**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 out 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).

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

ReactDOM.render(

  <div>

    <h1>Hello</h1>

    <h1>World</h1>

  </div>

  , document.getElementById("root"));

But we couldn’t render:

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** | **JSX** |
| <div tabindex="1">      <button onclick="myFunction()">click me</button>      <label for='name'>Name</label>      <input readonly id='name' />  </div> | 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>

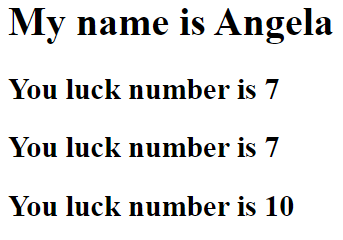
        <h2>You luck number is {num}</h2>

        <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**

In order to keep the styling in the CSS, we should use JSX to give classes to the elements, and not to style them. 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).

The code below is simply a list of 3 items (images) with a heading, and styles applied to the different elements

**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:**

.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 to 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, basically, the **Index.js** file will always look very simple, like it is below.

|  |  |
| --- | --- |
| **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**

Have a look at the example below.

|  |  |
| --- | --- |
| **Index.js** | **Math.jsx** |
| 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}; |
| **Result** | |
|  | |

Here 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 **{}**.

**Importing Local Images**

When importing local images, you can 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,

  },

];

**React Props**

**Props** stands for **properties**. **Props** are **arguments** passed into React components when they are created or setup. Then they are 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>{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 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; |

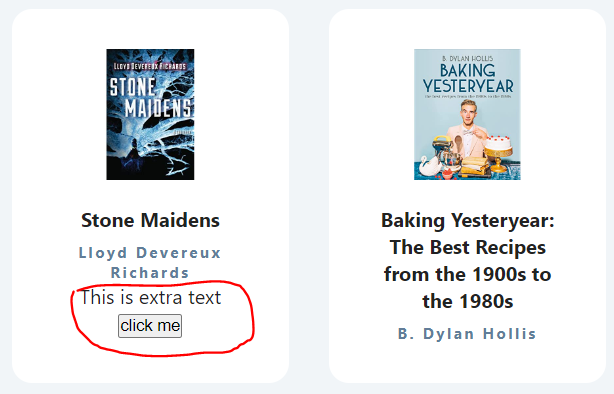
We could also destructure directly in the **Card** component arguments:

|  |
| --- |
| **Card.jsx** |
| import React from "react";  function Card({name, tel, email}) {  return (  <div>  <h2>{props.name}</h2>  <p>{props.tel}</p>  <p>{props.email}</p>  </div>  );  }  export default Card; |

**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.

|  |  |
| --- | --- |
| **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; |



**Mapping Data to Components**

**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]

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.

|  |  |
| --- | --- |
| **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 (      <section className="booklist">        {books.map((book) => {          return <Book {...book} key={book.id} />;        })}      </section>    );  }  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; |

**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";  var 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:

|  |  |  |
| --- | --- | --- |
| **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 isLoggedIn = false; |

In this case, we want to show the field “confirm password” if the user is not registered. Thus, we have to had the component inside a condition statement (ternary operator).

Also, there are different ways having a condition where it either shows or doesn’t show something.

|  |
| --- |
| 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. Lastly, we set the initial state to an empty string: **useState("")**

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.

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>

    </>

  )

}

Both would result in:



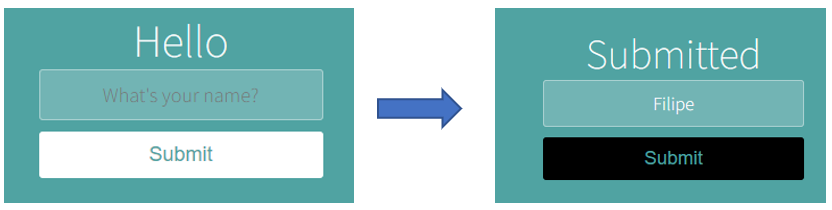
Consider the following code, where we have a counter that goes up every time the button **+** is pressed.

|  |  |
| --- | --- |
| 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>  ) | The variable **count** increases in value. However, it is not re-rendered to the screen. |
| 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>  ) | 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**. |
| 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; | The correct way is to use **React.useState()**.  It takes the initial value as the argument (in this case **0**).  It has **2** **variables** associated:   * 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. |

**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 the state of two components:

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

So we need to create 2 hooks:

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

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

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, as you will see later).

  function handleClick() {

    setHeadingText("Submitted");

  }

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);

  }

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

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

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. The **remove** button removes the individual name, while the **clear items** removes the whole array.

|  |  |
| --- | --- |
| **main.js** | **App.jsx** |
| import React from 'react';  import ReactDOM from 'react-dom/client';  import './index.css';  import App from './App';  const root = ReactDOM.createRoot(document.getElementById('root'));  root.render(<App />); | import Starter from "./tutorial/01-useState/starter/03-useState-array.jsx";  function App() {    return (      <div className="container">        <Starter />      </div>    );  }  export default App; |
| **data.js** | **Result** |
| export const data = [    { id: 1, name: 'john' },    { id: 2, name: 'peter' },    { id: 3, name: 'susan' },    { id: 4, name: 'anna' },  ]; |  |
| **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; | |

**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**.

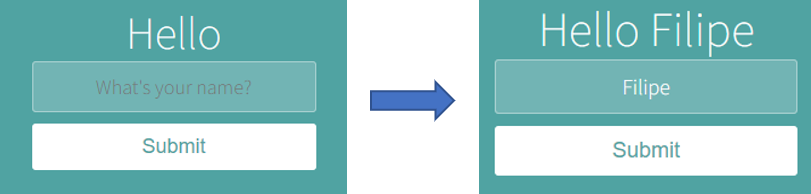
|  |  |
| --- | --- |
| const displayPerson = () => {      setPerson({name: "john"})    }; |  |

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"})    }; |  |

**React Forms**

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



To do this, we need to:

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 need 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);

  }

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 final code 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>

  );

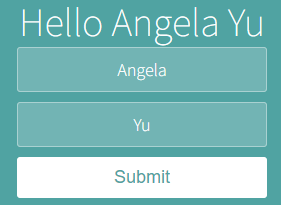
}

export default App;

**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**

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'}

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

What its syntax means is:

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 this:

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 a state being managed inside our **ToDoItem**. But, this state is localized to this item (which works for the line-though goal, which only has to do with the items in the list).

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 components, we should always have a **key**, which identifies the object (in this case the item) 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;

So 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; |

**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>

**Creating a React app with Vite**

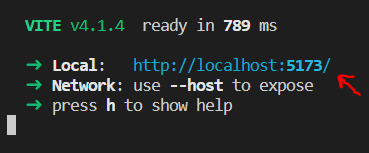
You need Node.js and Hyper installed in order to create a react app with Vite.

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.

**React useEffect**

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.

|  |  |
| --- | --- |
| **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 your want. So pretty much you can set up a user effect 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.

**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, 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 is used to call 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; |

**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 following code, which accesses the firstName property of an object:

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 make 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 doens’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,y as:

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

**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"}

**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 image, 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;

**Controlled Inputs**

And now let's start working on controlled inputs. And essentially, when you hear controlled inputs, just think that there's going to be a state value.

Now it can be one value which represents all of the inputs, or it can be a case where each input is going to have a state value that is associated with that input and as you're changing the value in the input. You're automatically also changing the value in the state.

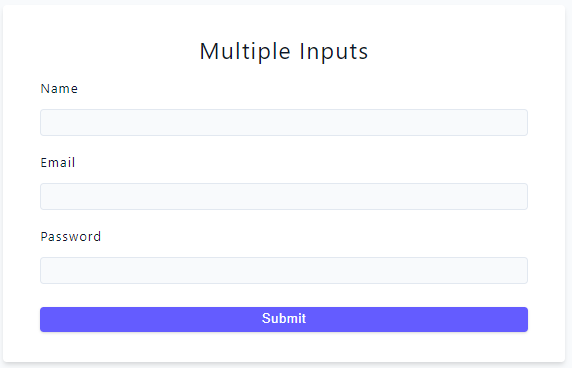
Basically, whatever we're going to be typing, this is going to be added to the state value.

And then whenever you're ready to submit the form, you can 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 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

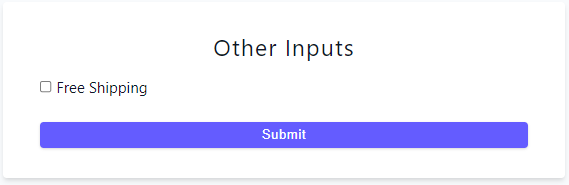
        </button>

      </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>

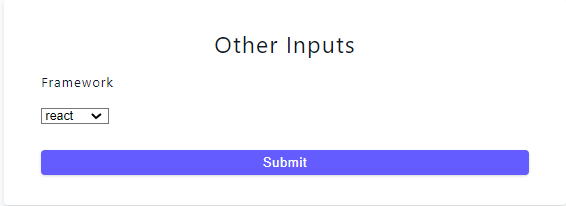
      </form>

    </div>

  );

};

export default OtherInputs;



**Form Data API**

Form Data API is a interface that allows us to contract a set of key-value pairs representing the form fields and their values.

**Form Data API**

There is an alternative to control the inputs. We can also submit the form with uncontrolled inputs, basically without referencing the state value using data API. This is arguably a better approach if you have more than 1 input.

It comes from Vanilla JS, and is a interface that allows us to contract a set of key-value pairs representing the form fields and their values.

**useRef**

**useRef** is a lot like your state, so it preserves the value between the renders. But the difference is that updating use ref 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

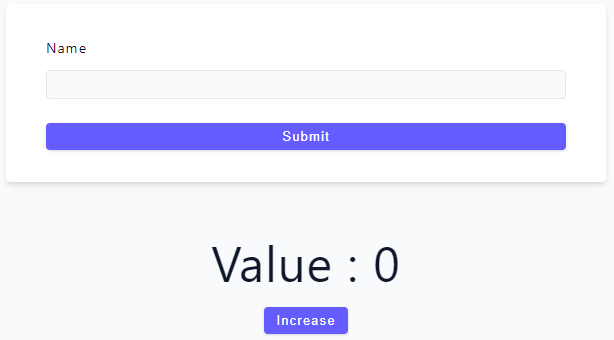
      </button>

    </div>

  );

};

export default UseRefBasics;



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

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

Once we have this in plate, 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 set it ourselves, using some kind of functionality
2. We can use this refContainer and set it equal to any of the DOM nodes.

We are going to try the second 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 rende. 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

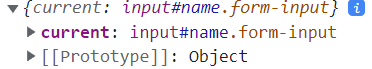
  });

  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);    }; |  |

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

1. So 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 have a add a useEffect and add the state value (the number after “Value” in the image above) 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]); |  |