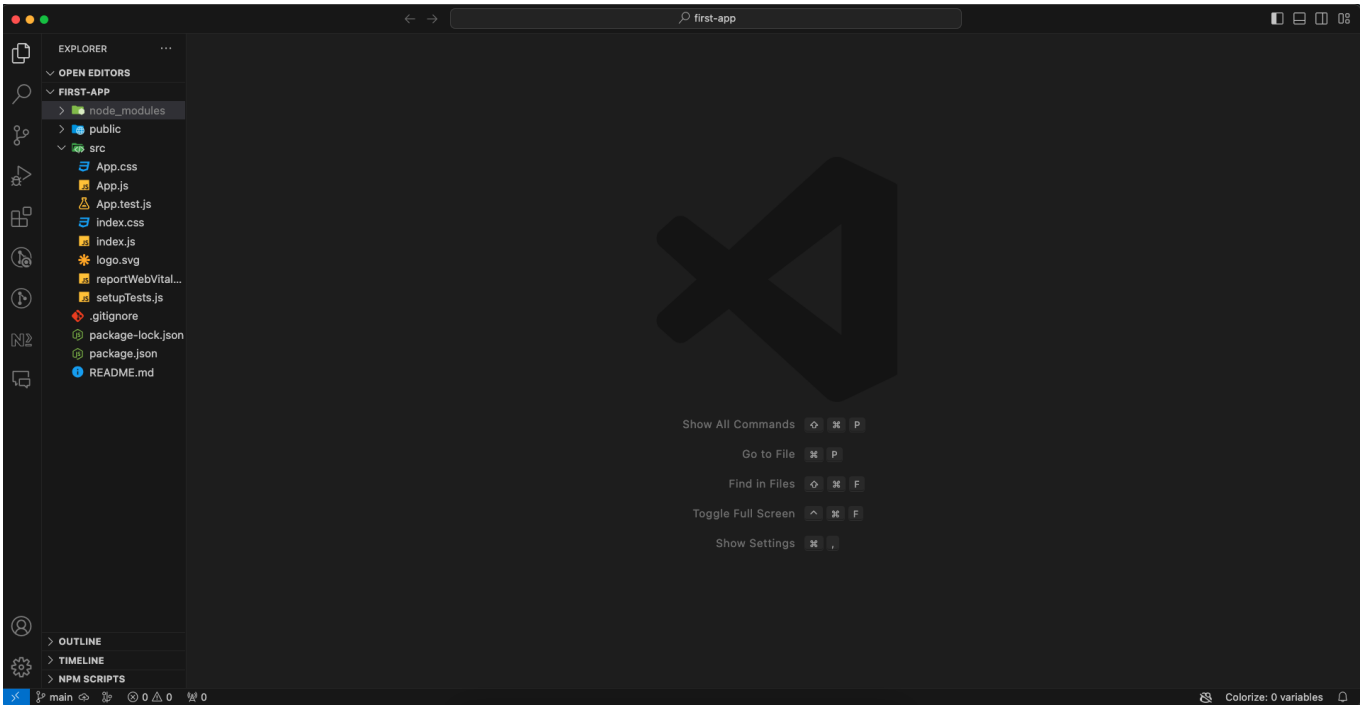
https://stackblitz.com/ or https://vscode.dev/

I. Creating your first React App

To create a project, using the command prompt or terminal run the following commands:

```
npx create-react-app my-app
cd my-app
npm start
```

Generated ReactJS Files after creating your first project



II. React Fundamentals

React Native runs on React, a popular open source library for building user interfaces with JavaScript. To make the most of React Native, it helps to understand React itself

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:

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

export default function MyApp() {
```

```
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton />
    </div>
  );
}
```

The `export default` keywords specify the main component in the file.

The `export` declaration is used to export values from a JavaScript module. Exported values can then be imported into other programs with the `import`.

In JavaScript, a default export is a way to share a single value, function.

Writing markup with JSX

The markup syntax you've seen above is called *JSX (Javascript XML)*. It is optional, but most React projects use JSX for its convenience.

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 `<>...</> (Fragment)` wrapper:

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

Adding styles

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

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

```
/* In your CSS */

.avatar {
  border-radius: 50%;
}
```

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

```
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,
        }}
      />
    </>
  );
}
```

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>
```

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.

```
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>;
}
```

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 separately:

"ExampleUseState.js" could not be found.

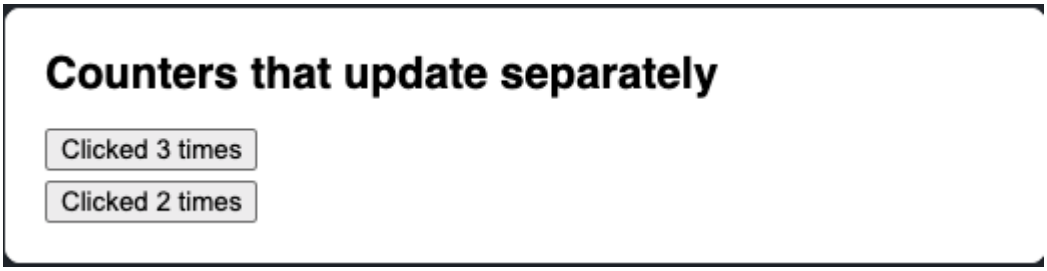
```jsx
import { useState } from 'react';

export default function MyApp() {
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

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

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

  return <button onClick={handleClick}>Clicked {count} times</button>;
}
```



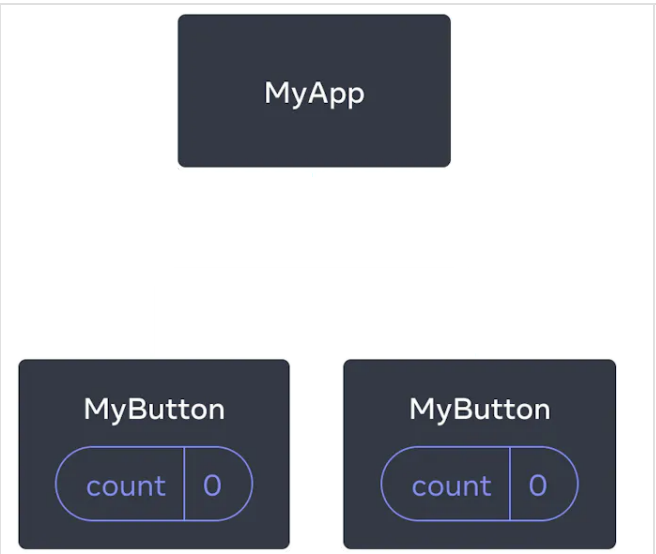
Using Hooks

Functions starting with `use` are called *Hooks*. `useState` is a built-in Hook provided by React. You can find other built-in Hooks in the API reference. You can also write your own Hooks by combining the existing ones.

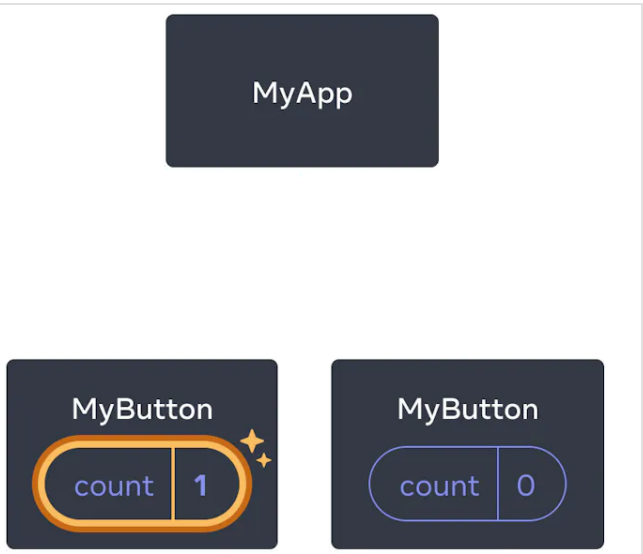
Hooks are more restrictive than other functions. You can only call Hooks *at the top* of your components (or other Hooks). If you want to use `useState` in a condition or a loop, extract a new component and put it there.

Sharing data between components

In the previous example, each `MyButton` had its own independent `count`, and when each button was clicked, only the `count` for the button clicked changed:



Initially, each `MyButton`'s `count` state is `0`

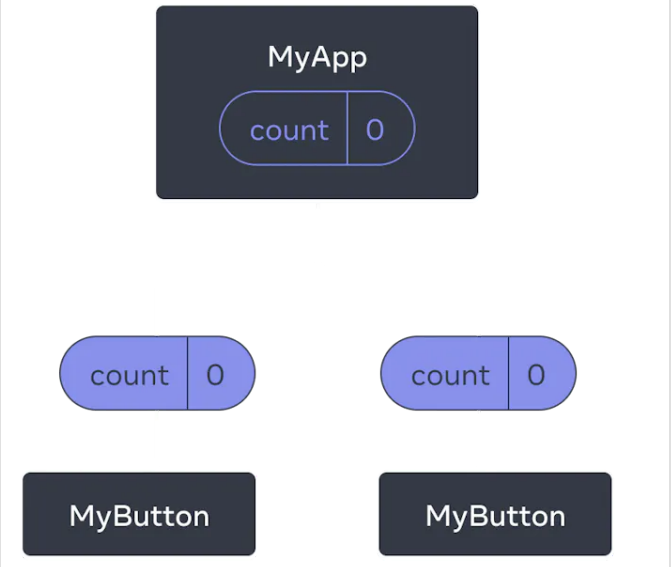


The first `MyButton` updates its `count` to `1`

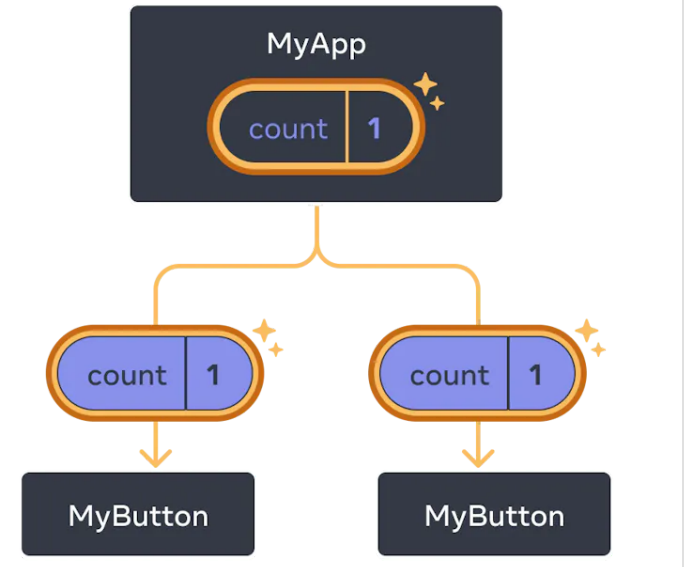
However, often you'll need components to *share data and always update together*.

To make both `MyButton` components display the same `count` and update together, you need to move the state from the individual buttons "upwards" to the closest component containing all of them.

In this example, it is `MyApp`:



Initially, `MyApp`'s `count` state is `0` and is passed down to both children



On click, `MyApp` updates its `count` state to `1` and passes it down to both children

Now when you click either button, the `count` in `MyApp` will change, which will change both of the counts in `MyButton`. Here's how you can express this in code.

First, *move the state up* from `MyButton` into `MyApp`:

```
export default function MyApp() {
  const [count, setCount] = useState(0);

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

  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

function MyButton() {
  // ... we're moving code from here ...
}```

Then, _pass the state down_ from `MyApp` to each `MyButton`, together with the shared click handler. You can pass information to `MyButton` using the JSX curly braces, just like you previously did with built-in tags like `<img>`:

```javascript
export default function MyApp() {
  const [count, setCount] = useState(0);

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

  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}
```
```

The information you pass down like this is called *props*. Now the `MyApp` component contains the `count` state and the `handleClick` event handler, and *passes both of them down as props* to each of the buttons.

Finally, change `MyButton` to *read* the props you have passed from its parent component:

```
function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}
```

When you click the button, the `onClick` handler fires. Each button's `onClick` prop was set to the `handleClick` function inside `MyApp`, so the code inside of it runs. That code calls `setCount(count + 1)`, incrementing the `count` state variable. The new `count` value is passed as a prop to each button, so they all show the new value. This is called "lifting state up". By moving state up, you've shared it between components.

```
import { useState } from 'react';

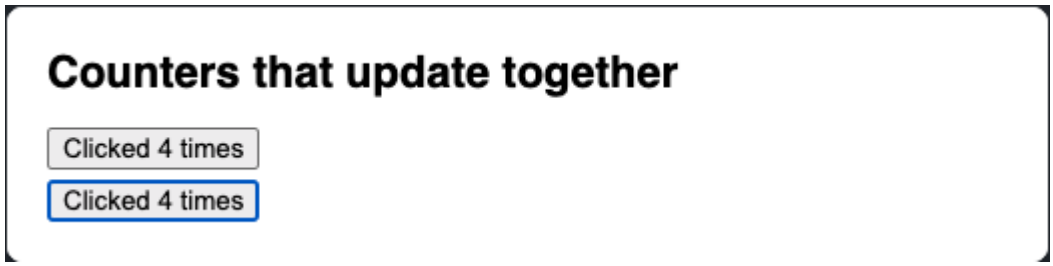
export default function MyApp() {
  const [count, setCount] = useState(0);

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

  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}

function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
```

```
  );
}
```



III. Built-in React Hooks

***Hooks* let you use different React features from your components. You can either use the built-in Hooks or combine them to build your own. This page lists all built-in Hooks in React.**

State Hooks

State lets a component "remember" information like user input. For example, a form component can use state to store the input value, while an image gallery component can use state to store the selected image index.

To add state to a component, use one of these Hooks:

useState

`useState` is a React Hook that lets you add a state variable to your component.

```
const [state, setState] = useState(initialState)
```

Usage

```
import { useState } from 'react';

function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

The convention is to name state variables like `[something, setSomething]` using array destructuring.

`useState` returns an array with exactly two items:

1. The current state of this state variable, initially set to the initial state you provided.
2. The `set` function that lets you change it to any other value in response to interaction.

`set` functions, like `setSomething(nextState)`

The `set` function returned by `useState` lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

To update what's on the screen, call the `set` function with some next state:

```
const [name, setName] = useState('Edward');

function handleClick() {
  setName('Taylor');
  setAge(a => a + 1);
  // ...
```

React will store the next state, render your component again with the new values, and update the UI.

useReducer

`useReducer` is a React Hook that lets you add a reducer to your component.
Reducer is consolidation of all state update logic in a single function.

```
const [state, dispatch] = useReducer(reducer, initialArg, init?)
```

Example

```
import { useReducer } from 'react';

function reducer(state, action) {
  // ...
```

```
}

function MyComponent() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });
  // ...

  function handleClick() {
        dispatch({ type: 'incremented_age' });
  }
```

`useReducer` returns an array with exactly two items:

1. The **`current state`** of this state variable, initially set to the initial state you provided.
2. The **`dispatch`** function that lets you change it in response to interaction.

To update what's on the screen, call `dispatch` with an object representing what the user did, called an *action*.

Parameters

- `reducer`: The reducer function that specifies how the state gets updated. It must be pure, should take the state and action as arguments, and should return the next state. State and action can be of any types.
- `initialArg`: The value from which the initial state is calculated. It can be a value of any type. How the initial state is calculated from it depends on the next `init` argument.
- **optional** `init`: The initializer function that should return the initial state. If it's not specified, the initial state is set to `initialArg`. Otherwise, the initial state is set to the result of calling `init(initialArg)`.

Returns

`useReducer` returns an array with exactly two values:

1. The current state. During the first render, it's set to `init(initialArg)` or `initialArg` (if there's no `init`).
2. The `dispatch` function that lets you update the state to a different value and trigger a re-render.

`dispatch` function

The `dispatch` function returned by `useReducer` lets you update the state to a different value and trigger a re-render. You need to pass the action as the only argument to the `dispatch` function:

```
const [state, dispatch] = useReducer(reducer, { age: 42 });



function handleClick() {

dispatch({ type: 'incremented_age' });

// ...
```

React will set the next state to the result of calling the reducer function you've provided with the current state and the action you've passed to dispatch.

React will pass the current state and the action to your reducer function. Your reducer will calculate and return the next state. React will store that next state, render your component with it, and update the UI.

Parameters

- `action`: The action performed by the user. It can be a value of any type. By convention, an action is usually an object with a `type` property identifying it and, optionally, other properties with additional information.

Usage of useReducer

Adding a reducer to a component

Call `useReducer` at the top level of your component to manage state with a reducer.

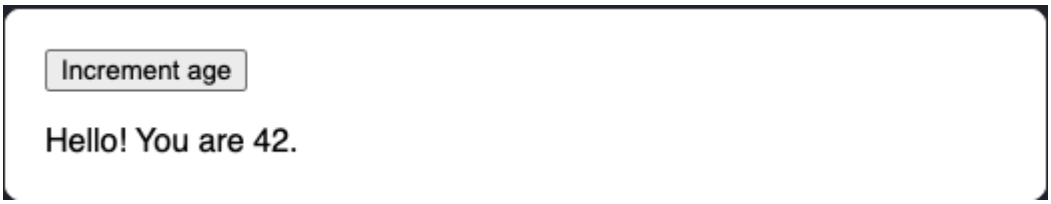
```
import { useReducer } from 'react';

function reducer(state, action) {
  if (action.type === 'incremented_age') {
```

```
    return {
      age: state.age + 1
    };
  }
  throw Error('Unknown action.');
}

export default function Counter() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });

  return (
    <>
      <button onClick={() => {
        dispatch({ type: 'incremented_age' })
      }}>
        Increment age
      </button>
      <p>Hello! You are {state.age}.</p>
    </>
  );
}
```



`useReducer` is very similar to `useState`, but it lets you move the state update logic from event handlers into a single function outside of your component.

Writing the reducer function

A reducer function is declared like this:

```
function reducer(state, action) {
  // ...
}
```

Then you need to fill in the code that will calculate and return the next state. By convention, it is common to write it as a `switch` statement. For each `case` in the `switch`, calculate and return

some next state.

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      return {
        name: state.name,
        age: state.age + 1
      };
    }
    case 'changed_name': {
      return {
        name: action.nextName,
        age: state.age
      };
    }
  }
  throw Error('Unknown action: ' + action.type);
}
```

Actions can have any shape. By convention, it's common to pass objects with a `type` property identifying the action. It should include the minimal necessary information that the reducer needs to compute the next state.

```
function Form() {
  const [state, dispatch] = useReducer(reducer, { name: 'Taylor', age: 42
});

  function handleButtonClick() {
    dispatch({ type: 'incremented_age' });
  }

  function handleInputChange(e) {
    dispatch({
      type: 'changed_name',
      nextName: e.target.value
    });
  }
  // ...
```

The action type names are local to your component. Each action describes a single interaction, even if that leads to multiple changes in data. The shape of the state is arbitrary, but usually it'll be an object or an array.

Pitfall

State is read-only. Don’t modify any objects or arrays in state:

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // 🚩 Don't mutate an object in state like this:
      state.age = state.age + 1;
      return state;
    }
```

Instead, always return new objects from your reducer:

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // ✅ Instead, return a new object
      return {
        ...state,
        age: state.age + 1
      };
    }
```

Context Hooks

Context lets a component receive information from distant parents without passing it as props. For example, your app’s top-level component can pass the current UI theme to all components below, no matter how deep.

```
import { useContext } from 'react';

function MyComponent() {
  const theme = useContext(ThemeContext);
  // ...
```

Parameters

- `SomeContext`: The context that you've previously created with `createContext`. The context itself does not hold the information, it only represents the kind of information you can provide or read from components.

Returns

`useContext` returns the context value for the calling component. It is determined as the `value` passed to the closest `SomeContext.Provider` above the calling component in the tree. If there is no such provider, then the returned value will be the `defaultValue` you have passed to `createContext` for that context. The returned value is always up-to-date. React automatically re-renders components that read some context if it changes.

Usage

Passing data deeply into the tree

Call `useContext` at the top level of your component to read and subscribe to context.

```
import { useContext } from 'react';

function Button() {
  const theme = useContext(ThemeContext);
  // ...
```

`useContext` returns the context value for the context you passed. To determine the context value, React searches the component tree and finds **the closest context provider above** for that particular context.

To pass context to a `Button`, wrap it or one of its parent components into the corresponding context provider:

```
function MyPage() {
  return (
    <ThemeContext.Provider value="dark">
      <Form />
    </ThemeContext.Provider>
  );
}

function Form() {
```

```
  // ... renders buttons inside ...
}
```

Example

`App.js`

```
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  return (
    <ThemeContext.Provider value="light">
      <Form />
    </ThemeContext.Provider>
  )
}

function Form() {
  return (
    <Panel title="Welcome">
      <Button>Sign up</Button>
      <Button>Log in</Button>
    </Panel>
  );
}

function Panel({ title, children }) {
  const theme = useContext(ThemeContext);
  const className = 'panel-' + theme;
  return (
    <section className={className}>
      <h1>{title}</h1>
      {children}
    </section>
  )
}

function Button({ children }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;
  return (
    <button className={className}>
```

```
      {children}
    </button>
  );
}
```

style.css

```
* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

h1 {
  margin-top: 0;
  font-size: 22px;
}

h2 {
  margin-top: 0;
  font-size: 20px;
}

h3 {
  margin-top: 0;
  font-size: 18px;
}

h4 {
  margin-top: 0;
  font-size: 16px;
}

h5 {
  margin-top: 0;
  font-size: 14px;
}

h6 {
  margin-top: 0;
```

```
  font-size: 12px;
}

code {
  font-size: 1.2em;
}

ul {
  padding-inline-start: 20px;
}

.panel-light,
.panel-dark {
  border: 1px solid black;
  border-radius: 4px;
  padding: 20px;
}
.panel-light {
  color: #222;
  background: #fff;
}

.panel-dark {
  color: #fff;
  background: rgb(23, 32, 42);
}

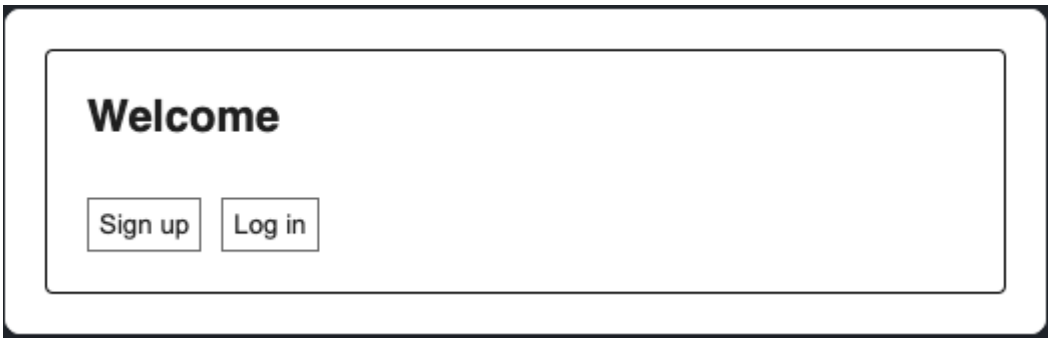
.button-light,
.button-dark {
  border: 1px solid #777;
  padding: 5px;
  margin-right: 10px;
  margin-top: 10px;
}

.button-dark {
  background: #222;
  color: #fff;
}

.button-light {
  background: #fff;
  color: #222;
}
```

```
<ThemeContext.Provider value="dark">
  <Form />
</ThemeContext.Provider>
```


```
<ThemeContext.Provider value="light">
  <Form />
</ThemeContext.Provider>
```



Ref Hooks

Refs let a component hold some information that isn't used for rendering, like a DOM node or a timeout ID. Unlike with state, updating a ref does not re-render your component. Refs are an "escape hatch" from the React paradigm. They are useful when you need to work with non-React systems, such as the built-in browser APIs.

- `useRef` declares a ref. You can hold any value in it, but most often it's used to hold a DOM node.
- `useImperativeHandle` lets you customize the ref exposed by your component. This is rarely used.

```
function Form() {
  const inputRef = useRef(null);
  // ...
```

useRef

`useRef` is a React Hook that lets you reference a value that's not needed for rendering.

```
const ref = useRef(initialValue)
```

Parameters

- `initialValue`: The value you want the ref object's `current` property to be initially. It can be a value of any type. This argument is ignored after the initial render.

Returns

`useRef` returns an object with a single property:

- `current`: Initially, it's set to the `initialValue` you have passed. You can later set it to something else. If you pass the ref object to React as a `ref` attribute to a JSX node, React will set its `current` property.

On the next renders, `useRef` will return the same object.

Usage

Referencing a value with a ref

```
import { useRef } from 'react';

function Stopwatch() {
  const intervalRef = useRef(0);
  // ...
```

`useRef` returns a ref object with a single `current` property initially set to the initial value you provided.

On the next renders, `useRef` will return the same object. You can change its `current` property to store information and read it later. This might remind you of state, but there is an important difference.

Changing a ref does not trigger a re-render. This means refs are perfect for storing information that doesn't affect the visual output of your component. For example, if you need to store an interval ID and retrieve it later, you can put it in a ref. To update the value inside the ref, you need to manually change its `current` property:

```
function handleStartClick() {
  const intervalId = setInterval(() => {
    // ...
  }, 1000);
  intervalRef.current = intervalId;
}
```

Later, you can read that interval ID from the ref so that you can call clear that interval:

```
function handleStopClick() {
  const intervalId = intervalRef.current;
  clearInterval(intervalId);
}
```

By using a ref, you ensure that:

- You can **store information** between re-renders (unlike regular variables, which reset on every render).
- Changing it **does not trigger a re-render** (unlike state variables, which trigger a re-render).
- The **information is local** to each copy of your component (unlike the variables outside, which are shared).

Changing a ref does not trigger a re-render, so refs are not appropriate for storing information you want to display on the screen. Use state for that instead.

Manipulating the DOM with a ref

It's particularly common to use a ref to manipulate the DOM. React has built-in support for this.

```
import { useRef } from 'react';
```

```
function MyComponent() {
  const inputRef = useRef(null);
  // ...
```

Then pass your ref object as the `ref` attribute to the JSX of the DOM node you want to manipulate:

```
  // ...
  return <input ref={inputRef} />;
```

After React creates the DOM node and puts it on the screen, React will set the current property of your ref object to that DOM node. Now you can access the  's DOM node and call methods like focus():

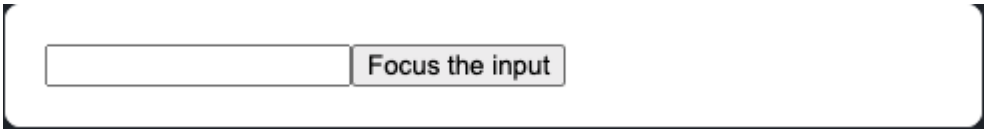
```
 function handleClick() {
    inputRef.current.focus();
  }
```

```
import { useRef } from 'react';

export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```



Effect Hooks

Effects let a component connect to and synchronize with external systems.This includes dealing with network, browser DOM, animations, widgets written using a different UI library, and other non-React code.

- [`useEffect`]connects a component to an external system.

```
function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);
  // ...
```

Effects are an “escape hatch” from the React paradigm. Don’t use Effects to orchestrate the data flow of your application. If you’re not interacting with an external system, [you might not need an Effect.]

useEffect

`useEffect` is a React Hook that lets you [synchronize a component with an external system.]

```
useEffect(setup, dependencies?)
```

Reference

`useEffect(setup, dependencies?)`

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
```

```
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

Parameters

- `setup`: The function with your Effect's logic. Your setup function may also optionally return a *cleanup* function. When your component is added to the DOM, React will run your setup function. After every re-render with changed dependencies, React will first run the cleanup function (if you provided it) with the old values, and then run your setup function with the new values. After your component is removed from the DOM, React will run your cleanup function.
- **optional** `dependencies`: The list of all reactive values referenced inside of the `setup` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body.

Usage

Connecting to an external system

`App.js`

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [roomId, serverUrl]);

  return (
    <>
      <label>
```

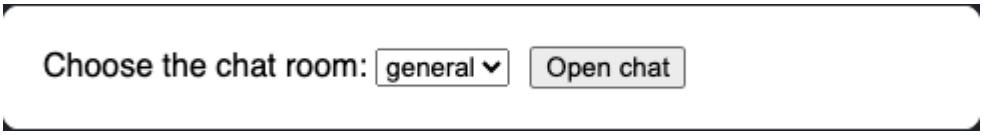
```
        Server URL:{' '}
        <input
          value={serverUrl}
          onChange={e => setServerUrl(e.target.value)}
        />
      </label>
      <h1>Welcome to the {roomId} room!</h1>
    </>
  );
}

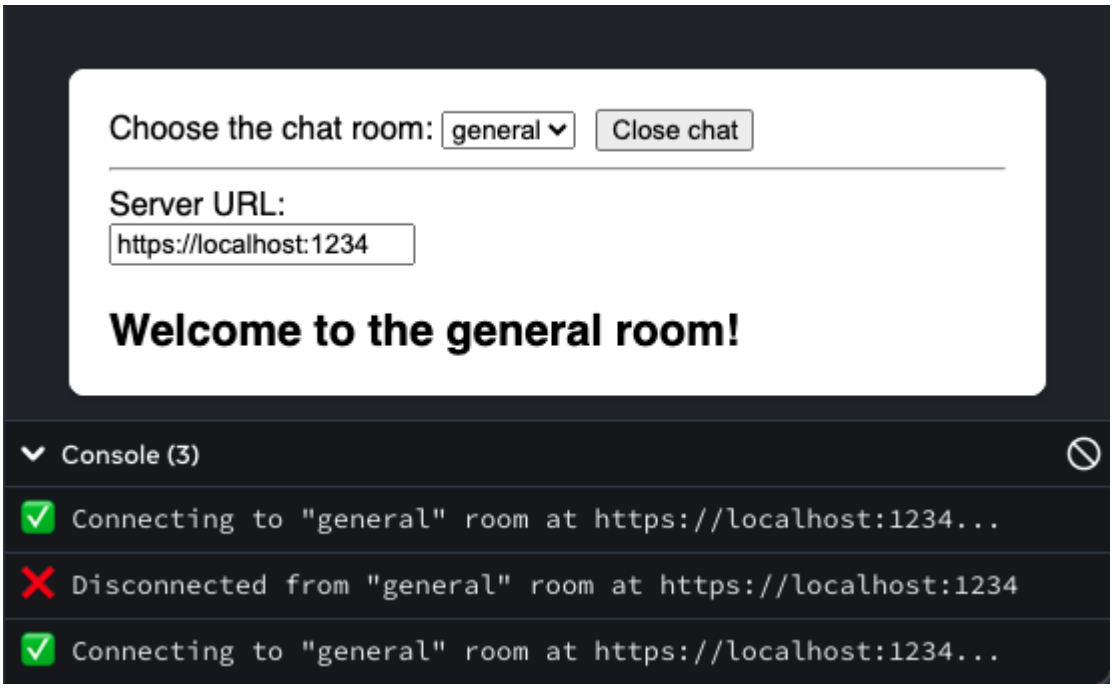
export default function App() {
  const [roomId, setRoomId] = useState('general');
  const [show, setShow] = useState(false);
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <button onClick={() => setShow(!show)}>
        {show ? 'Close chat' : 'Open chat'}
      </button>
      {show && <hr />}
      {show && <ChatRoom roomId={roomId} />}
    </>
  );
}
```

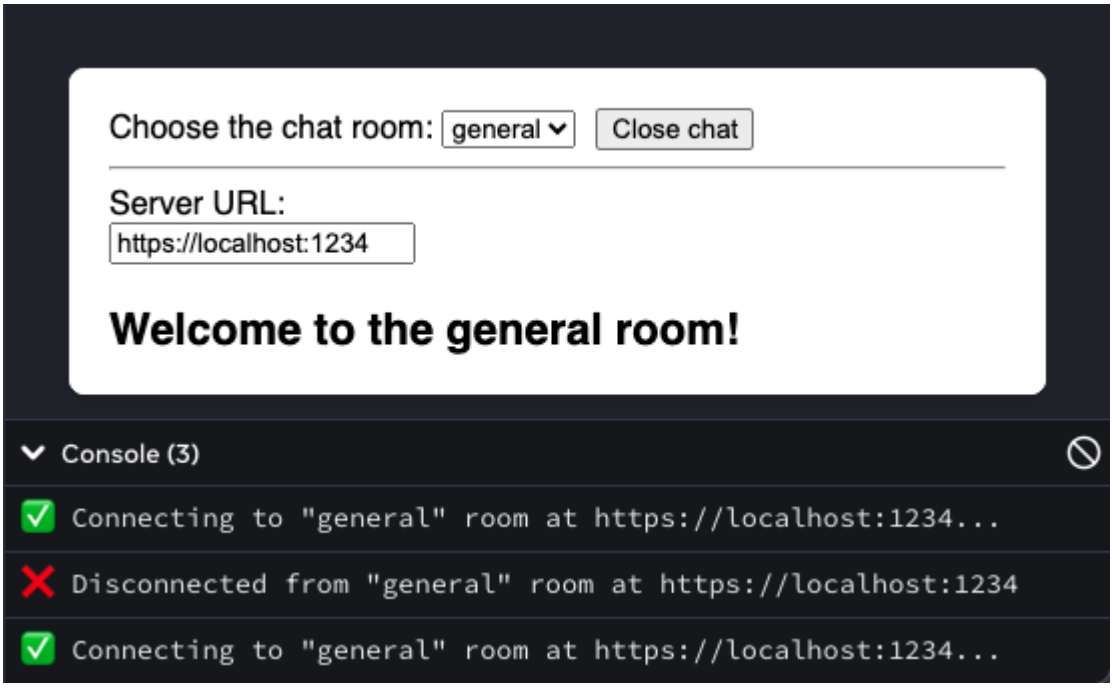
chat.js

```
export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting to "' + roomId + '" room at ' + serverUrl +
'...');
```

```
    },
    disconnect() {
      console.log('❌ Disconnected from "' + roomId + '" room at ' +
serverUrl);
    }
  };
}
```







Performance Hooks

A common way to optimize re-rendering performance is to skip unnecessary work. For example, you can tell React to reuse a cached calculation or to skip a re-render if the data has not changed since the previous render.

To skip calculations and unnecessary re-rendering, use one of these Hooks:

- `useMemo` lets you cache the result of an expensive calculation.
- `useCallback` lets you cache a function definition before passing it down to an optimized component.