => {

      return index !== id

    })

  })

}

So, the final code will look like this:

|  |
| --- |
| **App.jsx** |
| import React, { useState } from "react";  import ToDoItem from "./ToDoItem";  function App() {    const [inputText, setInputText] = useState("");    const [items, setItems] = useState([]);    function handleChange(event) {      const newValue = event.target.value;      setInputText(newValue);    }    function addItem() {      setItems(prevItems => {        return [...prevItems, inputText];      });      setInputText("");    }    function deleteItem(id) {      setItems(prevItems => {        return prevItems.filter((item, index) => {          return index !== id;        });      });    }    return (      <div className="container">        <div className="heading">          <h1>To-Do List</h1>        </div>        <div className="form">          <input onChange={handleChange} type="text" value={inputText} />          <button onClick={addItem}>            <span>Add</span>          </button>        </div>        <div>          <ul>            {items.map((todoItem, index) => (              <ToDoItem                key={index}                id={index}                text={todoItem}                onChecked={deleteItem}              />            ))}          </ul>        </div>      </div>    );  }  export default App; |
| **ToDoItem.jsx** |
| import React from "react";  function ToDoItem(props) {    return (      <div        onClick={() => {          props.onChecked(props.id);        }}      >        <li>{props.text}</li>      </div>    );  }  export default ToDoItem; |

**Creating a React app with Vite**

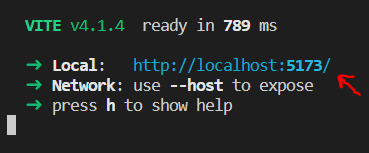
You need **Node.js** and **Hyper** installed in order to create a react app with **Vite**.

1. Open the Hyper.
2. Change the working directory to the directory you want to create the app in.
3. Copy + paste **npm create vite@latest my-react-app -- --template react**

**my-react-app** is just the name of your app, so choose whatever you want.

1. Open VS.
2. Type in the VS terminal **npm install** to install the dependencies.
3. Type in the VS terminal **npm run dev**

To see the website changes as you code it, click in the link provided in the terminal (*ctrl + click):*



In VITE, all the Javascript files must have the .jsx extension.

To build the application, run the command **npm build**.

**Converting HTML to React files**

1. Create a react app.
2. Copy and paste the HTML code inside the App component.
3. Delete comments or make them JS comments by putting them between {/\* *comment* \*/}
4. Replace all the **class** for **className** (use Ctrl + D to help selecting).
5. Create a components folder.
6. Separate the different elements into components (e.g. Navbar, Hero, Footer, etc).
7. Correct the local images source links. Instead of using the path directly in the source, use import.
8. Replace repetitive sections of data for iterations. For example, on the HTML, a navbar would look something like this, where the structure repeats itself.

<ul className="nav-links" id="nav-links">

  <li><a href="#home" className="nav-link">home</a></li>

  <li><a href="#about" className="nav-link">about</a></li>

  <li><a href="#services" className="nav-link">services</a></li>

  <li><a href="#tours" className="nav-link">tours</a></li>

</ul>

We could create a separate **data.js** file inside the **src** folder with the array below. This way, it’s very easy to add new elements to the Navbar component, we just need to add more elements to the **pageLinks** array.

export const pageLinks = [

  { id: 1, href: "#home", text: "home" },

  { id: 2, href: "#about", text: "about" },

  { id: 3, href: "#services", text: "services" },

  { id: 4, href: "#tours", text: "tours" },

];

And on the **Navbar** component:

<ul className="nav-links" id="nav-links">

  {pageLinks.map((link) => {

    return (

      <li key={link.key}>

        <a href={link.href} className="nav-link">

          {link.text}

        </a>

      </li>

    );

  })}

</ul>

We could of course also use **deconstruction** to access the properties:

<ul className="nav-links" id="nav-links">

  {pageLinks.map((link) => {

    const { id, href, text } = link;

    return (

      <li key={id}>

        <a href={href} className="nav-link">

          {text}

        </a>

      </li>

    );

  })}

</ul>

**React useEffect**

**Introduction**

The **useEffect** Hook allows you to perform side effects in your components. Some examples of side effects are: fetching data, directly updating the DOM, and timers.

**It allows you to have control of when a function does actually run**.

useEffect(<function>, <dependency>)

**useEffect** has two arguments:

* **callback function**: whatever is going to run. By default, it runs after every render.
* **dependency array (optional)**: dictates when the callback function is going to run.

If you don’t specify the dependency array as an argument, the code runs for every render and re-render by default. In the example below, setValue runs every time the button is clicked, because that makes the value state value change, and when a state value changes, it triggers re-rerender.

|  |  |
| --- | --- |
| **useEffect with no dependency array** | |
| import { useState, useEffect } from 'react';  const UseEffectBasics = () => {    const [value, setValue] = useState(0);    const sayHello = () => {      console.log('hello there');    };    sayHello();    useEffect(()=> {      console.log("hello from useEffect")    })    return (      <div>        <h1>value : {value}</h1>        <button className='btn' onClick={() => setValue(value + 1)}>          click me        </button>      </div>    );  };  export default UseEffectBasics; |  |

If the dependency array is just an empty array **[]**, then the code is only going to run in the initial render.

|  |  |
| --- | --- |
| **useEffect with empty dependency array** | |
| import { useState, useEffect } from 'react';  const UseEffectBasics = () => {    const [value, setValue] = useState(0);    const sayHello = () => {      console.log('hello there');    };    sayHello();    useEffect(()=> {      console.log("hello from useEffect")    },[])    return (      <div>        <h1>value : {value}</h1>        <button className='btn' onClick={() => setValue(value + 1)}>          click me        </button>      </div>    );  };  export default UseEffectBasics; |  |

If you put the value being updated inside the dependency array, the callback function is only going to run when that value is updated.

|  |  |
| --- | --- |
| **useEffect with the value being updated in the dependency array** | |
| import { useState, useEffect } from 'react';  const MultipleEffects = () => {    const [value, setValue] = useState(0);    const [secondValue, setSecondValue] = useState(0);    useEffect(() => {      console.log('hello from first useEffect');    }, [value]);    useEffect(() => {      console.log('hello from second useEffect');    }, [secondValue]);    return (      <div>        <h1>value : {value}</h1>        <button className='btn' onClick={() => setValue(value + 1)}>          value        </button>        <h1>second value : {secondValue}</h1>        <button className='btn' onClick={() => setSecondValue(secondValue + 1)}>          second value        </button>      </div>    );  };  export default MultipleEffects; |  |

You can have as many **useEffects** in your component as you want. So pretty much you can set up a **useEffect** for every smallest feature. Now, just because you can doesn't mean you should.

**React useEffect Fetch Data**

We cannot return a **promise** from a **callback function**. However, the **callback function** inside the **useEffect** can by an **async** function and thus return a **promise**.

The code below shows a generic example of how to fetch data with useEffect.

import { useState, useEffect } from 'react';

const url = 'https://api.github.com/users';

const FetchData = () => {

  const [users, setUsers] = useState([]);

  useEffect(() => {

    const fetchData = async () => {

      try {

        const response = await fetch(url);

        const users = await response.json();

        setUsers(users);

      } catch (error) {

        console.log(error);

      }

    };

    fetchData();

  }, []);

  return (

    <section>

      <h3>github users</h3>

      <ul className='users'>

        {users.map((user) => {

          const { id, login, avatar\_url, html\_url } = user;

          return (

            <li key={id}>

              <img src={avatar\_url} alt={login} />

              <div>

                <h5>{login}</h5>

                <a href={html\_url}>profile</a>

              </div>

            </li>

          );

        })}

      </ul>

    </section>

  );

};

export default FetchData;

**Short Circuit**

**“truthy” and “falsy” introduction**

In JavaScript, a value is considered "**truthy**" if it is evaluated to true when used in a boolean context. A value is considered "**falsy**" if it is evaluated to false when used in a boolean context.

Here is a list of values that are considered **falsy** in JavaScript:

* false
* 0 (zero)
* "" (empty string)
* null
* undefined
* NaN (Not a Number)
* All other values, including objects and arrays, are considered truthy.

For example:

const x = 'Hello';

const y = '';

const z = 0;

if (x) {

  console.log('x is truthy');

}

if (y) {

  console.log('y is truthy');

} else {

  console.log('y is falsy');

}

if (z) {

  console.log('z is truthy');

} else {

  console.log('z is falsy');

}

// Output:

// "x is truthy"

// "y is falsy"

// "z is falsy"

In this example, the variable x is a non-empty string, which is considered truthy, so the first if statement is executed. The variable y is an empty string, which is considered **falsy**, so the else block of the second if statement is executed. The variable **z** is the number 0, which is considered **falsy**, so the else block of the third if statement is executed.

**Short Circuit Introduction**

In JavaScript, **short-circuit** evaluation is a technique that allows you to use logical operators (such as **&&** and **||**) to perform **conditional evaluations in a concise way**.

The **&&** operator (logical AND) returns the first operand if it is "**falsy**", or the second operand if the first operand is "**truthy**". For example:

const x = 0;

const y = 1;

console.log(x && y); // Output: 0 (the first operand is falsy, so it is returned)

console.log(y && x); // Output: 0 (the second operand is falsy, so it is returned)

The **||** operator (logical OR) returns the first operand if it is "**truthy**", or the second operand if the first operand is "**falsy**". For example:

const x = 0;

const y = 1;

console.log(x || y); // Output: 1 (the first operand is falsy, so the second operand is returned)

console.log(y || x); // Output: 1 (the first operand is truthy, so it is returned)

Short-circuit evaluation can be useful in cases where you want to **perform a certain action only if a certain condition is met**, or you want to return a **default value if a certain condition is not met**. For example:

function displayName(name) {

  return name || 'Anonymous';

}

console.log(displayName('Pizza')); // Output: "Pizza"

console.log(displayName()); // Output: "Anonymous"

In this example, the **displayName()** function returns the name property of the user object if it exists, or "Anonymous" if the name property is not present. This is done using the **||** operator and short-circuit evaluation.

**React use case – toggling**

A typical use case for this is toggling, i.e. showing something depending if a certain state is **true** or **false**. In the example below, by clicking on the “toggle alert” button, it changes “showAlert” from true to false and vice-versa, which will hide or show the “hello world” alert.

import { useState } from 'react';

const ToggleChallenge = () => {

  const [showAlert, setShowAlert] = useState(false);

  return (

    <div>

      <button className='btn' onClick={() => setShowAlert(!showAlert)}>

        toggle alert

      </button>

      {showAlert && <Alert />}

    </div>

  );

};

const Alert = () => {

  return <div className='alert alert-danger'>hello world</div>;

};

export default ToggleChallenge;

**Ternary Operator Introduction**

In JavaScript, the **ternary operator** is a way to **concisely express a simple conditional statement**. Here is the basic syntax for using the ternary operator:

condition ? expression1 : expression2;

**If condition is truthy, the operator will return expression1. If condition is falsy, it will return expression2.**

**React use case – Ternary Operator**

A typical use case for the Ternary Operator is to show the Loading screen if the isLoading is true, or else show the full page. Or, as is it below, if the user exists (i.e. is logged in) display something. If not, display a login screen.

import { useState } from 'react';

const UserChallenge = () => {

  const [user, setUser] = useState(null);

  const login = () => {

    // normally connect to db or api

    setUser({ name: 'vegan food truck' });

  };

  const logout = () => {

    setUser(null);

  };

  return (

    <div>

      {user ? (

        <div>

          <h4>hello there, {user.name}</h4>

          <button className='btn' onClick={logout}>

            logout

          </button>

        </div>

      ) : (

        <div>

          <h4>Please Login</h4>

          <button className='btn' onClick={login}>

            login

          </button>

        </div>

      )}

    </div>

  );

};

export default UserChallenge;

**useEffect Cleanup Function**

**Introduction**

With **useEffect()**, although you may have a dependency array (e.g. empty array will only run the functionality on initial render), if you keep hiding and displaying a component, it will run every time the component is displayed.

For example, in the code below, each time <RandomComponent /> is displayed by toggling it, the program will log 'hmm, this is interesting'.

import { useEffect, useState } from 'react';

const CleanupFunction = () => {

  const [toggle, setToggle] = useState(false);

  return (

    <div>

      <button className='btn' onClick={() => setToggle(!toggle)}>

        toggle component

      </button>

      {toggle && <RandomComponent />}

    </div>

  );

};

const RandomComponent = () => {

  useEffect(() => {

      console.log('hmm, this is interesting');

  }, []);

  return <h1>hello there</h1>;

};

export default CleanupFunction;

This is not bad or good, just something you need to be aware. In some cases, you might not want it to happen.

**Cleanup function example**

If we had a setInterval function inside the RandomComponent, then it that means it would run every second while the component is mounted (showing on the screen). However, the reality is that when we toggle the component to hide it, the function keeps going. And, the more we toggle the component, the faster setInterval will run, because it’s never reset and stacks up with the previous one.

Because of this, it’s a good idea to run the cleanup function. Now, the setInterval will only run whenever the item is mounted.

|  |  |
| --- | --- |
| **Before Cleanup function** | **After Cleanup function** |
| const RandomComponent = () => {    useEffect(() => {      console.log("hmm, this is interesting");      setInterval(() => {        console.log("hello from interval");      }, 1000);    }, []);    return <h1>hello there</h1>;  }; | const RandomComponent = () => {    useEffect(() => {      console.log("hmm, this is interesting");      const intId = setInterval(() => {        console.log("hello from interval");      }, 1000);      return () => {        clearInterval(intId);      };    }, []);    return <h1>hello there</h1>;  }; |

**removeEventListener**

In the example below, each time we toggle the component, it will attach another event listener to the window, which can create some issues.

import { useEffect, useState } from "react";

const CleanupFunction = () => {

  const [toggle, setToggle] = useState(false);

  return (

    <div>

      <button className="btn" onClick={() => setToggle(!toggle)}>

        toggle component

      </button>

      {toggle && <RandomComponent />}

    </div>

  );

};

const RandomComponent = () => {

  useEffect(() => {

    const someFunc = () => {

      //some logic

    };

    window.addEventListener("scroll", someFunc);

  }, []);

  return <h1>hello there</h1>;

};

export default CleanupFunction;

A way around it is to use removeEventListener.

import { useEffect, useState } from "react";

const CleanupFunction = () => {

  const [toggle, setToggle] = useState(false);

  return (

    <div>

      <button className="btn" onClick={() => setToggle(!toggle)}>

        toggle component

      </button>

      {toggle && <RandomComponent />}

    </div>

  );

};

const RandomComponent = () => {

  useEffect(() => {

    const someFunc = () => {

      //some logic

    };

    window.addEventListener("scroll", someFunc);

    return () => window.removeEventListener("scroll", someFunc);

  }, []);

  return <h1>hello there</h1>;

};

export default CleanupFunction;

**Dealing with missing data when fetching**

Not every API comes perfectly organized when fetching data, so it’s a good idea to deal with it so the browser doesn’t throw errors at us. To do that, we need to understand about **Optional Chaining**, **Default Values** and **Dynamic Object Keys**.

**Optional Chaining**

**Optional Chaining** offers more straightforward syntax for working with **deeply nested properties**. In JavaScript, the optional chaining operator (**?.**) is a new feature that allows you to access properties of an object without worrying about whether the object or the property is null or undefined. It's a shorthand for a common pattern of **checking for null or undefined before accessing an object's property**.

For example, consider the code below, which accesses the **firstName** property:

const person = { name: { first: 'John', last: 'Doe' } };

console.log(person.name.first);

If the name property is **null** or **undefined**, **this code will throw an error**. To prevent this, we can use the optional chaining operator:

console.log(person?.name?.first);

Now, if the **person.name** is **null** or **undefined**, **this code will simply return undefined instead of throwing an error**. This makes the code more robust and readable.

This is useful if you’re iterating over some API and some object doesn’t have one of the properties you are after. For example, in the API below, the second object doesn’t have the **timezone** property.

const people = [

  {

    name: "bob",

    location: { street: "123 main street", timezone: { offset: "+7.00" } },

  },

  { name: "peter", location: { street: "123 Pine Street" } },

  {

    name: "susan",

    location: { street: "123 Apple street", timezone: { offset: "+9.00" } },

  },

];

So, if we try to iterate over it and access the offset property of every object like shown below, we are going to get an error and the application will stop.

person.forEach((person)=> {

  person.location.timezone.offset

})

We need to use the optional chaining, which checks if that property exists, and if it doesn’t return undefined, instead of an error that stops the application.

person.forEach((person)=> {

  person?.location?.timezone?.offset

})

**Default Values**

In JavaScript, when defining a function, you can specify **default values** for its parameters. This means that **if a caller of the function does not provide a value for a particular parameter, the default value will be used instead**. Default parameters are defined by assigning a value to the parameter in the function definition.

For example, consider the following function, which takes two parameters, x and y, and returns their sum:

function add(x, y) {

  return x + y;

}

If we call this function with only one argument, it will return **NaN** because y is undefined.

We can set default values for **x** and **y** as 0:

function add(x = 0, y = 0) {

  return x + y;

}

Now, if we call **add(3)**, the function will return 3, because the default value of 0 is used for the **y** parameter.

**Dynamic Object Keys**

**Dot notation**

Imagine we have the following object:

const person = {

  name: "john"

}

We can access the properties of the object:

console.log(person.name) // john

We can add new properties:

person.age = 25;

**Square Brackets Notation**

We can do the same thing with the square brackets notation.

console.log(person["name"]) //john

**Using default values and optional chaining**

It is very common for API’s to come nested or with missing properties/objects like the example below.

export const people = [

  { id: 1, name: "bob", nickName: "Stud Muffin" },

  { id: 2, name: "peter" },

  {

    id: 3,

    name: "oliver",

    images: [

      {

        small: {

          url: "https://res.cloudinary.com/diqqf3eq2/image/upload/ar\_1:1,bo\_5px\_solid\_rgb:ff0000,c\_fill,g\_auto,r\_max,w\_1000/v1595959121/person-1\_aufeoq.jpg",

        },

      },

    ],

  },

  { id: 4, name: "david" },

];

In the code below, we use a **default value** for the nickname, so every time we search for nickname, it will default to shakeAndBake.

Also, we use **optional chaining** to access the url property because not every object has it. And if it doesn’t find the img, we set a default image avatar using the || operator. This is another option to set a default value.

import avatar from '../../../assets/default-avatar.svg';

const Person = ({ name, nickName = 'shakeAndBake', images }) => {

  // before optional chaining

  // const img =

  //   (images && images[0] && images[0].small && images[0].small.url) || avatar;

  const img = images?.[0]?.small?.url || avatar;

  return (

    <div>

      <img src={img} alt={name} style={{ width: '50px' }} />

      <h4>{name} </h4>

      <p>Nickname : {nickName}</p>

    </div>

  );

};

export default Person;

**Setting up keys dynamic with brackets notation**

We can **set up keys dynamically by using the square brackets notation**. In the example below, we define a variable appState, which has the same **name** as the **key** in the app object.

So, what happens is, whatever is the value of the appState variable, that will be the key name in the app object.

let appState = "loading";

const app = {

  [appState]: true,

};

console.log(app) // {loading: true}

We can also add new keys with this notation.

let appState = "loading";

let keyName = "computer";

const app = {

  [appState]: true,

};

app[keyName] = "apple";

console.log(app); // {loading: true, computer: "apple"}

**Useful application**

We create a function that takes 2 arguments (key and value) and use it to update the property values of various objects:

const state = {

  name: "",

  job: "",

};

function updateState(key, value) {

  state[key] = value;

}

updateState("name", "john");

updateState("job", "developer");

console.log(state); // {name: "john", job: "developer"}

**Folder Structure for big projects**

When you have a big project, a good idea is to create a folder for each components and, inside that folder, put every file related to that component (**.jsx**, **.css**, **.html**, etc).

This kind of structure only makes sense in big projects! You don’t have to use it all the time. For smaller projects you could have all your components in the same folder called “components”.

**Separate components with Index.jsx import**

See the example below where we create a Navbar component.

**App.jsx** is used to invoke the **Navbar** component. However, in reality, App.jsx has access to the Navbar component through the Index.jsx imported right in the first line. When no specific file is specified in the import file path (which is the case), it will import the index.jsx file by default (this is a node thing), which by itself is exporting the Navbar component.

|  |  |
| --- | --- |
| **Index.jsx** | **Navbar.jsx** |
| export { default } from './Navbar'; | import './Navbar.css';  const Navbar = () => {    return (      <div className='navbar'>        <h2>navbar component</h2>      </div>    );  };  export default Navbar; |
| **App.jsx** | **Navbar.css** |
| import Navbar from ".tutorial/04-project-structure/starter/Navbar"  function App() {      return (          <div className="container">              <Navbar>          </div>      );  }  export default App; | .navbar {    background: blue;    color: white;  } |

**Separate components with Index.jsx import**

A way to organize different pages is by importing and exporting the pages in the index.jsx, and then import index.jsx in the App.jsx

|  |  |
| --- | --- |
| **Index.jsx** | **Home.jsx** |
| import Home from './Home';  import About from './About';  export { Home, About }; | const Home = () => {    return <div>Home Page</div>;  };  export default Home; |
| **About.jsx** | **App.jsx** |
| const About = () => {    return <div>About Page</div>;  };  export default About; | import Navbar from ".tutorial/04-project-structure/starter/Navbar"  function App() {      return (          <div className="container">              <Home />              <About/>          </div>      );  }  export default App; |

**useRef**

**useRef** is a lot like **useState**, so it preserves the value between the renders. But the difference is that updating use **useRef** **does not trigger re-render**. For example, consider the following form:

import { useEffect, useRef, useState } from 'react';

const UseRefBasics = () => {

  const [value, setValue] = useState(0);

  const handleSubmit = (e) => {

    e.preventDefault();

  };

  return (

    <div>

      <form className='form' onSubmit={handleSubmit}>

        <div className='form-row'>

          <label htmlFor='name' className='form-label'>

            Name

          </label>

          <input type='text' id='name' className='form-input' />

        </div>

        <button type='submit' className='btn btn-block'>

          submit

        </button>

      </form>

      <h1>value : {value}</h1>

      <button onClick={() => setValue(value + 1)} className='btn'>

        increase

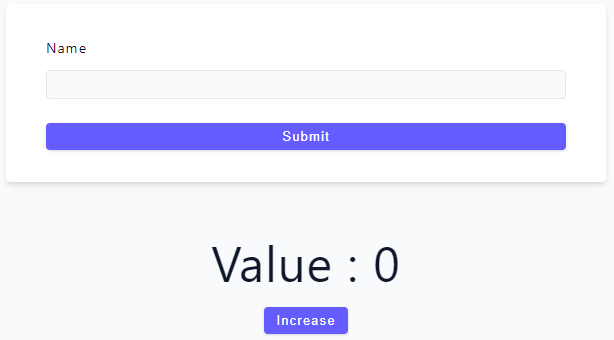
      </button>

    </div>

  );

};

export default UseRefBasics;



If we create a variable that stores a null, you’ll see it creates an object, with a **current** property of null.

|  |  |
| --- | --- |
| const refContainer = useRef(null);  console.log(refContainer); |  |

Once we have this in place, we have 2 ways how we can set this value, because of course you don’t want to always keep it as null:

1. We can use this **refContainer** and set it equal to any of the DOM nodes.
2. We can set it ourselves, using some kind of functionality

**Approach 1: refContainer**

We are going to try the first approach first.

1. First, we have to add a **ref** attribute to the input and set it equal to the **refContainer** variable.

<input

   type="text"

   id="name"

   className="form-input"

   ref={refContainer}

/>

1. Now we don’t see anything, because **refContainer** **runs during the initial render**. So, in order to see the actual value in the console, we have to use **useEffect**, which runs after the initial render. Note that we now have access to the DOM node (in this case the input).

const UseRefBasics = () => {

const [value, setValue] = useState(0);

const refContainer = useRef(null);

console.log(refContainer); //Happens during the initial render

useEffect(() => {

  console.log(refContainer);//Happens after the initial render

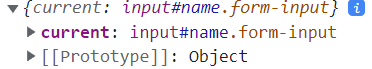
});

const handleSubmit = (e) => {

  e.preventDefault();

  const name = refContainer.current.value;

};



1. We now can grab the value by using **value** property.

|  |  |
| --- | --- |
| const handleSubmit = (e) => {      e.preventDefault();      const name = refContainer.current.value;      console.log(name);    }; |  |

**Approach 2: changing the value of useRef**

Now let’s take a look at our second example, where we change the value of **useRef**.

1. Let’s start by creating a value with useRef and set false as the default value. This again creates an object isMounted with a **current** property which is equal to false.

|  |  |
| --- | --- |
| const isMounted = useRef(false); |  |

1. Now if we add a useEffect and add the state value (the number after “Value” in the image below) in the dependency array, every time we change the value (increase it), the functionality inside the useEffect is going to run. As you can see in the image below, we clicked twice, but it logged 3 times, because it logged in the initial render.

|  |  |
| --- | --- |
| useEffect(() => {    console.log(isMounted);  }, [value]); |  |

What if we don’t want to run it in the initial render, but we want to run it after every time the value changes? To do that, we need to check the value of isMounted. If it’s false, we set it equal to true. Then if it’s true, we run the functionality. Note that it only ran twice, because in the initial render it was false.

|  |  |
| --- | --- |
| useEffect(() => {    if (!isMounted.current) {      isMounted.current = true;      return;    }  console.log(isMounted);  }, [value]); |  |

**getBoundingClientRect()–Width/Height (Vanilla JS)**

You can access the width and height of the whole window by doing:

window.innerHeight //gives us the height of the window

window.innerWidth //gives us the width of the window

The **getBoundingClientRect()** method returns the **size** of an element and its **position relative to the viewport.** It returns a DOMRect object with eight properties: left, top, right, bottom, x, y, width, height.

The scrolling that has been done is taken into account. This means that the rectangle's edges (top, left, bottom, and right) change their values every time the scrolling position changes.

const box = document.querySelector('box');

box.getBoundingClientRect()

One useful case for this is that you can detect if a certain element is off the screen if the position values start being negative, and then do something about it.

**Custom Hooks**

**Custom Hooks** are great for abstracting some functionality, basically to have **less lines of code in the component**. And as a result, it's going to be **easier to manage our component**. Also, once we have custom hook, we can bring it from project to project, which of course **speeds up our workflow**.

Rule of thumb, quite often **useEffect** is a great option for custom hook. All the hooks rules apply to custom hooks (for example, they need to start with **use**).

**Example - Toggle**

Consider the following code where we are using **useEffect** to toggle between the “some stuff” text.

|  |  |
| --- | --- |
| import { useState } from 'react';  const ToggleExample = () => {    const [show, setShow] = useState(false);    return (      <div>        <h4>toggle custom hook</h4>        <button className='btn' onClick={() => setShow(!show)}>          toggle        </button>        {show && <h4>some stuff</h4>}      </div>    );  };  export default ToggleExample; | |
|  |  |

Instead of using **useState** hook, we could create a custom hook that **stores and changes the state value of show**. Note that the hook could also return an object with curly braces **{}** instead of an array, both methods work. You would just have to correct it when invoking the hook.

const {show, toggle} = useToggle(false);

|  |  |
| --- | --- |
| **useToggle.jsx** | **Toggle.jsx** |
| import { useState } from "react";  const useToggle = (defaultValue) => {    const [show, setShow] = useState(defaultValue);    const toggle = () => {      setShow(!show);    };    return [show, toggle];  };  export default useToggle; | import useToggle from "./useToggle";  const ToggleExample = () => {    const [show, toggle] = useToggle(false);    return (      <div>        <h4>toggle custom hook</h4>        <button className="btn" onClick={toggle}>          toggle        </button>        {show && <h4>some stuff</h4>}      </div>    );  };  export default ToggleExample; |

**Example – Fetch User**

Consider the code below where we are fetching a user from GitHub.

import { useEffect, useState } from 'react';

const url = 'https://api.github.com/users/QuincyLarson';

const FetchData = () => {

  const [isLoading, setIsLoading] = useState(true);

  const [isError, setIsError] = useState(false);

  const [user, setUser] = useState(null);

  useEffect(() => {

    const fetchUser = async () => {

      try {

        const resp = await fetch(url);

        // console.log(resp);

        if (!resp.ok) {

          setIsError(true);

          setIsLoading(false);

          return;

        }

        const user = await resp.json();

        setUser(user);

      } catch (error) {

        setIsError(true);

        // console.log(error);

      }

      // hide loading

      setIsLoading(false);

    };

    fetchUser();

  }, []);

  // order matters

  // don't place user JSX before loading or error

  if (isLoading) {

    return <h2>Loading...</h2>;

  }

  if (isError) {

    return <h2>There was an error...</h2>;

  }

  const { avatar\_url, name, company, bio } = user;

  return (

    <div>

      <img

        style={{ width: '100px', borderRadius: '25px' }}

        src={avatar\_url}

        alt={name}

      />

      <h2>{name}</h2>

      <h4>works at {company}</h4>

      <p>{bio}</p>

    </div>

  );

};

export default FetchData;



We can move isLoading, isError and user states values to a custom hook. The **useFetchPerson** custom hook would **take the state values**, **fetch the user from the API** and **change the state values** accordingly to the response it gets.

The **FetchPerson** component would **grab those state values** and handle what to **display** depending on their values.

|  |  |
| --- | --- |
| **useFecthPerson.jsx** | **FecthPerson.jsx** |
| import { useState, useEffect } from "react";  const useFetchPerson = (url) => {    const [isLoading, setIsLoading] = useState(true);    const [isError, setIsError] = useState(false);    const [user, setUser] = useState(null);    useEffect(() => {      const fetchUser = async () => {        try {          const resp = await fetch(url);          if (!resp.ok) {            setIsError(true);            setIsLoading(false);            return;          }          const user = await resp.json();          setUser(user);        } catch (error) {          setIsError(true);        }        // hide loading        setIsLoading(false);      };      fetchUser();    }, []);    return { isLoading, isError, user };  };  export default useFetchPerson; | import useFetchPerson from "./useFetchPerson";  const url = "https://api.github.com/users/QuincyLarson";  const FetchData = () => {    const { isLoading, isError, user } = useFetchPerson(url);    // order matters    // don't place user JSX before loading or error    if (isLoading) {      return <h2>Loading...</h2>;    }    if (isError) {      return <h2>There was an error...</h2>;    }    const { avatar\_url, name, company, bio } = user;    return (      <div>        <img          style={{ width: "100px", borderRadius: "25px" }}          src={avatar\_url}          alt={name}        />        <h2>{name}</h2>        <h4>works at {company}</h4>        <p>{bio}</p>      </div>    );  };  export default FetchData; |

We can even make the **useFecthUser** hook more generic. For example, instead of using the word **user**, use **data**. And then you can use this hook in various projects every time you want to fetch data.

Chances are we'll probably want to fetch more than just a user. For example, we want to fetch an array or something along those lines.

**Context API**

**Prop Drilling**

Prop drilling is the process of **passing data from one component via several interconnected components to the component that needs it**. This approach has various disadvantages:

* Code harder to read
* Increase complexity of the component tree
* Difficult to manage state and props in large applications

For example, see below how the user and logout state values have to be passed from **Navbar** to **NavbarLinks**, and the from **NavbarLinks** to **UserContainer**.

|  |  |
| --- | --- |
| **App.jsx** | **Navbar.jsx** |
| import Starter from "./tutorial/09-context-api/starter/Navbar.jsx";  function App() {    return (      <div className="container">        <Starter />      </div>    );  }  export default App; | import UserContainer from "./UserContainer";  const NavLinks = ({ user, logout }) => {    return (      <div className="nav-container">        <ul className="nav-links">          <li>            <a href="#">home</a>          </li>          <li>            <a href="#">about</a>          </li>        </ul>        <UserContainer user={user} logout={logout} />      </div>    );  };  export default NavLinks; |
| **NavLinks.jsx** | **UserContainer.jsx** |
| import { useState } from "react";  import NavLinks from "./NavLinks";  const Navbar = () => {    const [user, setUSer] = useState({ name: "bob" });    const logout = () => {      setUSer(null);    };    return (      <nav className="navbar">        <h5>CONTEXT API</h5>        <NavLinks user={user} logout={logout} />      </nav>    );  };  export default Navbar; | const UserContainer = ({ user, logout }) => {    return (      <div className="user-container">        {user ? (          <>            <p>Hello There, {user?.name?.toUpperCase()}</p>            <button className="btn" onClick={logout}>              logout            </button>          </>        ) : (          <p>Please login</p>        )}      </div>    );  };  export default UserContainer; |
|  | |
| Logout Click | |
|  | |

**Context API**

Effectively, we're looking for a way to bypass multiple components that we have sitting in between the main one, the parent, and then whichever component wants to use those values.

To do this, we need to use **Context API**. Basically, we need to **create a context in the parent file**, in this case **Navbar.jsx** (the component which is going to provide those values). This gives us access to **Consumer** and **Provider** components.

export const NavbarContext = createContext();

console.log(NavbarContext);



We then want to wrap the return of that provider. Whatever you pass in the value prop in that component, you’re going to have access to it anywhere in that tree. We can then pass the user and logout values inside an object:

const Navbar = () => {

  const [user, setUSer] = useState({ name: "bob" });

  const logout = () => {

    setUSer(null);

  };

  return (

    <NavbarContext.Provider value={{ user, logout }}>

      <nav className="navbar">

        <h5>CONTEXT API</h5>

        <NavLinks />

      </nav>

    </NavbarContext.Provider>

  );

};

We can then use **useContext** **in the component that receives the values (child) to have access to them**.

|  |  |
| --- | --- |
| **NavLinks.jsx** | **UserContainer.jsx** |
| import { useState, createContext } from "react";  import NavLinks from "./NavLinks";  export const NavbarContext = createContext();  const Navbar = () => {    const [user, setUser] = useState({ name: "bob" });    const logout = () => {      setUser(null);    };    return (      <NavbarContext.Provider value={{ user, logout }}>        <nav className="navbar">          <h5>CONTEXT API</h5>          <NavLinks />        </nav>      </NavbarContext.Provider>    );  };  export default Navbar; | import { useContext } from "react";  import { NavbarContext } from "./Navbar";  const UserContainer = () => {    const { user, logout } = useContext(NavbarContext);    return (      <div className="user-container">        {user ? (          <>            <p>Hello There, {user?.name?.toUpperCase()}</p>            <button className="btn" onClick={logout}>              logout            </button>          </>        ) : (          <p>Please login</p>        )}      </div>    );  };  export default UserContainer; |

**Context API – Custom Hook**

So pretty much everywhere we want to use the state values user and the function logout we have to import useContext and NavbarContext.

This is a good use for a **custom hook**. So instead of these two inputs we can setup one custom hook and then just import it. We will use the hook using useContext.

|  |  |
| --- | --- |
| **NavLinks.jsx** | **UserContainer.jsx** |
| import { useState, createContext } from "react";  import NavLinks from "./NavLinks";  import { useContext } from "react";  export const NavbarContext = createContext();  //custom hook  export const useAppContext = () => useContext(NavbarContext);  const Navbar = () => {    const [user, setUser] = useState({ name: "bob" });    const logout = () => {      setUser(null);    };    return (      <NavbarContext.Provider value={{ user, logout }}>        <nav className="navbar">          <h5>CONTEXT API</h5>          <NavLinks />        </nav>      </NavbarContext.Provider>    );  };  export default Navbar; | import { useAppContext } from "./Navbar";  const UserContainer = () => {    const { user, logout } = useAppContext();    return (      <div className="user-container">        {user ? (          <>            <p>Hello There, {user?.name?.toUpperCase()}</p>            <button className="btn" onClick={logout}>              logout            </button>          </>        ) : (          <p>Please login</p>        )}      </div>    );  };  export default UserContainer; |

**Global Context API**

A even more neat way to do this, is define our states and custom hooks in a separate file. Then, by importing that file, we have access to every state. To do that:

1. Create a separate file where you create the context and pass the values as GlobalContext.Provider attribute – see Context.jsx
2. Wrap the entire application (the App component) in <AppContext> - see main.jsx.
3. In the component where you want to use those states, import them using curly braces (because they were exported with curly braces), as well as the hook you created useGlobalContext

As you can see, **App.jsx** (or any other component) now has access to the states inside **Context.jsx**, as long as they import useGlobalContext

|  |  |
| --- | --- |
| **main.jsx** | **App.jsx** |
| import React from "react";  import ReactDOM from "react-dom/client";  import App from "./App.jsx";  import "./index.css";  import AppContext from "./context.jsx";  ReactDOM.createRoot(document.getElementById("root")).render(      <AppContext>        <App />      </AppContext>  ); | import "./App.css";  import { useGlobalContext } from "./context";  function App() {    const { name } = useGlobalContext();    console.log(name);    return <></>;  }  export default App; |
| **Context.jsx** | **Console.log** |
| import { createContext, useContext, useState } from "react";  const GlobalContext = createContext();  export const useGlobalContext = () => useContext(GlobalContext);  const AppContext = (props) => {    const [name, setName] = useState("peter");    return (      <GlobalContext.Provider value={{ name, setName }}>        {props.children}      </GlobalContext.Provider>    );  };  export default AppContext; |  |

**useReducer**

You can think of **useReducer** as a light version of Redux, one of the most famous state management libraries out there.

As your application grows in size, it's going to be very hard to manage everything with just **useState**, especially if you have multiple developers working on the project. That's where state management libraries like Redux come into play, since they provide much needed structure and set of rules which in a perfect world leads to less bugs and easier code management.

The problem with such libraries is that they require quite a bit of boilerplate code and some time to get used to the terminology.

So, react released, **useReducer** hook. And I like to think of use reducer as a middle ground between using full blown state library and just using useState.

**Example**

Consider the following code where we use **useState** to create a list of people and buttons to **remove**, **clear** and **reset** the list.

|  |  |
| --- | --- |
| import React from "react";  import { data } from "../../../data";  const ReducerBasics = () => {    const [people, setPeople] = React.useState(data);    const removeItem = (id) => {      let newPeople = people.filter((person) => person.id !== id);      setPeople(newPeople);    };    const clearList = () => {      setPeople([]);    };    const resetList = () => {      setPeople(data);    };    return (      <div>        {people.map((person) => {          const { id, name } = person;          return (            <div key={id} className="item">              <h4>{name}</h4>              <button onClick={() => removeItem(id)}>remove</button>            </div>          );        })}        {people.length < 1 ? (          <button            className="btn"            style={{ marginTop: "2rem" }}            onClick={resetList}          >            reset          </button>        ) : (          <button            className="btn"            style={{ marginTop: "2rem" }}            onClick={clearList}          >            clear items          </button>        )}      </div>    );  };  export default ReducerBasics; | |
|  |  |

When it comes to **useState**, we pass the default state. With **useReducer**, is a bit more complex. We need to pass two things:

* **Default state** – object that incorporates every state.
* **Reducer** – a function that is going to manipulate the state.

In this case, our default state would only take the people state and would look like this, where we have the people state and the default value data, which is the array of people. If you decide you want more states, you can keep adding them to defaultState.

|  |  |
| --- | --- |
| const defaultState = {    people: data,  }; | export const data = [    { id: 1, name: 'john' },    { id: 2, name: 'peter' },    { id: 3, name: 'susan' },    { id: 4, name: 'anna' },  ]; |

With useReducer, we are getting back a state and a dispatch. All your states will be represented in the state below. Then you can access the states by using dot notation (e.g. **state.people** accesses people state).

const[state,dispatch] = useReducer(reducer, defaultState);

We will still update the state like with **useState**, it’s just not happening directly. With **useState**, we just invoke **setPeople** and that automatically updates the state, but that’s not how **useReducer** works.

1. You’ll **dispatch** (i.e. **pass in the action that you want to do**)
2. it’s going to go through the reducer and whatever we get from the reducer, that’s going to be our new state.

So, the basic setup of use reducer looks like this:

import { useReducer } from "react";

import { data } from "../../../data";

const defaultState = {

  people: data,

};

//Reducer, where we are going to control our entire state

const reducer = () => {};

const ReducerBasics = () => {

  const [state, dispatch] = useReducer(reducer, defaultState);

  const removeItem = (id) => {};

  const clearList = () => {};

  const resetList = () => {};

  return (

    <div>

      {state.people.map((person) => {

        const { id, name } = person;

        return (

          <div key={id} className="item">

            <h4>{name}</h4>

            <button onClick={() => removeItem(id)}>remove</button>

          </div>

        );

      })}

      {state.people.length < 1 ? (

        <button

          className="btn"

          style={{ marginTop: "2rem" }}

          onClick={resetList}

        >

          reset

        </button>

      ) : (

        <button

          className="btn"

          style={{ marginTop: "2rem" }}

          onClick={clearList}

        >

          clear items

        </button>

      )}

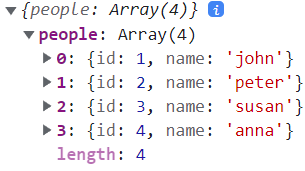
    </div>

  );

};

export default ReducerBasics;

and if we log the state, console.log(state); we get the default array with all the objects.



Although we have the basic setup, we have the default values, **no functionality** is being used. For that, we need to **dispatch something called** **action**, and that **action** is going to be handled in the **reducer**. The **reducer is going to be the new state**. The reducer gets the current state (before the update) and the action (what we are trying to do).

const reducer = (state,action) => {};

Then in dispatch, we provide a **type**, which is going to be out action. By convention, **this action is written in upper case and underscores**. The **action is always a string** (any string you want).

const clearList = () => {

  dispatch({ type: "CLEAR\_LIST" });

};

**Then, in the reducer, we handle that type**. Remember we have to use the spread operator so we don’t delete the other properties. **Whatever we return from the reducer is then going to be out state value.**

const reducer = (state, action) => {

  if (action.type === "CLEAR\_LIST") {

    return { ...action, people: [] };

  }

};

One common convention to avoid errors is to assign the string to a variable:

const CLEAR\_LIST = "CLEAR\_LIST";

const reducer = (state, action) => {

  if (action.type === CLEAR\_LIST) {

    return { ...action, people: [] };

  }

};

  const clearList = () => {

    dispatch({ type: CLEAR\_LIST });

  };

Also, in case we don’t handle a certain action in the reducer, we want to throw the following error:

const reducer = (state, action) => {

  if (action.type === CLEAR\_LIST) {

    return { ...action, people: [] };

  }

  throw new Error (`No matching ${action.type} - actipon type`)

};

For the remove people, we have to pass the id as payload (the payload name is just a convention for the property name).

import { useReducer } from "react";

import { data } from "../../../data";

const defaultState = {

  people: data,

};

const CLEAR\_LIST = "CLEAR\_LIST";

const RESET\_LIST = "RESET\_LIST";

const REMOVE\_ITEM = "REMOVE\_ITEM";

//Reducer, where we are going to control our entire state

const reducer = (state, action) => {

  if (action.type === CLEAR\_LIST) {

    return { ...action, people: [] };

  }

  if (action.type === RESET\_LIST) {

    return { ...action, people: data };

  }

  if (action.type === REMOVE\_ITEM) {

    const newPeople = state.people.filter(

      (person) => person.id !== action.payload.id

    );

    return { ...action, people: newPeople };

  }

  throw new Error(`No matching ${action.type} - actipon type`);

};

const ReducerBasics = () => {

  const [state, dispatch] = useReducer(reducer, defaultState);

  const removeItem = (id) => {

    dispatch({ type: REMOVE\_ITEM, payload: { id } });

  };

  const clearList = () => {

    dispatch({ type: CLEAR\_LIST });

  };

  const resetList = () => {

    dispatch({ type: RESET\_LIST });

  };

  console.log(state);

  return (

    <div>

      {state.people.map((person) => {

        const { id, name } = person;

        return (

          <div key={id} className="item">

            <h4>{name}</h4>

            <button onClick={() => removeItem(id)}>remove</button>

          </div>

        );

      })}

      {state.people.length < 1 ? (

        <button

          className="btn"

          style={{ marginTop: "2rem" }}

          onClick={resetList}

        >

          reset

        </button>

      ) : (

        <button

          className="btn"

          style={{ marginTop: "2rem" }}

          onClick={clearList}

        >

          clear items

        </button>

      )}

    </div>

  );

};

export default ReducerBasics;

**useReduucer – Organizing the component tree**

As the applications grow, it is helpful to split the actions and reducer in a separate file, otherwise the projects start getting harder to read. So, the structure for this program would be as following:

|  |  |
| --- | --- |
| **useReducer.jsx** | |
| import { useReducer } from "react";  import { data } from "../../../data";  import { CLEAR\_LIST, RESET\_LIST, REMOVE\_ITEM } from "./actions";  import reducer from "./reducer";  const defaultState = {    people: data,  };  const ReducerBasics = () => {    const [state, dispatch] = useReducer(reducer, defaultState);    const removeItem = (id) => {      dispatch({ type: REMOVE\_ITEM, payload: { id } });    };    const clearList = () => {      dispatch({ type: CLEAR\_LIST });    };    const resetList = () => {      dispatch({ type: RESET\_LIST });    };    console.log(state);    return (      <div>        {state.people.map((person) => {          const { id, name } = person;          return (            <div key={id} className="item">              <h4>{name}</h4>              <button onClick={() => removeItem(id)}>remove</button>            </div>          );        })}        {state.people.length < 1 ? (          <button            className="btn"            style={{ marginTop: "2rem" }}            onClick={resetList}          >            reset          </button>        ) : (          <button            className="btn"            style={{ marginTop: "2rem" }}            onClick={clearList}          >            clear items          </button>        )}      </div>    );  };  export default ReducerBasics; | |
| **actions.js** | **reducer.js** |
| export const CLEAR\_LIST = "CLEAR\_LIST";  export const RESET\_LIST = "RESET\_LIST";  export const REMOVE\_ITEM = "REMOVE\_ITEM"; | import { CLEAR\_LIST, RESET\_LIST, REMOVE\_ITEM } from "./actions";  import { data } from "../../../data";  const reducer = (state, action) => {    if (action.type === CLEAR\_LIST) {      return { ...action, people: [] };    }    if (action.type === RESET\_LIST) {      return { ...action, people: data };    }    if (action.type === REMOVE\_ITEM) {      const newPeople = state.people.filter(        (person) => person.id !== action.payload.id      );      return { ...action, people: newPeople };    }    throw new Error(`No matching ${action.type} - actipon type`);  };  export default reducer; |

**Data Structures**

**Array of objects**

Using an array to store shopping cart data may not be the best option because:

* it can be **less efficient for lookups and updates**, especially for larger datasets.
* Arrays are also **less flexible than maps** when it comes to associating values with unique identifiers (IDs).

const cart = [

  { id: 1, name: 'first', price: 10 },

  { id: 2, name: 'second', price: 20 },

];

**Array of objects**

The downsides of using an object to store shopping cart data include:

* The risk of **unintended property overwriting** or **unexpected behaviour** when iterating over inherited properties.
* Objects can **only use string keys**, which can be limiting if you need to use non-string keys.
* **Deleting properties from an object can also be tricky**, especially when dealing with inherited properties.

const cart = {

  'id-1': { id: 1, name: 'first', price: 10 },

  'id-2': { id: 2, name: 'second', price: 20 },

};

**Mapping data structures**

For a shopping cart application, using a **new Map()** to store the cart data is beneficial because it allows for efficient lookups and updates based on unique product IDs. Using a **Map** can also ensure that **each item in the cart has a unique identifier** and **can easily be updated or removed without affecting other items in the cart**.

A **Map** is a built-in data structure in JavaScript that allows you to **store key-value pairs**, where both the **keys and values can be any data type** (not just string keys like object structure). Here's a simple example:

// create a new Map instance

const cart = new Map();

// set some key-value pairs

cart.set('apple', { name: 'Apple', price: 0.5, quantity: 3 });

cart.set('banana', { name: 'Banana', price: 0.3, quantity: 6 });

cart.set('orange', { name: 'Orange', price: 0.4, quantity: 4 });

// get the value associated with a key

const appleDetails = cart.get('apple'); // returns { name: 'Apple', price: 0.5, quantity: 3 }

// check if a key exists in the map

const hasPear = cart.has('pear'); // returns false

// get the number of key-value pairs in the map

const size = cart.size; // returns 3

// delete a key-value pair from the map

cart.delete('banana');

// loop over the key-value pairs in the map

for (let [key, { name, price, quantity }] of cart) {

  console.log(key, name, price, quantity); // prints 'apple' 'Apple' 0.5 3, 'banana' 'Banana' 0.3 6, 'orange' 'Orange' 0.4 4

}

**Converting from Array to Map Data Structure**

Consider we have the following array of objects:

const items = [

  { id: 1, name: 'first', price: 10 },

  { id: 2, name: 'second', price: 20 },

];

1. First, we want to **map over each object** and for every item I want to return a new array, with the **first element as the ID and then the entire object**.

const cartItems = items.map((item) => [item.id, item]);

console.log(cartItems);

// prints:

// [

//   [1, { id: 1, name: 'first', price: 10 }],

//   [2, { id: 2, name: 'second', price: 20 }],

// ];

1. We store that new array as a new Map instance.

// create a new Map instance

const cart = new Map(cartItems);

1. We turn the **Map** instance into an array of key-value pairs, because we need to use that whenever we want to render the items.

// convert the Map to an array of key-value pairs

const cartArray = Array.from(cart.entries());

console.log(cartArray);

// prints:

// [

//   [1, { id: 1, name: 'first', price: 10 }],

//   [2, { id: 2, name: 'second', price: 20 }]

// ]

**Performance**

Let's see how we can test and improve the performance of our React applications. **React is fast by default**. So, all the techniques we are going to discuss **make way more sense when you are working on a bigger application**.

**Example**

See the following example where we render a list of people and we also have a count.

|  |  |
| --- | --- |
| **Index.jsx** | |
| import { useState } from 'react';  import { data } from '../../../../data';  import List from './List';  const LowerState = () => {    const [people, setPeople] = useState(data);    const [count, setCount] = useState(0);    return (      <section>        <button          className='btn'          onClick={() => setCount(count + 1)}          style={{ marginBottom: '1rem' }}        >          count {count}        </button>        <List people={people} />      </section>    );  };  export default LowerState; | |
| **List.jsx** | |
| import Person from './Person';  const List = ({ people }) => {    return (      <div>        {people.map((person) => {          return <Person key={person.id} {...person} />;        })}      </div>    );  };  export default List; | |
| **Person.jsx** | **Result** |
| import { useEffect } from 'react';  const Person = ({ name }) => {    console.log('render');    return (      <div>        <h4>{name}</h4>      </div>    );  };  export default Person; |  |

Each time we click on the button, we want the number on the button to change. This change occurs through re-rendering. But it **doesn’t re-render only the number, it also re-renders the names**.

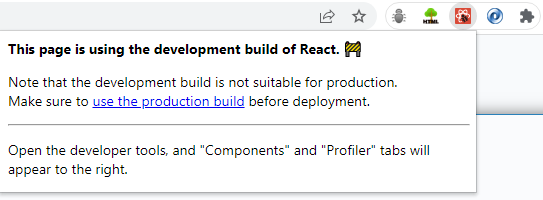
Not a problem in this application, but if you have 60 components and you keep re-rendering them for nothing, that is a problem.

Remember, there are 2 situations when re-rendering occurs:

* When the component's state or props change, React will re-render the component to reflect these changes.
* When the parent element re-renders, even if the component's state or props have not changed.

**React Dev Tools**

One way to know the performance of our website is through **React Dev Tools**. Bare in mind that while you are developing the website react does extra checks and will run slower than when it’s built and deployed.

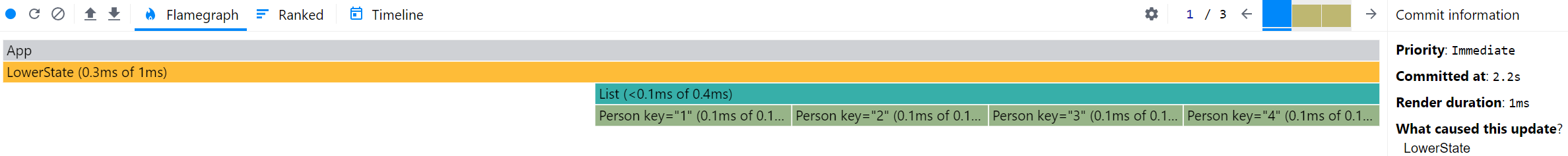


You can use the following Dev Tool settings.

|  |  |
| --- | --- |
|  |  |
|  |  |

If you go to the “*profiler*” tab and click “*start profiling*”, it will allow you to see:

* Which components re-rendered
* Why they re-rendered
* How much time they took to re-render



As you can see, every component re-rendered just because we clicked the button to change the count value.

Note: To use the profiler, **Dev Tools needs to be with the developer mode ON**, which can be done in the settings.

**Possible Solutions - Lower the state**

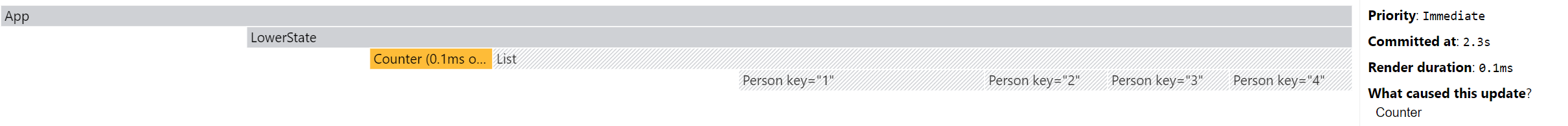
We can lower the state. In other words, we could just **move the state variable as well as the button to a separate component**.

There are going to be instances where you can simply just split up the logic. Instead of jamming everything in one component and having those unnecessary re-renders. If we move the logic to a separate component, it won’t re-rerender the people list anymore.

If you look at **Index.js**, we don’t have the count state value anymore. The count state value is in a separate component (**Counter.jsx**). As a result, **we will be updating that state value, but that is not going to trigger re-renders in the lower state component**.

|  |
| --- |
| **Index.jsx** |
| import { useState } from "react";  import { data } from "../../../../data";  import List from "./List";  import Counter from "./Counter";  const LowerState = () => {    const [people, setPeople] = useState(data);    return (      <section>        <Counter />        <List people={people} />      </section>    );  };  export default LowerState; |
| **List.jsx** |
| import Person from './Person';  const List = ({ people }) => {    return (      <div>        {people.map((person) => {          return <Person key={person.id} {...person} />;        })}      </div>    );  };  export default List; |
| **Person.jsx** |
| import { useEffect } from 'react';  const Person = ({ name }) => {    console.log('render');    return (      <div>        <h4>{name}</h4>      </div>    );  };  export default Person; |
| **Counter.jsx** |
| import { useState } from "react";  const Counter = () => {    const [count, setCount] = useState(0);    return (      <button        className="btn"        onClick={() => setCount(count + 1)}        style={{ marginBottom: "1rem" }}      >        count {count}      </button>    );  };  export default Counter; |

Notice now how only the **Counter** component re-renders when we click the button.

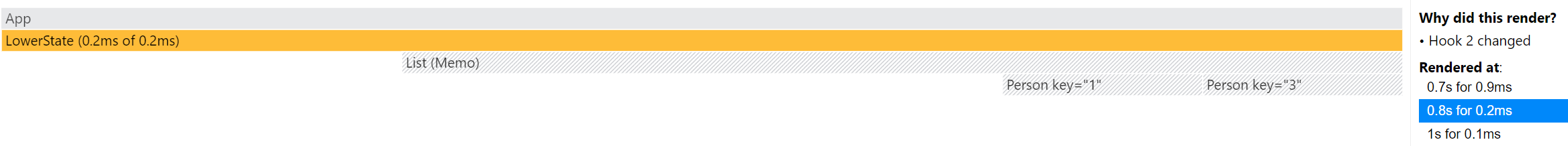


**React.memo()**

Sometimes is not possible to push the state down. An alternative to that would be use **React.memo()** function. Using memo will cause **React to skip rendering a component if its props have not changed**. All you have to do is wrap **List** component (which is the component that triggers the re-render in people) when exporting it and importing *memo* from react.

|  |
| --- |
| **Index.jsx** |
| import { useState } from "react";  import { data } from "../../../../data";  import List from "./List";  const LowerState = () => {    const [people, setPeople] = useState(data);    const [count, setCount] = useState(0);    return (      <section>        <button          className="btn"          onClick={() => setCount(count + 1)}          style={{ marginBottom: "1rem" }}        >          count {count}        </button>        <List people={people} />      </section>    );  };  export default LowerState; |
| **List.jsx** |
| import Item from "./Person";  import { memo } from "react";  const List = ({ people }) => {    return (      <div>        {people.map((person) => {          return <Item key={person.id} {...person} />;        })}      </div>    );  };  export default memo(List); |
| **Person.jsx** |
| const Person = ({ name }) => {    return (      <div>        <h4>{name}</h4>      </div>    );  };  export default Person; |

As you can see, only **LowerState** re-rendered.



**useCallback Hook**

The **useCallback** is a hook in React that allows you to **memoize a function**. It takes two arguments:

* the function you want to memorize
* an array of dependencies.

The hook will **return a memoized version of the function that only changes if one of the values in the dependency array changes**.

By memoizing the function, you can avoid unnecessary re-renders and improve the performance of your React application. **The function will only be re-created if one of its dependencies change**, otherwise the same instance of the function will be returned.

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

function MyComponent() {

  const [data, setData] = useState([]);

  const handleClick = useCallback(() => {

    console.log(data);

  }, [data]);

  return (

    <div>

      <button onClick={handleClick}>Click me</button>

    </div>

  );

}

In this example, the handleClick function is memoized using useCallback and the data prop is passed as a dependency. This means that the handleClick function will only be re-created if the data prop changes.

Here is an example of how you might use useCallback, where the component will only re-render if the fetchData prop changes.

import Final from './tutorial/02-useEffect/final/04-fetch-data';

import { useState, useEffect, useCallback } from 'react';

const url = 'https://api.github.com/users';

const FetchData = () => {

  const [users, setUsers] = useState([]);

  const fetchData = useCallback(async () => {

    try {

      const response = await fetch(url);

      const users = await response.json();

      setUsers(users);

    } catch (error) {

      console.log(error);

    }

  }, []);

  useEffect(() => {

    fetchData();

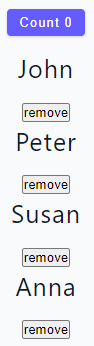
  }, [fetchData]);

  // rest of the logic

};

Going back to our previous example, but adding a “remove name” functionality:

|  |
| --- |
| **Index.jsx** |
| import { useState } from "react";  import { data } from "../../../../data";  import List from "./List";  const LowerState = () => {    const [people, setPeople] = useState(data);    const [count, setCount] = useState(0);    const removePerson = (id) => {      const newPeople = people.filter((person) => person.id !== id);      setPeople(newPeople);    };    return (      <section>        <button          className="btn"          onClick={() => setCount(count + 1)}          style={{ marginBottom: "1rem" }}        >          count {count}        </button>        <List people={people} removePerson={removePerson} />      </section>    );  };  export default LowerState; |
| **List.jsx** |
| import Item from "./Person";  import { memo } from "react";  const List = ({ people, removePerson }) => {    return (      <div>        {people.map((person) => {          return <Item key={person.id} {...person} removePerson={removePerson} />;        })}      </div>    );  };  export default memo(List); |
| **Person.jsx** |
| const Person = ({ name, removePerson, id }) => {    return (      <div>        <h4>{name}</h4>        <button onClick={() => removePerson(id)}>remove</button>      </div>    );  };  export default Person; |



In this case**, clicking the button would re-render the List component**.

Although **memo** doesn’t re-render a component if it didn’t change, in this case, each time a re-render occurs, removePerson is created from the scratch. Note that removePerson is now a prop of the List component. So, although the people prop in **List** is not changing, removePerson is.

One way to solve this, is by using useCallback() hook which **returns a memoized callback function**. It has a dependency array, so **the function will only run when the dependency updates**. In this case, when the people array updates.

Think of memoization as caching a value so that it does not need to be recalculated.

  const removePerson = useCallback(

    (id) => {

      const newPeople = people.filter((person) => person.id !== id);

      setPeople(newPeople);

    },

    [people]

  );

So, when we use the removePerson function as an argument of useCallback hook:

1. removePerson is called and it sets a new State for people array
2. Page re-renders and we see 3 items now
3. As the removePerson is not being created again, the people array value it has is same (the 4 item array)
4. Now when we click *remove* again, it sets the value to 3 items filter array again.

Any value of the properties it had inside won’t change. It will remain the same like when it was called the first time (it will work with the local copy it had made when it was called the first time).

**useMemo Hook**

The **useMemo** hook is a hook in React that allows you to memoize a value. It takes two arguments:

* a function that returns the value you want to memorize
* an array of dependencies.

**The hook will return the memoized value that will only change if one of the values in the dependency array changes.**

Here is an example of how you might use useMemo:

import React, { useMemo } from 'react';

function MyComponent({ data }) {

  const processedData = useMemo(() => {

    return data.map((item) => item.toUpperCase());

  }, [data]);

  return (

    <div>

      {processedData.map((item) => (

        <div key={item}>{item}</div>

      ))}

    </div>

  );

}

In this example, the processedData value is memoized using useMemo and the data prop is passed as a dependency. This means that the processedData value will only be recalculated if the data prop changes.

For example, if you have a slow function like the one below in the **index.jsx** (see previous example), every time you click on the button, everything is going to re-rendered, which means slowFunction runs each time the re-render occurs.

const slowFunction = () => {

  let value = 0;

  for (let i = 0; i <= 1000000000; i++) {

    value += i;

  }

  return value;

};

export default slowFunction;

If we put the slow function inside a useMemo(), with a dependency array of **[]**, then the function is only going to be used in the first render.

**useTransition Hook**

Imagine we have some kind of user interaction in our app which triggers heave compotation (for example, user type, something in the input and based on the input value you filter 20,000 items).

While the computation is taking place, it will actually block the UI (our app will stop responding)

So, **useTransition** just allows us to **mark certain functionality as less urgent, which in turn prevents the whole UI blocking thing**.

Even though this is a very cool addition to the library, it has very specific use cases and most likely it's not something you'll use immediately in all your applications.

In the example below, we create an array of 5000 items, which will all be svg files, and set it to the items state.

import { useState } from "react";

const LatestReact = () => {

  const [text, setText] = useState("");

  const [items, setItems] = useState([]);

  const handleChange = (e) => {

    setText(e.target.value);

    // slow down CPU

    const newItems = Array.from({ length: 5000 }, (\_, index) => {

      return (

        <div key={index}>

          <img src="/vite.svg" alt="" />

        </div>

      );

    });

    setItems(newItems);

  };

  return (

    <section>

      <form className="form">

        <input

          type="text"

          className="form-input"

          value={text}

          onChange={handleChange}

        />

      </form>

      <h4>Items Below</h4>

      <div

        style={{

          display: "grid",

          gridTemplateColumns: "1fr 1fr 1fr",

          marginTop: "2rem",

        }}

      >

        {items}

      </div>

    </section>

  );

};

export default LatestReact;

So, each time we type something, the whole page just freezes up until it is capable of showing the images.

By putting the slow code inside the useTransition(), making it less important, **rendering the** **input** **form won’t wait for the images to be processed**. Another good practice is set up a **loading** state that shows up every time the images are loading. Because each time you type something in the form, the images will load.

import { useState, useTransition } from "react";

const LatestReact = () => {

  const [text, setText] = useState("");

  const [items, setItems] = useState([]);

  const [isPending, startTransition] = useTransition();

  const handleChange = (e) => {

    setText(e.target.value);

    // slow down CPU

    startTransition(() => {

      const newItems = Array.from({ length: 5000 }, (\_, index) => {

        return (

          <div key={index}>

            <img src="/vite.svg" alt="" />

          </div>

        );

      });

      setItems(newItems);

    });

  };

  return (

    <section>

      <form className="form">

        <input

          type="text"

          className="form-input"

          value={text}

          onChange={handleChange}

        />

      </form>

      <h4>Items Below</h4>

      {isPending ? (

        <h4>Loading...</h4>

      ) : (

        <div

          style={{

            display: "grid",

            gridTemplateColumns: "1fr 1fr 1fr",

            marginTop: "2rem",

          }}

        >

          {items}

        </div>

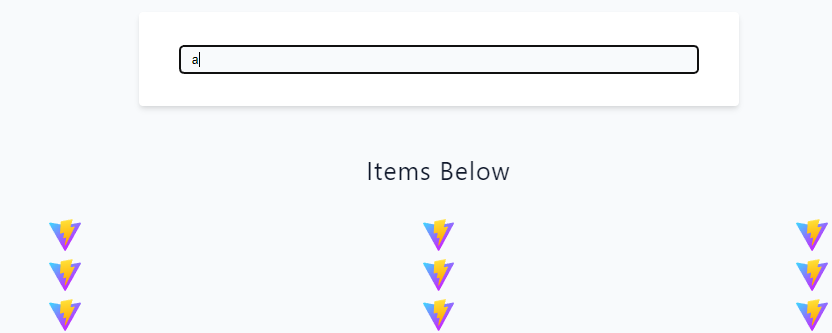
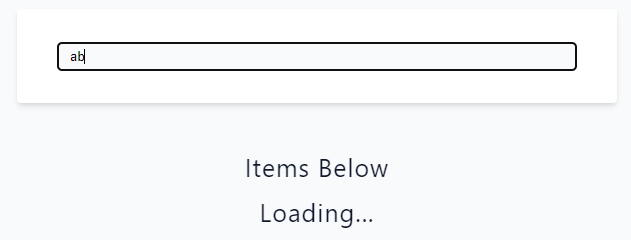
      )}

    </section>

  );

};

export default LatestReact;

**React Suspense API**

Let's see how we can do code splitting in React, which will allow us to **progressively load** or lazy load **our application**. Its benefits include:

* **Improve performance** by splitting up a code into smaller and more manageable chunks.
* **Only essential code needed for the initial render of your application is loaded**. The remaining code is loaded as the user interacts with you application, leading to smoother and less blocking user experience.
* **Reduce the size of the initial JavaScript** payload that needs to be loaded (resulting in faster load times).

When it comes to bigger projects, not all pages and components are equal, meaning some resources will be used more often by users. For example, a *homepage* gets more traffic than a *contact* page.

So, if some resources are used less often than the others, does it make sense to jam all of that code when we initially ship our application to the browser? Of course not. The more code we send, the more time it will take to compile.

So, we should **prioritize the important resources** over the less important ones. In order to lazy load our components or progressively load our application, we'll use a tool called **suspense**.

In our example, we'll progressively load a big component when the user clicks the button. However, a more realistic scenario is to lazy load our components when we have multiple pages, which is something we'll cover when we discuss routing in React.

The Suspense API is a feature in React that allows you to **manage the loading state of your components**. It provides a way to "suspend" rendering of a component until some data has been fetched, and display a fallback UI in the meantime. This makes it easier to handle asynchronous data loading and provide a smooth user experience in your React application.

Notice in the example below that we import **SlowComponent.jsx** (which renders an array of 5000 images) using lazy, and then we wrap it inside <Suspense></Suspense>. This way, **SlowComponent.jsx** will only be imported when show is true, i.e. **when we click the *toggle* button**, improving the performance when the page opens.

|  |
| --- |
| **index.jsx** |
| import { useState, useTransition, Suspense, lazy } from "react";  const SlowComponent = lazy(() => import("./SlowComponent"));  const LatestReact = () => {    const [text, setText] = useState("");    const [items, setItems] = useState([]);    const [isPending, startTransition] = useTransition();    const [show, setShow] = useState(false);    const handleChange = (e) => {      setText(e.target.value);      // slow down CPU      startTransition(() => {        const newItems = Array.from({ length: 5000 }, (\_, index) => {          return (            <div key={index}>              <img src="/vite.svg" alt="" />            </div>          );        });        setItems(newItems);      });    };    return (      <section>        <form className="form">          <input            type="text"            className="form-input"            value={text}            onChange={handleChange}          />        </form>        <h4>Items Below</h4>        {isPending ? (          <h4>Loading...</h4>        ) : (          <div            style={{              display: "grid",              gridTemplateColumns: "1fr 1fr 1fr",              marginTop: "2rem",            }}          >            {items}          </div>        )}        <button onClick={() => setShow(!show)} className="btn">          toggle        </button>        {show && (          <Suspense fallback={<h4>Loading...</h4>}>            <SlowComponent />          </Suspense>        )}      </section>    );  };  export default LatestReact; |
| **SlowComponent.jsx** |
| import { useState } from 'react';  const newItems = Array.from({ length: 5000 }, (\_, index) => {    return (      <div key={index}>        <img src='/vite.svg' alt='' />      </div>    );  });  const SlowComponent = () => {    const [items, setItems] = useState(newItems);    return (      <div        style={{          display: 'grid',          gridTemplateColumns: '1fr 1fr 1fr',          marginTop: '2rem',        }}      >        {items}      </div>    );  };  export default SlowComponent; |

