REACT

You will learn

How to create and nest components

How to add markup and styles

How to display data

How to render conditions and lists

How to respond to events and update the screen

How to share data between components

Creating and nesting components

React apps are made out of components. A component is a piece of the UI (user interface) that has its own logic and appearance. A component can be as small as a button, or as large as an entire page.

React components are JavaScript functions that return markup:

```
function MyButton() {
  return (
    <button>I'm a button</button>
  );
}
```

Now that you've declared MyButton, you can nest it into another component:

```
export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton />
    </div>
  );
```

}

Notice that <MyButton /> starts with a capital letter. That's how you know it's a React component. React component names must always start with a capital letter, while HTML tags must be lowercase.

Have a look at the result:

App.js

Download

Reset

Fork

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

```
function MyButton() {

  return (

    <button>

      I'm a button

    </button>

  );

}


export default function MyApp() {

  return (

    <div>

      <h1>Welcome to my app</h1>

      <MyButton />

    </div>

  );

}
```

Show more

The export default keywords specify the main component in the file. If you're not familiar with some piece of JavaScript syntax, MDN and javascript.info have great references.

Writing markup with JSX

The markup syntax you've seen above is called JSX. It is optional, but most React projects use JSX for its convenience. All of the tools we recommend for local development support JSX out of the box.

JSX is stricter than HTML. You have to close tags like <br />. Your component also can't return multiple JSX tags. You have to wrap them into a shared parent, like a <div>...</div> or an empty <>...</> wrapper:

```
function AboutPage() {
  return (
    <>
      <h1>About</h1>
      <p>Hello there.<br />How do you do?</p>
    </>
  );
}
```

If you have a lot of HTML to port to JSX, you can use an online converter.

Adding styles

In React, you specify a CSS class with className. It works the same way as the HTML class attribute:

```
<img className="avatar" />
```

Then you write the CSS rules for it in a separate CSS file:

```
/* In your CSS */
.avatar {
  border-radius: 50%;
}
```

React does not prescribe how you add CSS files. In the simplest case, you'll add a <link> tag to your HTML. If you use a build tool or a framework, consult its documentation to learn how to add a CSS file to your project.

Displaying data

JSX lets you put markup into JavaScript. Curly braces let you "escape back" into JavaScript so that you can embed some variable from your code and display it to the user. For example, this will display user.name:

```
return (
```

```
  <h1>

  {user.name}

  </h1>

);
```

You can also "escape into JavaScript" from JSX attributes, but you have to use curly braces instead of quotes. For example, className="avatar" passes the "avatar" string as the CSS class, but src={user.imageUrl} reads the JavaScript user.imageUrl variable value, and then passes that value as the src attribute:

```
return (

  <img

    className="avatar"

    src={user.imageUrl}

  />

);
```

You can put more complex expressions inside the JSX curly braces too, for example, string concatenation:

App.js

Download

Reset

Fork

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

```
const user = {
  name: 'Hedy Lamarr',
  imageUrl: 'https://i.imgur.com/yXOvdOSs.jpg',
  imageSize: 90,
};

export default function Profile() {
  return (
    <>
      <h1>{user.name}</h1>
      <img
        className="avatar"
```

```
      src={user.imageUrl}

      alt={'Photo of ' + user.name}

      style={{

       width: user.imageSize,

       height: user.imageSize

      }}

    />

   </>

  );

}
```

Show more

In the above example, style={{}} is not a special syntax, but a regular {} object inside the style={ } JSX curly braces. You can use the style attribute when your styles depend on JavaScript variables.

Conditional rendering

In React, there is no special syntax for writing conditions. Instead, you'll use the same techniques as you use when writing regular JavaScript code. For example, you can use an if statement to conditionally include JSX:

```
let content;

if (isLoggedIn) {

  content = <AdminPanel />;

} else {

  content = <LoginForm />;

}

return (

  <div>
```

```
  {content}
  </div>
);
```

If you prefer more compact code, you can use the conditional ? operator. Unlike if, it works inside JSX:

```
<div>
  {isLoggedIn ? (
    <AdminPanel />
  ) : (
    <LoginForm />
  )}
</div>
```

When you don't need the else branch, you can also use a shorter logical && syntax:

```
<div>
  {isLoggedIn && <AdminPanel />}
</div>
```

All of these approaches also work for conditionally specifying attributes. If you're unfamiliar with some of this JavaScript syntax, you can start by always using if...else.

Rendering lists

You will rely on JavaScript features like for loop and the array map() function to render lists of components.

For example, let's say you have an array of products:

```
const products = [
  { title: 'Cabbage', id: 1 },
  { title: 'Garlic', id: 2 },
```

{ title: 'Apple', id: 3 },

];

Inside your component, use the map() function to transform an array of products into an array of <li> items:

```
const listItems = products.map(product =>
  <li key={product.id}>
    {product.title}
  </li>
);

return (
  <ul>{listItems}</ul>
);
```

Notice how <li> has a key attribute. For each item in a list, you should pass a string or a number that uniquely identifies that item among its siblings. Usually, a key should be coming from your data, such as a database ID. React uses your keys to know what happened if you later insert, delete, or reorder the items.

App.js

Download

Reset

Fork

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

```
const products = [
  { title: 'Cabbage', isFruit: false, id: 1 },
  { title: 'Garlic', isFruit: false, id: 2 },
  { title: 'Apple', isFruit: true, id: 3 },
];

export default function ShoppingList() {
  const listItems = products.map(product =>
    <li
      key={product.id}
      style={{
```

```
      color: product.isFruit ? 'magenta' : 'darkgreen'

    }}

  >

    {product.title}

  </li>

);


  return (

   <ul>{listItems}</ul>

  );
}
```

Show more

Responding to events

You can respond to events by declaring event handler functions inside your components:

```
function MyButton() {
  function handleClick() {
    alert('You clicked me!');
  }

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

Notice how onClick={handleClick} has no parentheses at the end! Do not call the event handler function: you only need to pass it down. React will call your event handler when the user clicks the button.

Updating the screen

Often, you'll want your component to "remember" some information and display it. For example, maybe you want to count the number of times a button is clicked. To do this, add state to your component.

First, import useState from React:

import { useState } from 'react';

Now you can declare a state variable inside your component:

```
function MyButton() {
  const [count, setCount] = useState(0);

  // ...
```

You'll get two things from useState: the current state (count), and the function that lets you update it (setCount). You can give them any names, but the convention is to write [something, setSomething].

The first time the button is displayed, count will be 0 because you passed 0 to useState(). When you want to change state, call setCount() and pass the new value to it. Clicking this button will increment the counter:

```
function MyButton() {
  const [count, setCount] = useState(0);


  function handleClick() {
    setCount(count + 1);
  }


  return (
```

```
    <button onClick={handleClick}>

      Clicked {count} times

    </button>

  );

}
```

React will call your component function again. This time, count will be 1. Then it will be 2. And so on.

If you render the same component multiple times, each will get its own state. Click each button