REACTJS

https://react.dev/learn

What is React.js?

- React is a JavaScript library for building user interfaces.
- React is used to build single-page applications.
- React allows us to create reusable UI components.

Required environment: Node.js must be installed

Why Use React.js?

- Component-based architecture.
- Virtual DOM improves performance.
- Reusable components save development time.
- Strong community support and ecosystem.

Companies Using React.js

- Facebook
- Instagram
- WhatsApp
- Airbnb
- Netflix

Start a New React Project

Create new react project using Next.js.

Next.js **is a full-stack React framework.** It's versatile and lets you create React apps of any size—from a mostly static blog to a complex dynamic application. To create a new Next.js project, run in your terminal:

```
npx create-next-app@latest
```

OR

```
npx create-next-app project-name
cd project-name
npm run dev
```

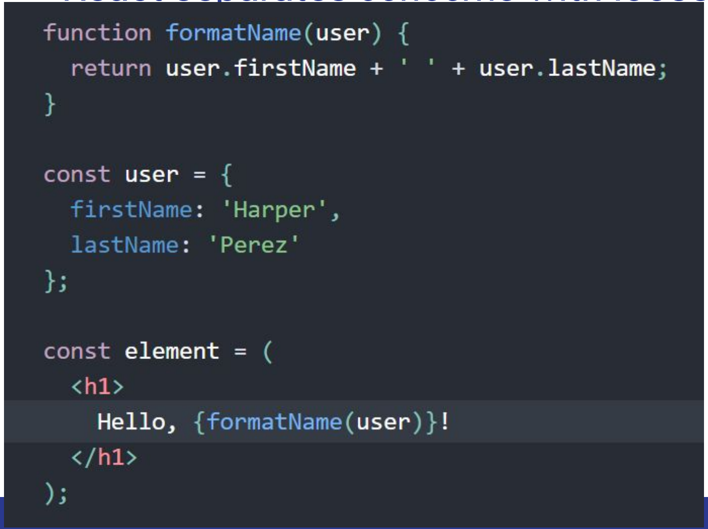
https://nextjs.org/docs/app/getting-started/installation

Key Features of React.js

- JSX (JavaScript XML)
- Virtual DOM
- Components and Props
- State Management
- Hooks
- React Router

Understanding JSX

- JSX is a syntax extension for JavaScript.
- Allows writing HTML-like code inside JavaScript.
- Instead of artificially separating *technologies* by putting markup and logic in separate files, React separates *concerns* with loosely coupled units called “components” that contain both.



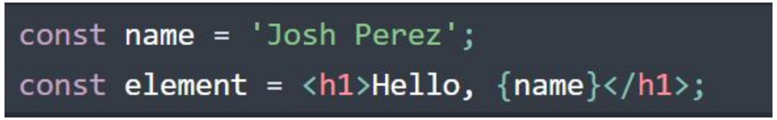
```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);
```

Embedding Expressions in JSX

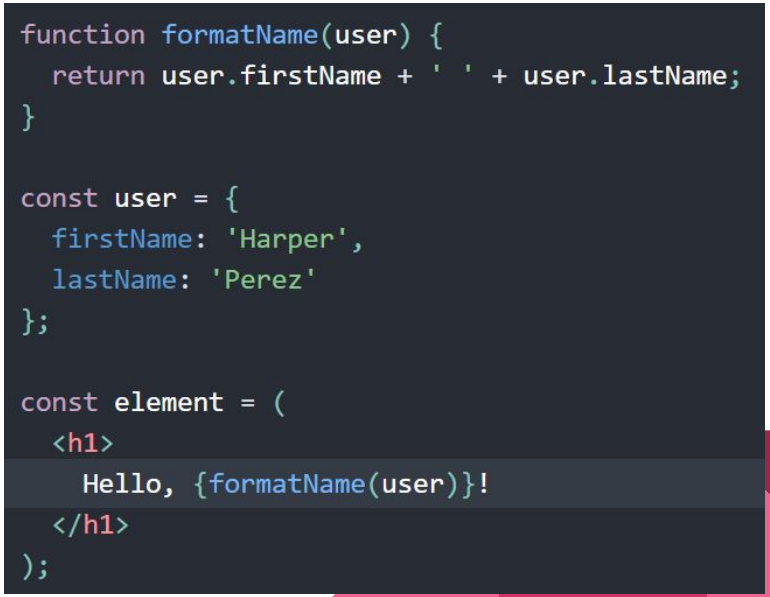
You can put any valid JavaScript expression inside the curly braces in JSX.

For example, `2 + 2`, `user.firstName`, or `formatName(user)` are all valid JavaScript expressions.

In the example below, we embed the result of calling a JavaScript function, `formatName(user)`, into an `<h1>` element.



```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

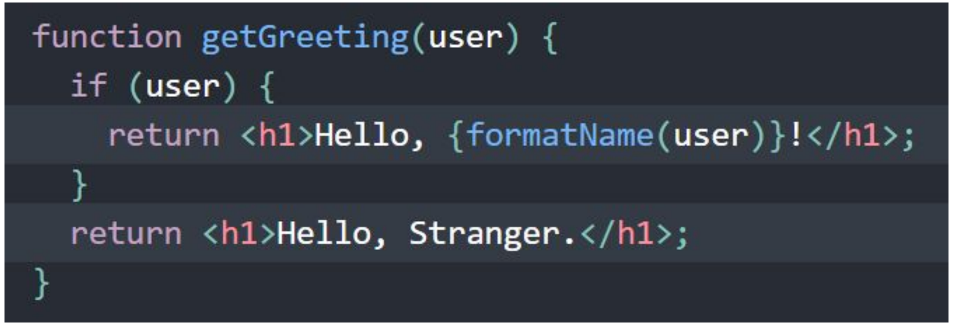


```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);
```

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

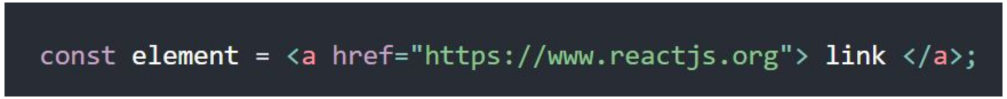
This means that you can use JSX inside of `if` statements and `for` loops, assign it to variables, accept it as arguments, and return it from functions:



```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

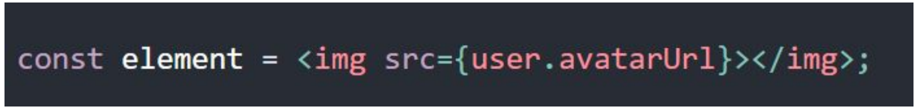
Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

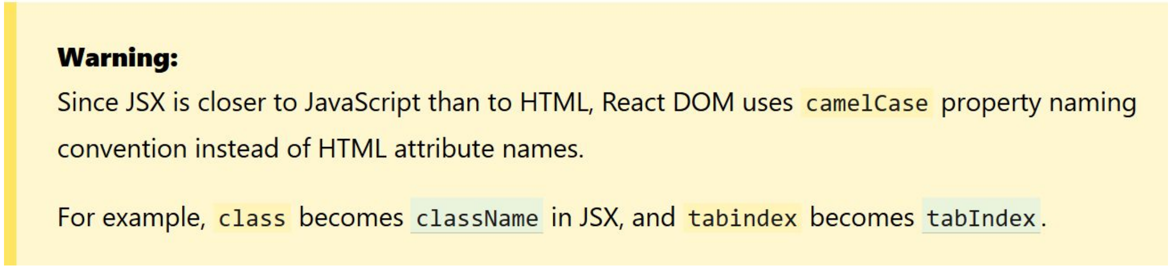


```
const element = <a href="https://www.reactjs.org"> link </a>;
```

You may also use curly braces to embed a JavaScript expression in an attribute:



```
const element = <img src={user.avatarUrl}></img>;
```

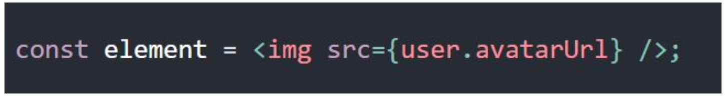


Warning:
Since JSX is closer to JavaScript than to HTML, React DOM uses `camelCase` property naming convention instead of HTML attribute names.

For example, `class` becomes `className` in JSX, and `tabindex` becomes `tabIndex`.

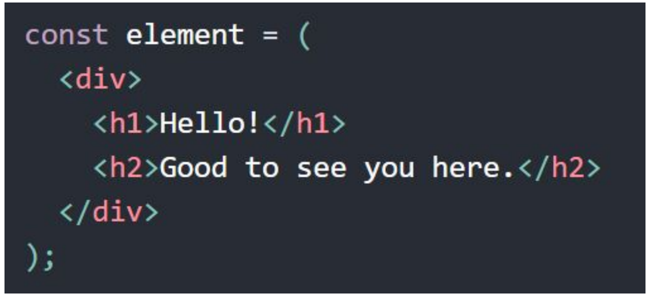
Specifying Children with JSX

If a tag is empty, you may close it immediately with `/>`, like XML:



```
const element = <img src={user.avatarUrl} />;
```

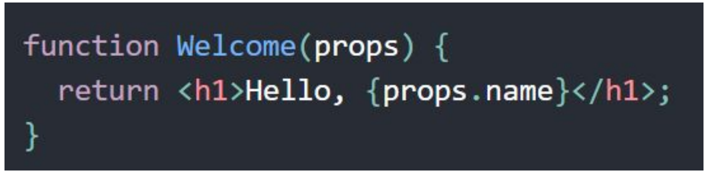
SX tags may contain children:



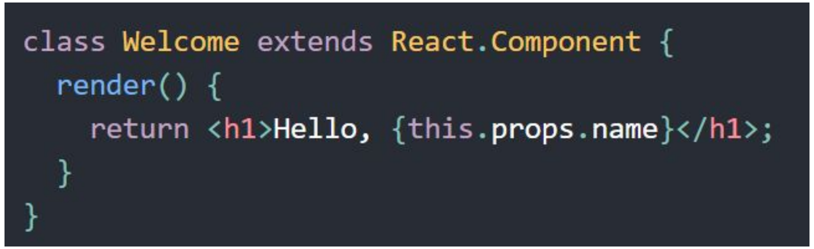
```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

Components in React

- Components are the building blocks of a React application.
- Two types:
    - Functional Components
    - Class Components



```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```



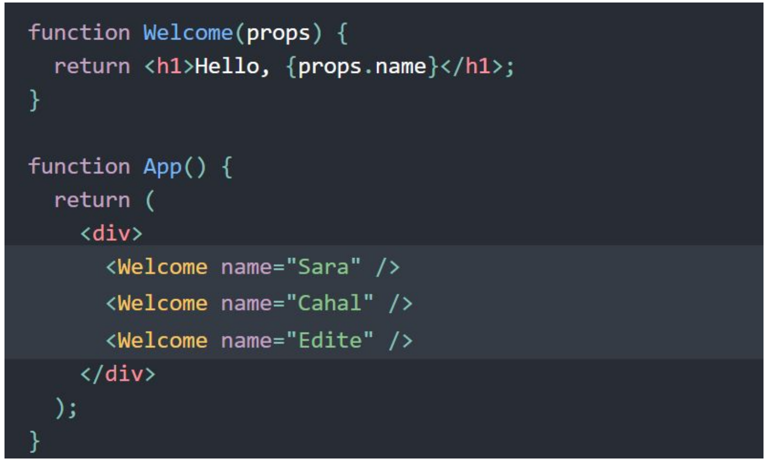
```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

The above two components are equivalent from React's point of view.

Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail.

For example, we can create an `App` component that renders `Welcome` many times:



```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

Extracting Components

Don’t be afraid to split components into smaller components.

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

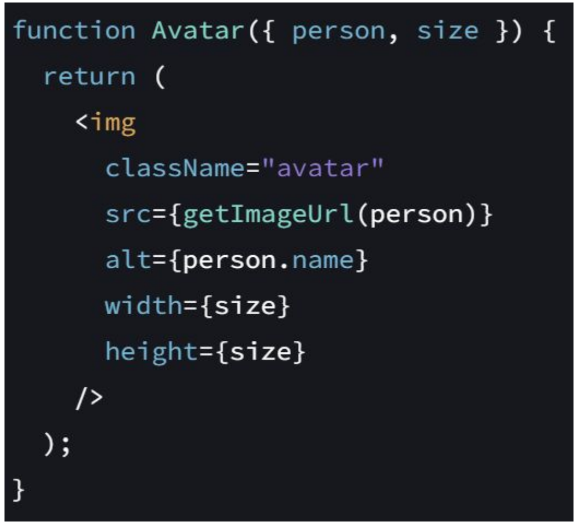
Props in React

React components use props to communicate with each other.

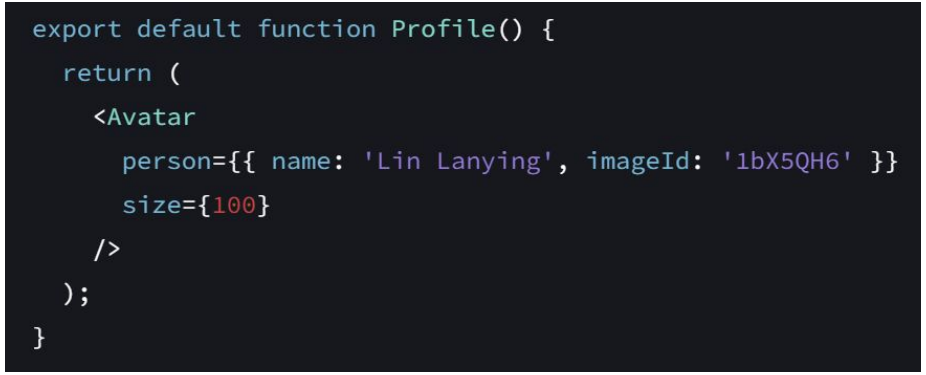
Every parent component can pass some information to its child components by giving them props.

Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, and functions.

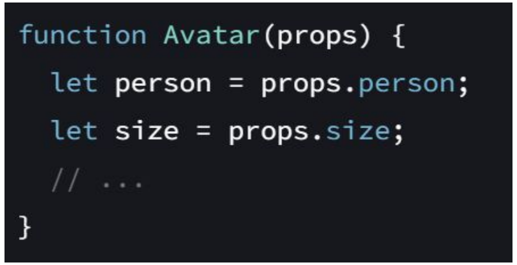
Pass props to the child component and Read props



```
function Avatar({ person, size }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}
```



```
export default function Profile() {
  return (
    <Avatar
      person={{ name: 'Lin Lanying', imageId: '1bX5QH6' }}
      size={100}
    />
  );
}
```



```
function Avatar(props) {
  let person = props.person;
  let size = props.size;
  // ...
}
```

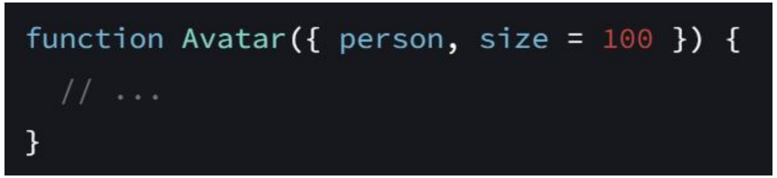
This syntax is called "Destructuring"

Specifying a default value for a prop

If you want to give a prop a default value to fall back on when no value is specified, you can do it with the destructuring by putting = and the default value right after the parameter

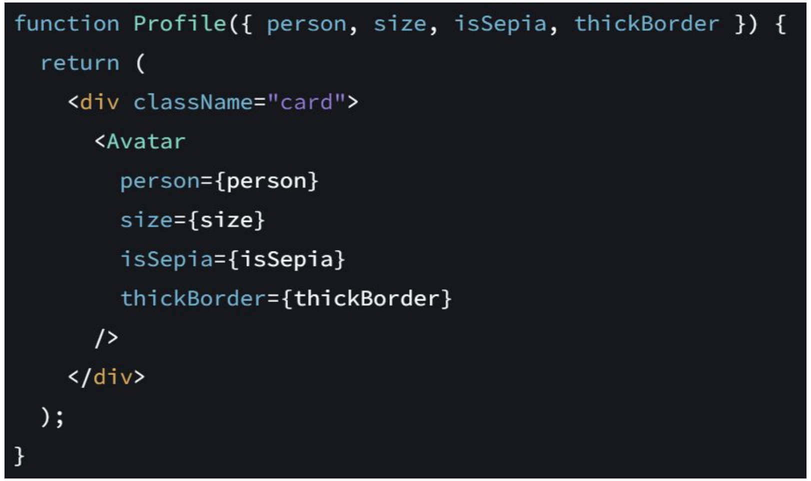
Now, if <Avatar person={...} /> is rendered with no size prop, the size will be set to 100.

The default value is only used if the size prop is missing or if you pass size={undefined}. But if you pass size={null} or size={0}, the default value will not be used.



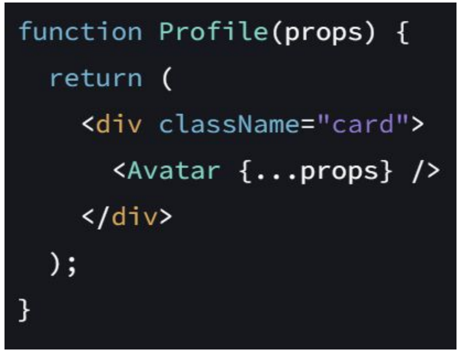
```
function Avatar({ person, size = 100 }) {
  // ...
}
```

Forwarding props with the JSX spread syntax



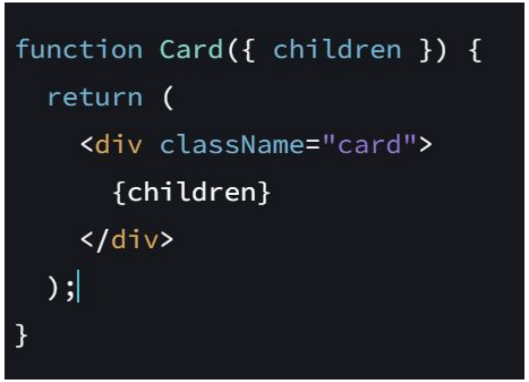
```
function Profile({ person, size, isSepia, thickBorder }) {
  return (
    <div className="card">
      <Avatar
        person={person}
        size={size}
        isSepia={isSepia}
        thickBorder={thickBorder}
      />
    </div>
  );
}
```

=

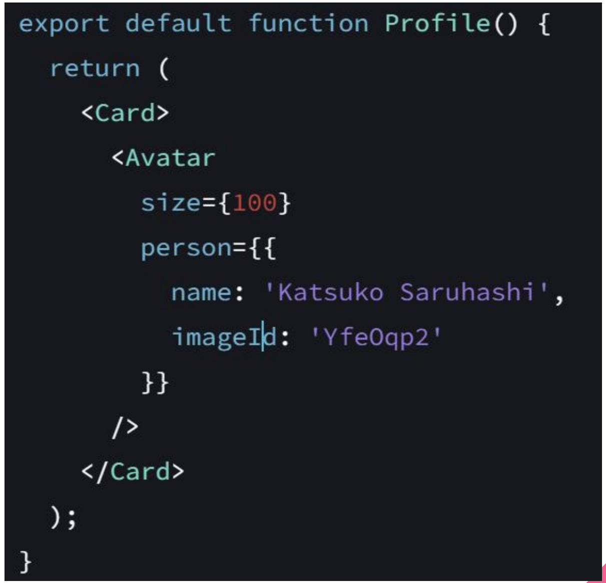


```
function Profile(props) {
  return (
    <div className="card">
      <Avatar {...props} />
    </div>
  );
}
```

Passing JSX as children



```
function Card({ children }) {
  return (
    <div className="card">
      {children}
    </div>
  );
}
```

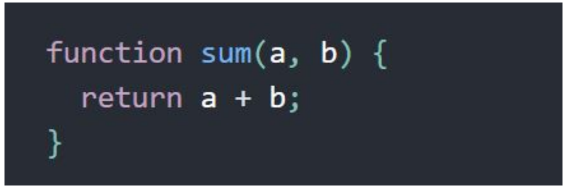


```
export default function Profile() {
  return (
    <Card>
      <Avatar
        size={100}
        person={{
          name: 'Katsuko Saruhashi',
          imageId: 'YfeOqp2'
        }}
      />
    </Card>
  );
}
```

Props are Read-Only

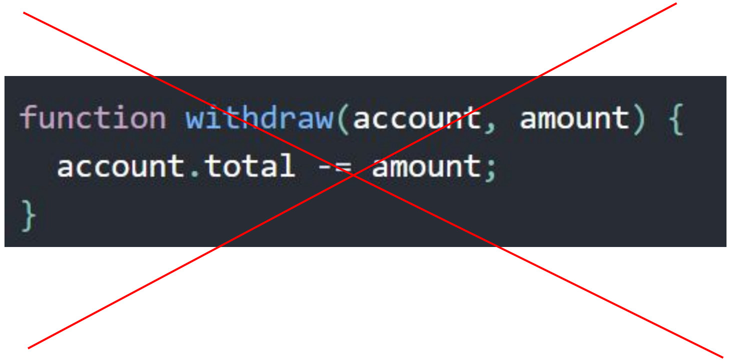
Whether you declare a component as a function or a class, it must never modify its own props.

All React components must act like pure functions with respect to their props.



```
function sum(a, b) {
  return a + b;
}
```

This function is called "pure" because they do not attempt to change their inputs, and always return the same result for the same inputs



```
function withdraw(account, amount) {
  account.total -= amount;
}
```

React Lifecycle Methods (Class Components)

Your home work

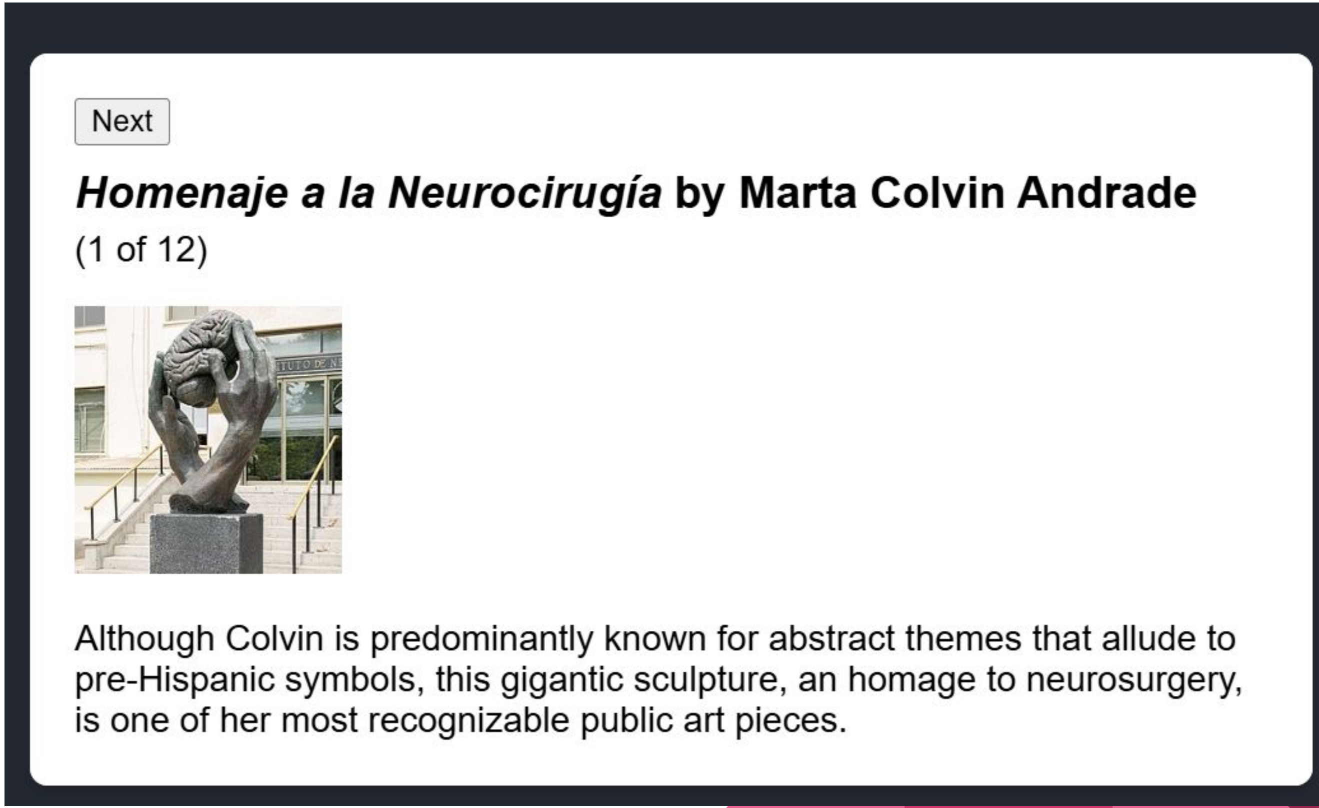
State in React - A component’s memory

Examples:

- Type in to the form => remember the current input value
- Click next on the image carousel => remember current image
- Click add product to cart => the shopping cart

Example

```jsx
import { sculptureList } from './data.js';

export default function Gallery() {
  let index = 0;

  function handleClick() {
    index = index + 1;
  }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
      </h2>
      <h3>
        ({index + 1} of {sculptureList.length})
      </h3>
      <img
        src={sculpture.url}
        alt={sculpture.alt}
      />
      <p>
        {sculpture.description}
      </p>
    </>
  );
}
```



Next

Homenaje a la Neurocirugía **by Marta Colvin Andrade**

(1 of 12)

Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.

Why nothing happen when click the Next button?

- **Local variables don’t persist between renders.** When React renders this component a second time, it renders it from scratch—it doesn’t consider any changes to the local variables.

- **Changes to local variables won’t trigger renders.** React doesn’t realize it needs to render the component again with the new data.

How to update component with new data?

- Retain the data between renders.
- Trigger React to render the component with new data (re-rendering).

The useState Hook

- A state variable to retain the data between renders.
- A state setter function to update the variable and trigger React to render the component again.

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

```
const [index, setIndex] = useState(0);
```

```
function handleClick() {
  setIndex(index + 1);
}
```

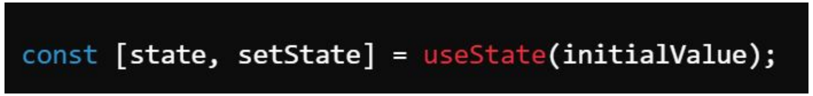
useState is a Hook

In React, useState, as well as any other function starting with “use”, is called a Hook.

Hooks are special functions that are only available while React is rendering.

State is just one of those features, but you will meet the other Hooks later.

Syntax



```
const [state, setState] = useState(initialValue);
```

`state`: The current state value.

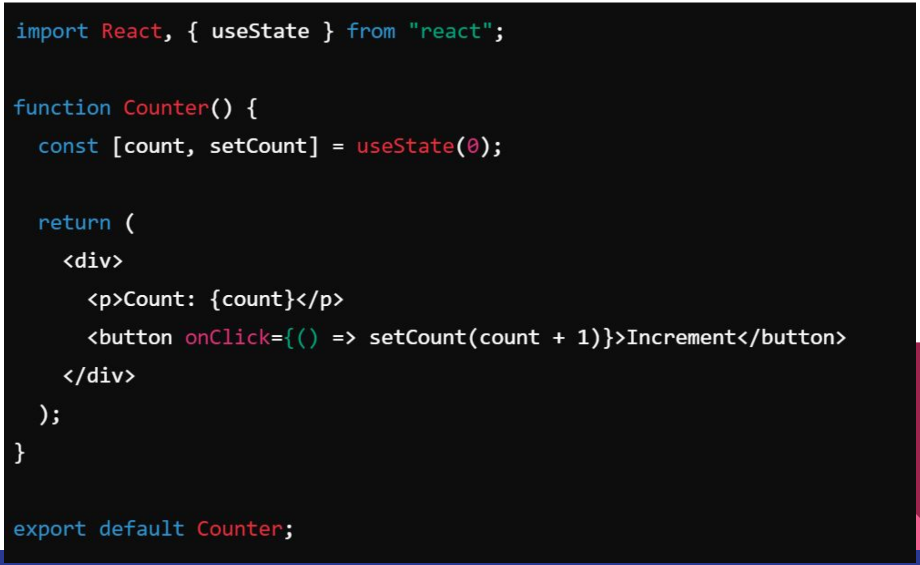
`setState`: A function to update the state.

`initialValue`: The starting value of the state.

`Example:`
`useState(0)` initializes `count` with `0`.

`setCount(count + 1)` updates the state when the button is clicked.

The component re-renders whenever the state updates.



```
import React, { useState } from "react";

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

Updating State

Using a Function (Best Practice for Previous State)

If your new state depends on the previous state, use a function inside `setState`. This ensures you get the latest state, especially in asynchronous updates.

```
setCount(prevCount => prevCount + 1);
```

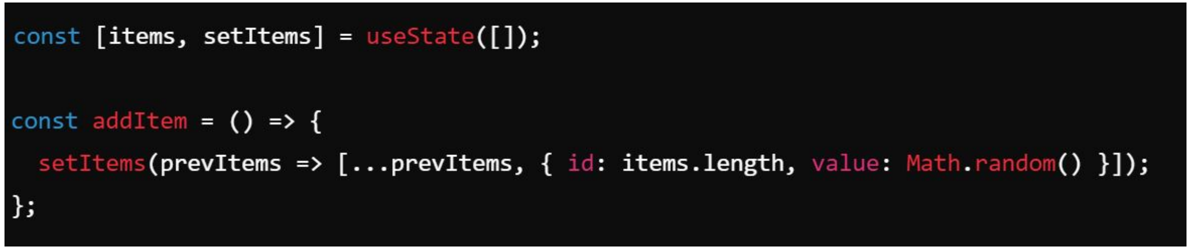
Updating an Object in State

Using `{ ...prevUser }` keeps other properties intact while updating `age`.

```
const [user, setUser] = useState({ name: "John", age: 25 });

const updateAge = () => {
  setUser(prevUser => ({ ...prevUser, age: prevUser.age + 1 }));
};
```

State with Arrays

Spread operator (`...prevItems`) ensures the array is not mutated directly.

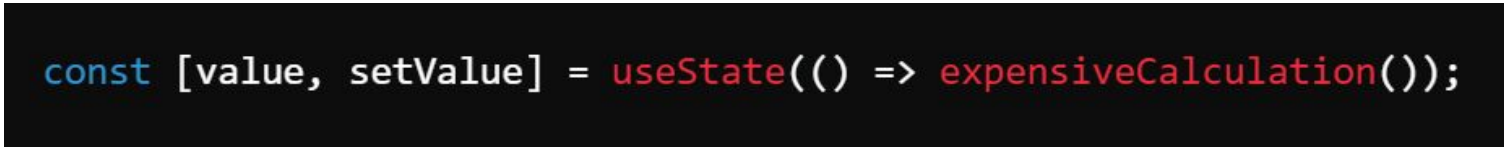


```
const [items, setItems] = useState([]);

const addItem = () => {
  setItems(prevItems => [...prevItems, { id: items.length, value: Math.random() }]);
};
```

Lazy Initialization

You can pass a function to `useState` to compute the initial state only once:

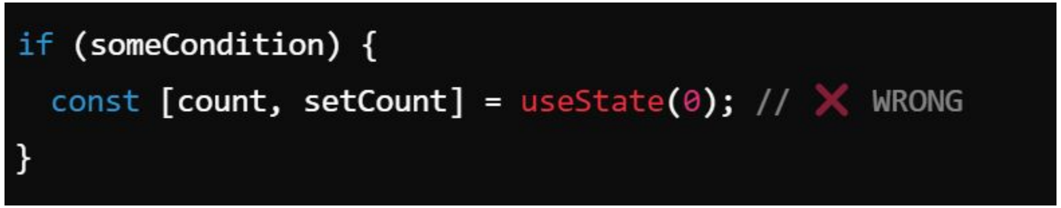
This prevents unnecessary calculations on every render.



```
const [value, setValue] = useState(() => expensiveCalculation());
```

Rules of useState

1. **Must be inside a function component** – Hooks cannot be used in class components or outside component functions.
2. **Cannot be used conditionally** – Hooks should always be called in the same order.



```
if (someCondition) {
  const [count, setCount] = useState(0); // ❌ WRONG
}
```

Instead, declare hooks at the top level.

When to Use useState

When you need to store and update local state in a component.

When you want reactive UI updates.

useState - Conclusion

`useState` is a powerful and easy-to-use hook for managing state in React functional components.

Understanding how it works helps in building interactive and dynamic applications.

Hooks

- Introduced in React 16.8.
- Allows functional components to use state and lifecycle features.
- Common hooks:
  - `useState`
  - `useEffect`
  - `useContext`
  - `useRef`
  - `useEffect`
  - `useCallback`
  - `useMemo`

Hook - Effect hook

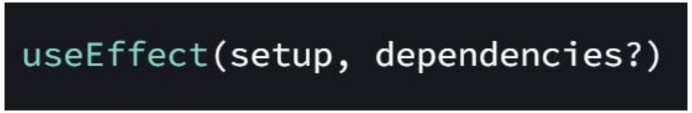
Effects let a component connect to and synchronize with external systems. This includes dealing with network, browser DOM, animations, widgets written using a different UI library, and other non-React code.

- useEffect connects a component to an external system.

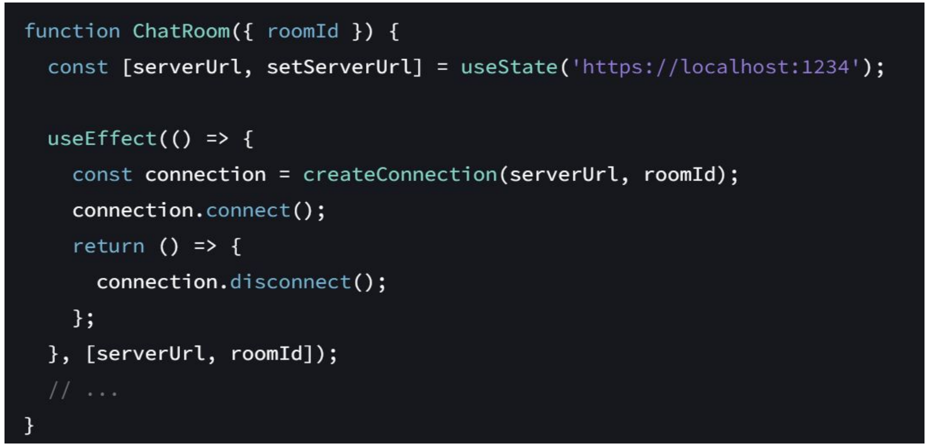
There are two rarely used variations of useEffect with differences in timing:

- useLayoutEffect fires before the browser repaints the screen.
- useInsertionEffect fires before React makes changes to the DOM.

Effect Hook - useEffect



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



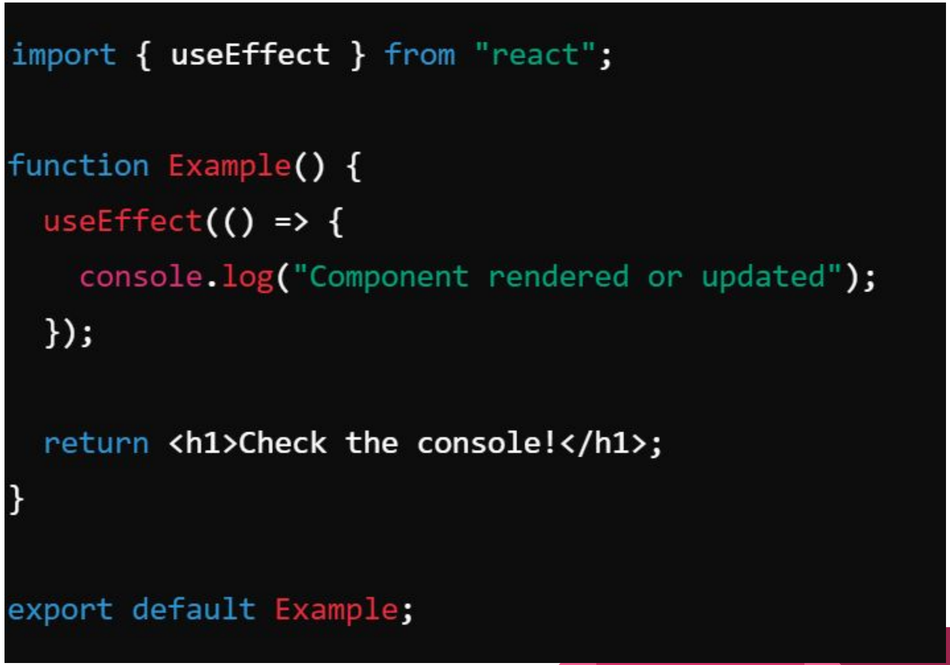
```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

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

- useEffect is a React Hook that lets you synchronize a component with an external system.
- Call useEffect at the top level of your component to declare an Effect
- setup: The function with your Effect’s logic. Your setup function may also optionally return a cleanup function.
- optional dependencies: The list of all reactive values referenced inside of the setup code.
- useEffect returns undefined.

useEffect

Run on Every Render

If no dependency array is provided, `useEffect` runs after every render.

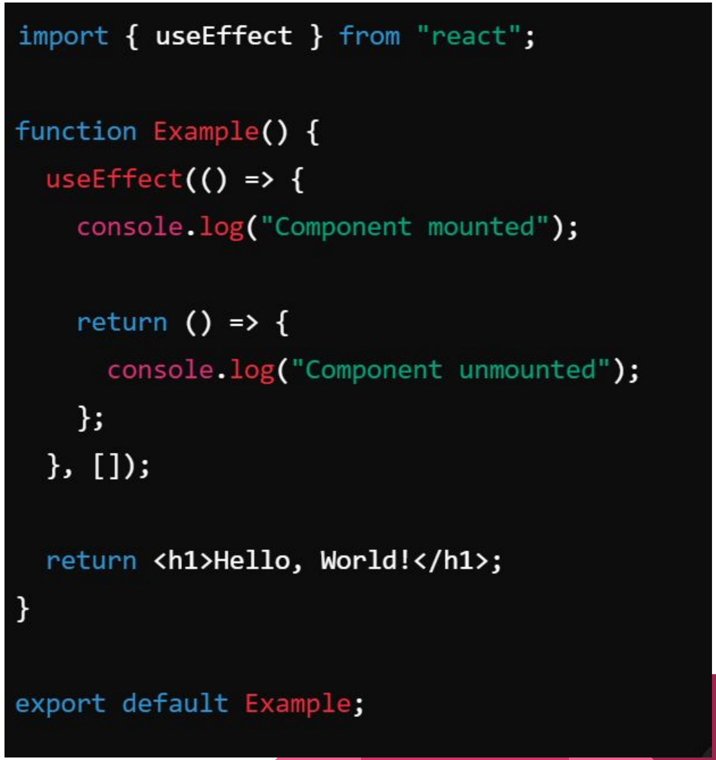


```
import { useEffect } from "react";

function Example() {
  useEffect(() => {
    console.log("Component rendered or updated");
  });

  return <h1>Check the console!</h1>;
}

export default Example;
```

useEffect

Run Only Once (on Mount)

Adding an empty dependency array ([ ]) ensures the effect runs only once when the component mounts.

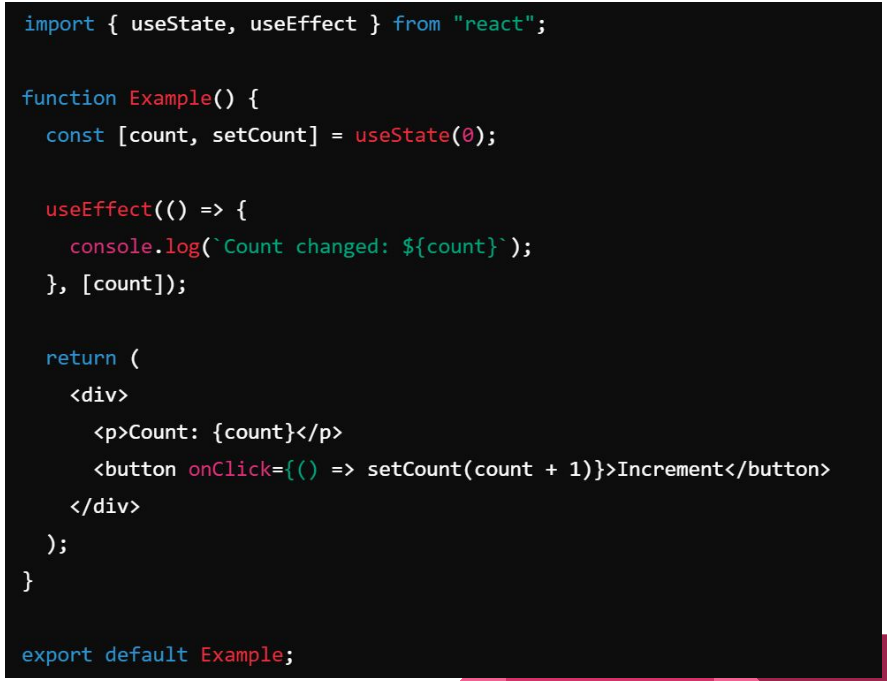


```
import { useEffect } from "react";

function Example() {
  useEffect(() => {
    console.log("Component mounted");

    return () => {
      console.log("Component unmounted");
    };
  }, []);

  return <h1>Hello, World!</h1>;
}

export default Example;
```

useEffect

Run When a Specific Value Changes

Provide dependencies in the array to run `useEffect` only when specific values change

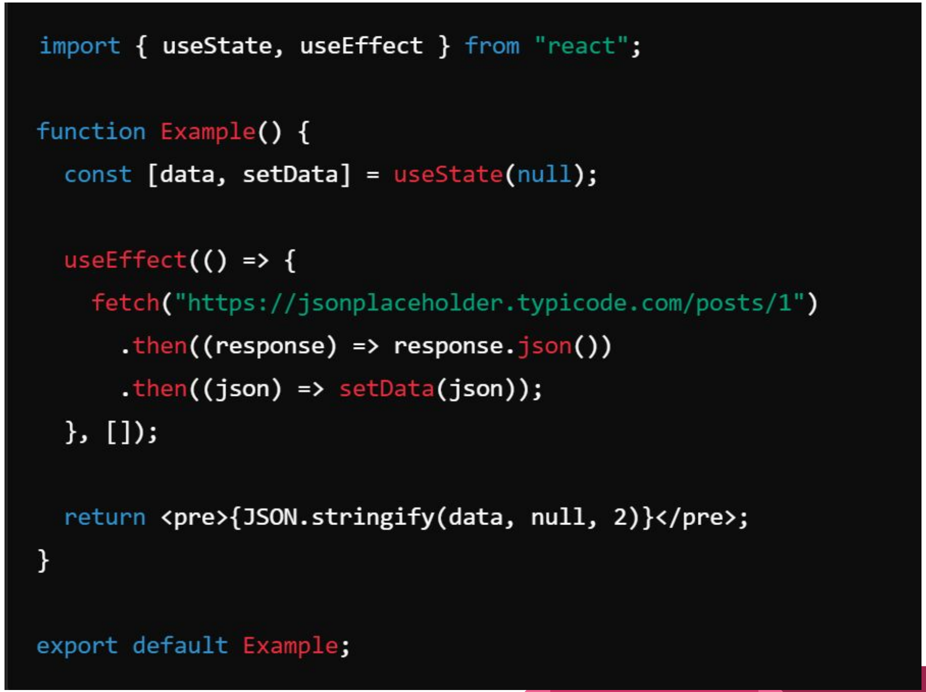


```
import { useState, useEffect } from "react";

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

  useEffect(() => {
    console.log(`Count changed: ${count}`);
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Example;
```

useEffect

Fetching Data Inside useEffect

Useful for making API calls when the component mounts.



```
import { useState, useEffect } from "react";

function Example() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/posts/1")
      .then((response) => response.json())
      .then((json) => setData(json));
  }, []);

  return <pre>{JSON.stringify(data, null, 2)}</pre>;
}

export default Example;
```

Hook - useContext

What is useContext?

=> useContext is a React Hook that lets you read and subscribe to context from your component.

When do we use useContext?

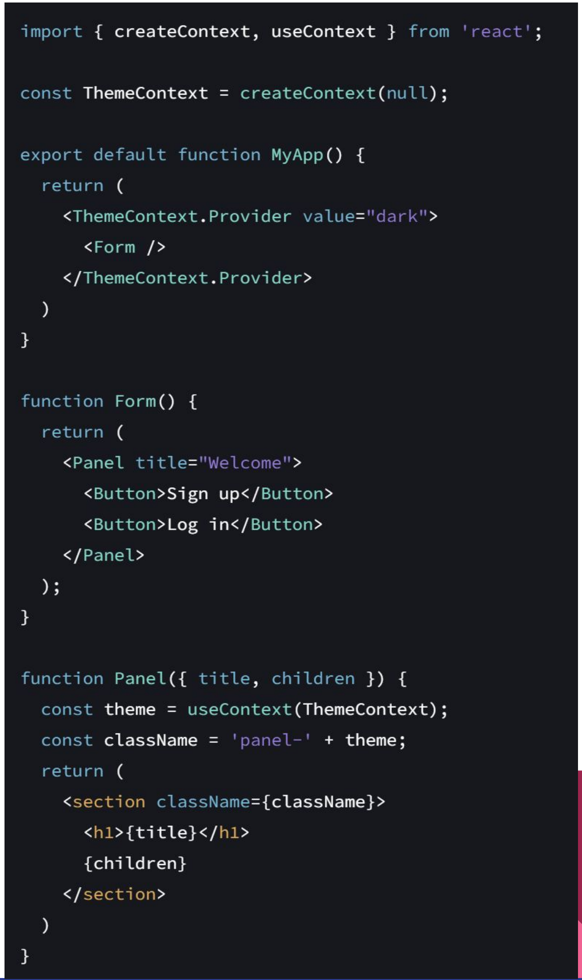
- Passing data deeply into the tree
- Updating data passed via context
- Specifying a fallback default value
- Overriding context for a part of the tree
- Optimizing re-renders when passing objects and functions

Passing data deeply into the tree

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

useContext returns the context value for the context you passed.

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

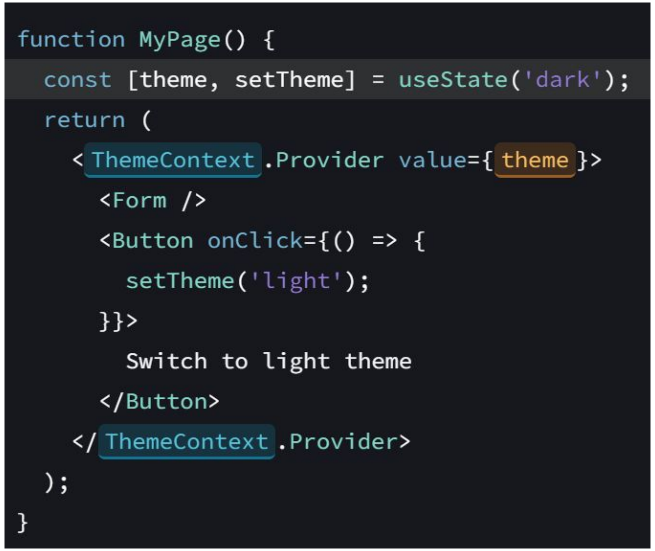


```
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  return (
    <ThemeContext.Provider value="dark">
      <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>
  )
}
```

Updating data passed via context

- To update context, combine it with state

- Declare a state variable in the parent component, and pass the current state down as the context value to the provider.

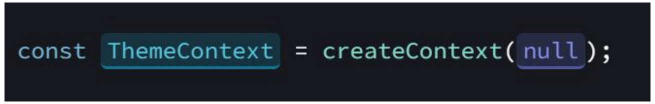


```
function MyPage() {
  const [theme, setTheme] = useState('dark');
  return (
    <ThemeContext.Provider value={theme}>
      <Form />
      <Button onClick={() => {
        setTheme('light');
      }}>
        Switch to light theme
      </Button>
    </ThemeContext.Provider>
  );
}
```

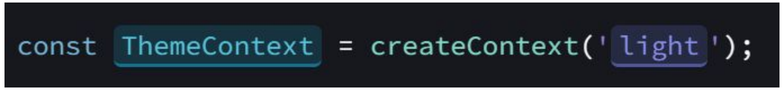
Specifying a fallback default value

If React can't find any providers of that particular context in the parent tree, the context value returned by useContext() will be equal to the default value that you specified when you created that context

This way, if you accidentally render some component without a corresponding provider, it won't break.



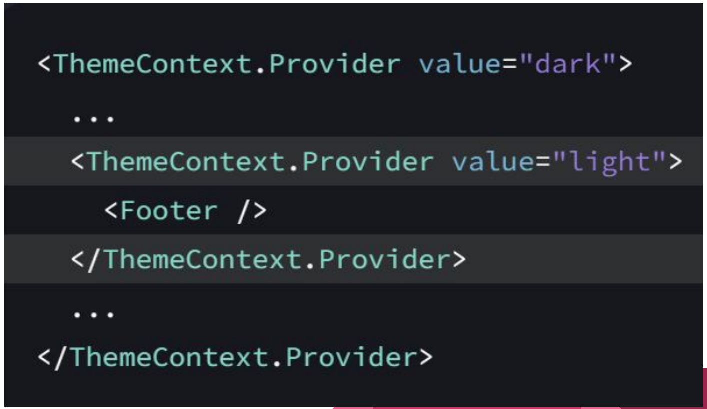
```
const ThemeContext = createContext(null);
```



```
const ThemeContext = createContext('light');
```

Overriding context for a part of the tree

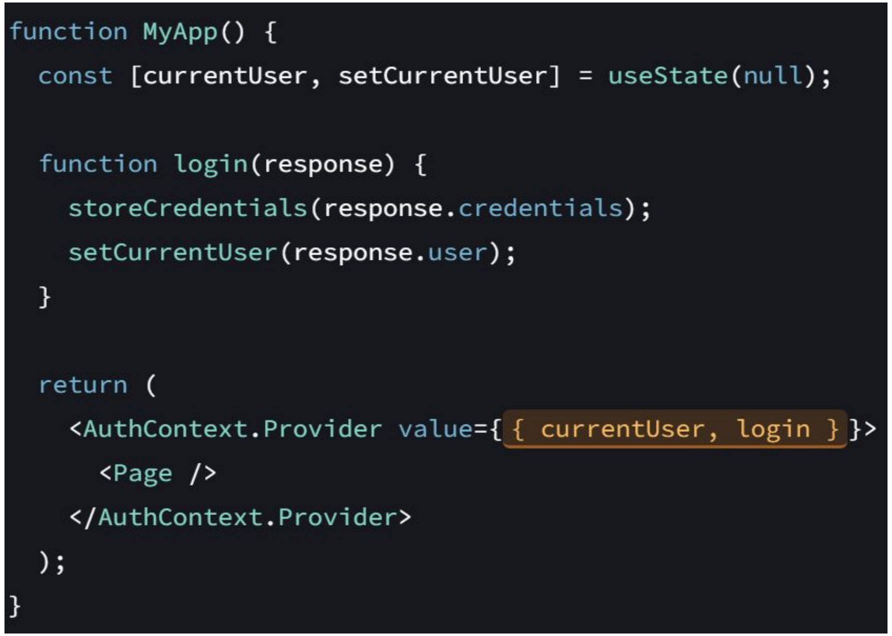
You can override the context for a part of the tree by wrapping that part in a provider with a different value.



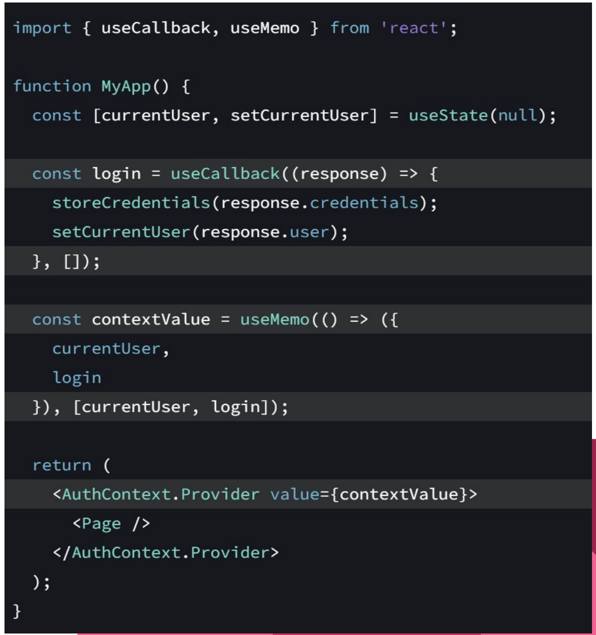
```
<ThemeContext.Provider value="dark">
  ...
  <ThemeContext.Provider value="light">
    <Footer />
  </ThemeContext.Provider>
  ...
</ThemeContext.Provider>
```

Optimizing re-renders when passing objects and functions

When MyApp need to re-render, this will be a different object pointing at a different function, so React will also have to re-render all components deep in the tree that call useContext(AuthContext)

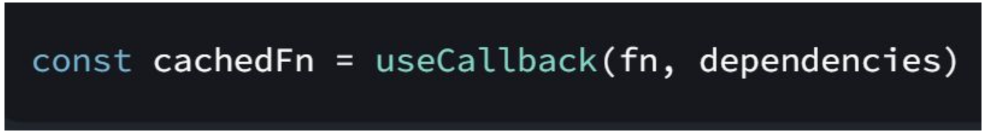


```
function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);

  function login(response) {
    storeCredentials(response.credentials);
    setCurrentUser(response.user);
  }

  return (
    <AuthContext.Provider value={{ currentUser, login }}>
      <Page />
    </AuthContext.Provider>
  );
}
```



```
import { useCallback, useMemo } from 'react';

function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);

  const login = useCallback((response) => {
    storeCredentials(response.credentials);
    setCurrentUser(response.user);
  }, []);

  const contextValue = useMemo(() => ({
    currentUser,
    login
  }), [currentUser, login]);

  return (
    <AuthContext.Provider value={contextValue}>
      <Page />
    </AuthContext.Provider>
  );
}
```

Hook - useCallback

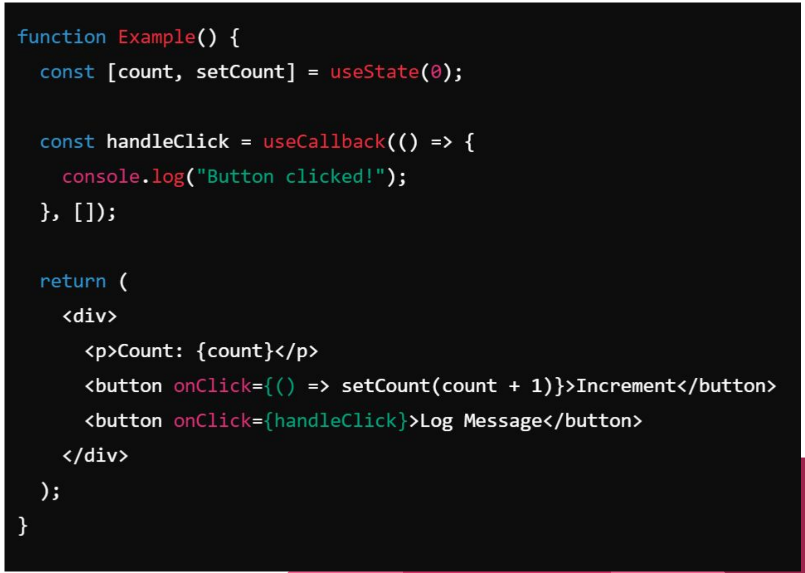


```
const cachedFn = useCallback(fn, dependencies)
```

useCallback is a React Hook that lets you cache a function definition between re-renders.

Usage

- Skipping re-rendering of components
- Updating state from a memoized callback
- Preventing an Effect from firing too often
- Optimizing a custom Hook

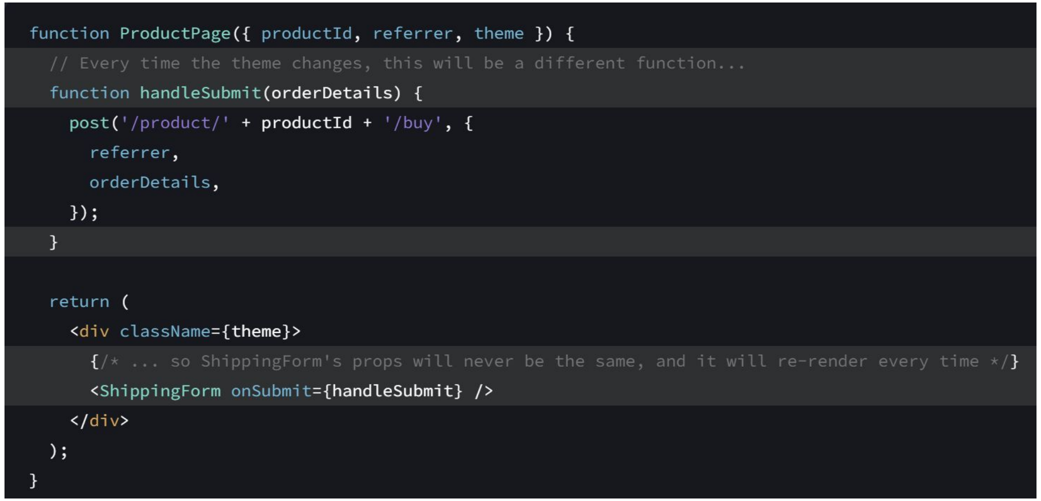


```
function Example() {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    console.log("Button clicked!");
  }, []);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={handleClick}>Log Message</button>
    </div>
  );
}
```

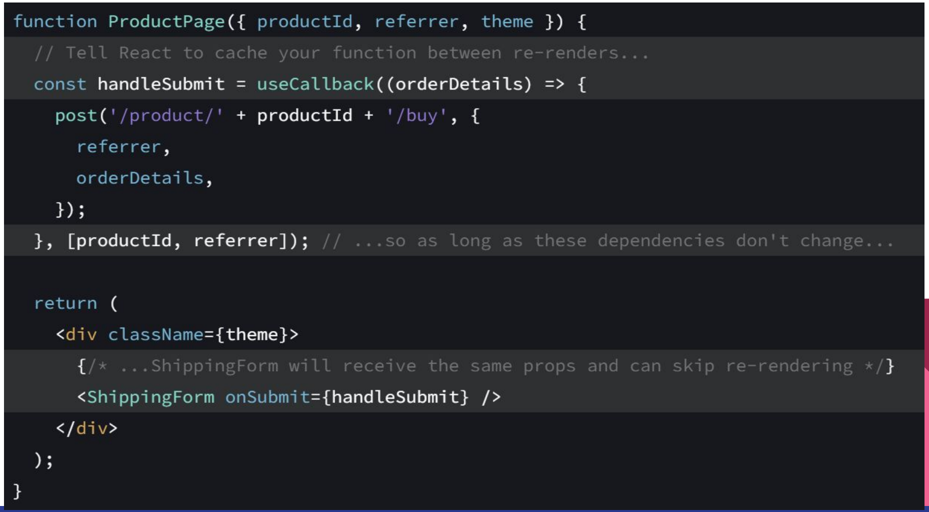
useCallback

Skipping re-rendering of components

When you optimize rendering performance, you will sometimes need to cache the functions that you pass to child components.



```
function ProductPage({ productId, referrer, theme }) {
  // Every time the theme changes, this will be a different function...
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }

  return (
    <div className={theme}>
      {/* ... so ShippingForm's props will never be the same, and it will re-render every time */}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

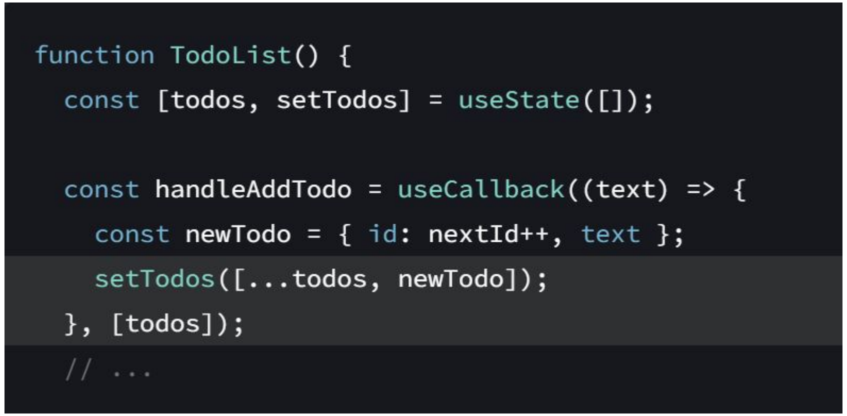


```
function ProductPage({ productId, referrer, theme }) {
  // Tell React to cache your function between re-renders...
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ...so as long as these dependencies don't change...

  return (
    <div className={theme}>
      {/* ...ShippingForm will receive the same props and can skip re-rendering */}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

useCallback

Updating state from a memoized callback

Sometimes, you might need to update state based on previous state from a memoized callback.



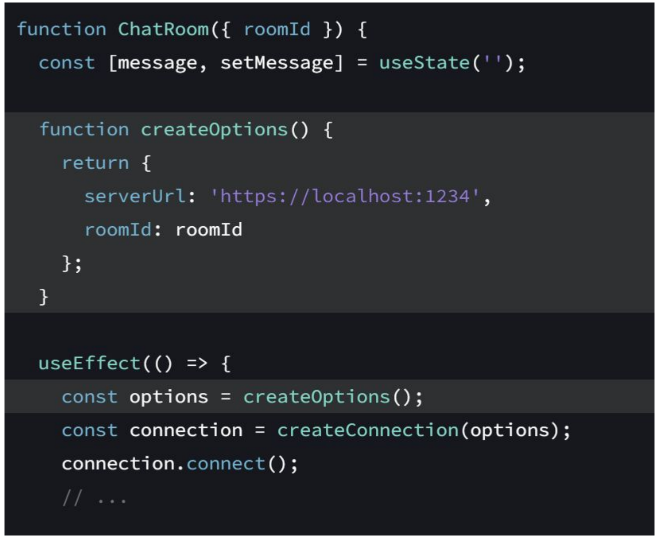
```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos([...todos, newTodo]);
  }, [todos]);
  // ...
```

useCallback

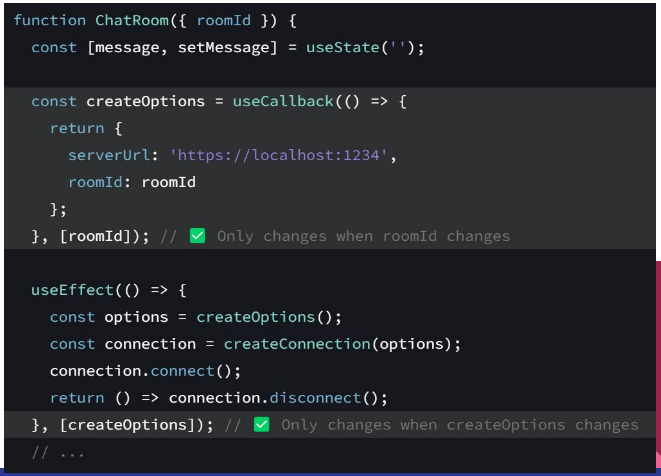
Preventing an Effect from firing too often

Sometimes, you might want to call a function from inside an Effect:

This creates a problem. Every reactive value must be declared as a dependency of your Effect. However, if you declare createOptions as a dependency, it will cause your Effect to constantly reconnect to the chat room



```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  function createOptions() {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    // ...
```

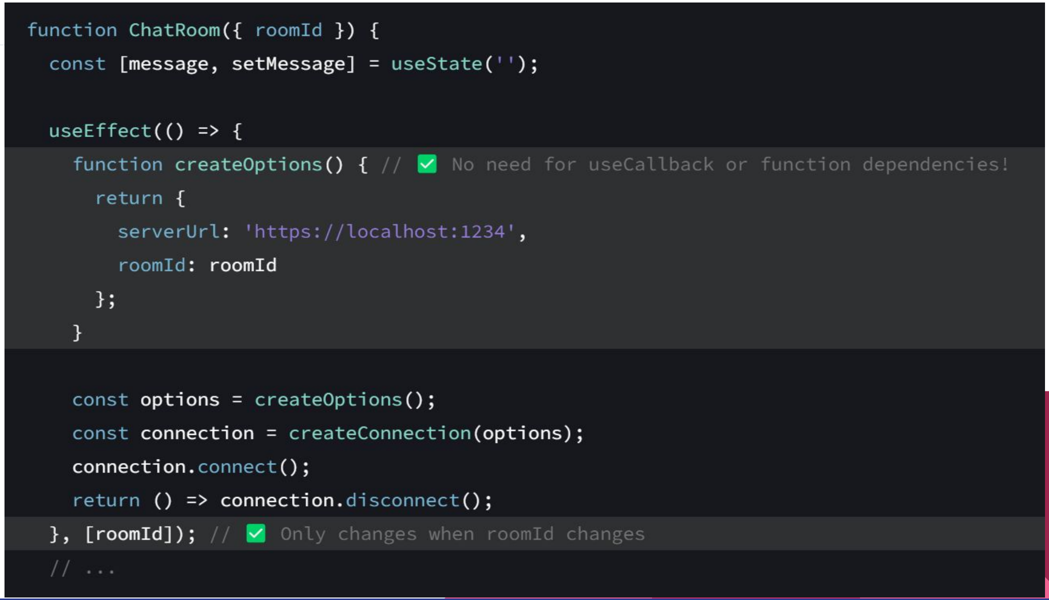


```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const createOptions = useCallback(() => {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }, [roomId]); // ✅ Only changes when roomId changes

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // ✅ Only changes when createOptions changes
  // ...
```

useCallback

It’s even better to remove the need for a function dependency. Move your function inside the Effect.

Now your code is simpler and doesn’t need useCal

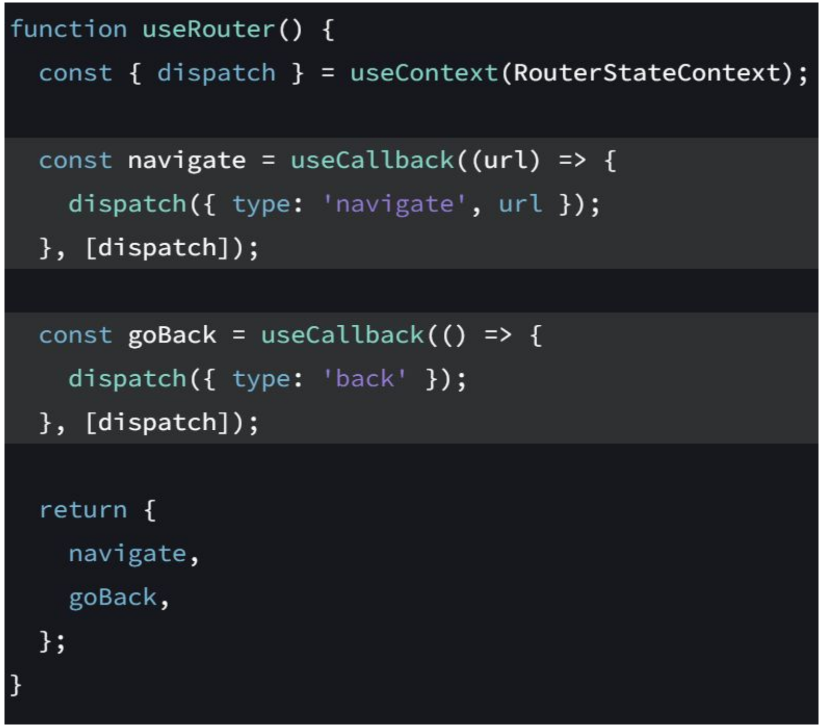


```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    function createOptions() { // ✅ No need for useCallback or function dependencies!
      return {
        serverUrl: 'https://localhost:1234',
        roomId: roomId
      };
    }

    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✅ Only changes when roomId changes
  // ...
```

useCallback

Optimizing a custom Hook

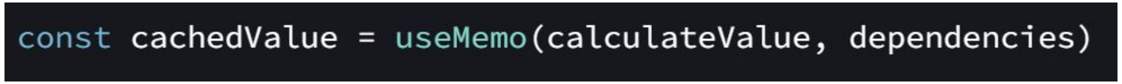
If you’re writing a custom Hook, it’s recommended to wrap any functions that it returns into useCallback



```
function useRouter() {
  const { dispatch } = useContext(RouterStateContext);

  const navigate = useCallback((url) => {
    dispatch({ type: 'navigate', url });
  }, [dispatch]);

  const goBack = useCallback(() => {
    dispatch({ type: 'back' });
  }, [dispatch]);

  return {
    navigate,
    goBack,
  };
}
```

This ensures that the consumers of your Hook can optimize their own code when needed.

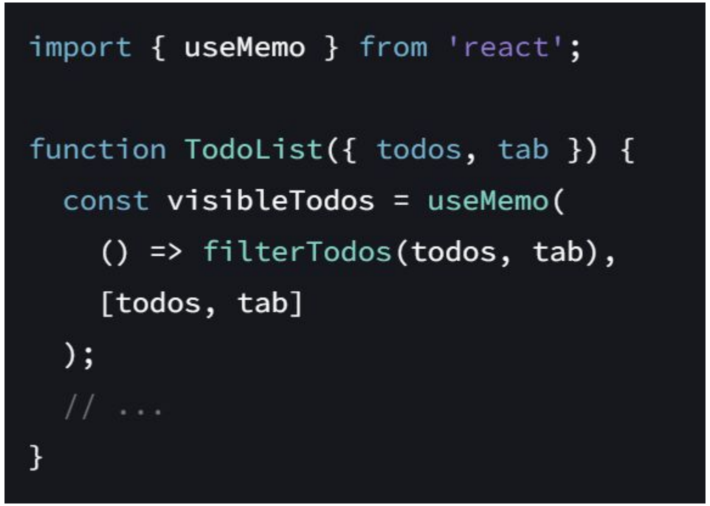
Hooks - useMemo



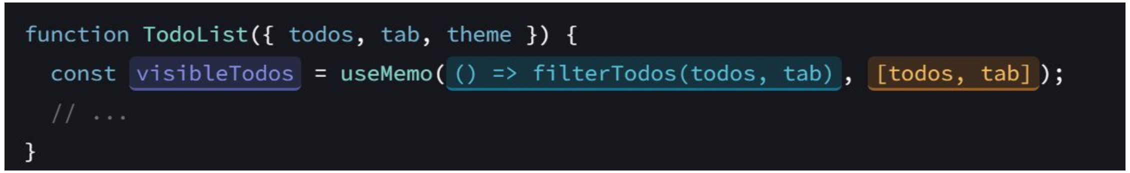
```
const cachedValue = useMemo(calculateValue, dependencies)
```

useMemo is a React Hook that lets you cache the result of a calculation between re-renders.

Call useMemo at the top level of your component to cache a calculation between re-renders:



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

function TodoList({ todos, tab }) {
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab]
  );
  // ...
}
```

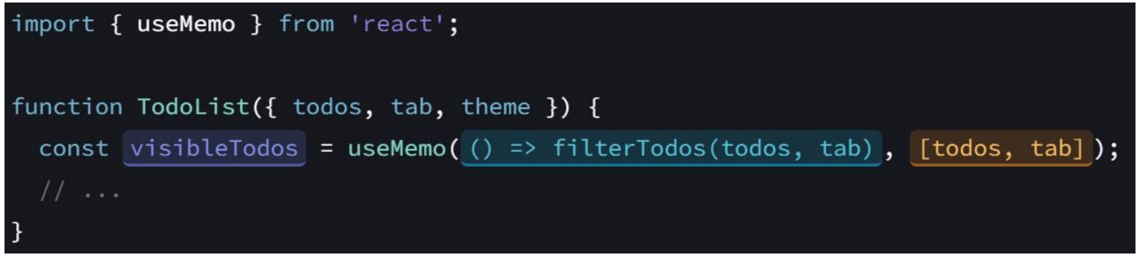


```
function TodoList({ todos, tab, theme }) {
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
}
```

useMemo

Skipping expensive recalculations

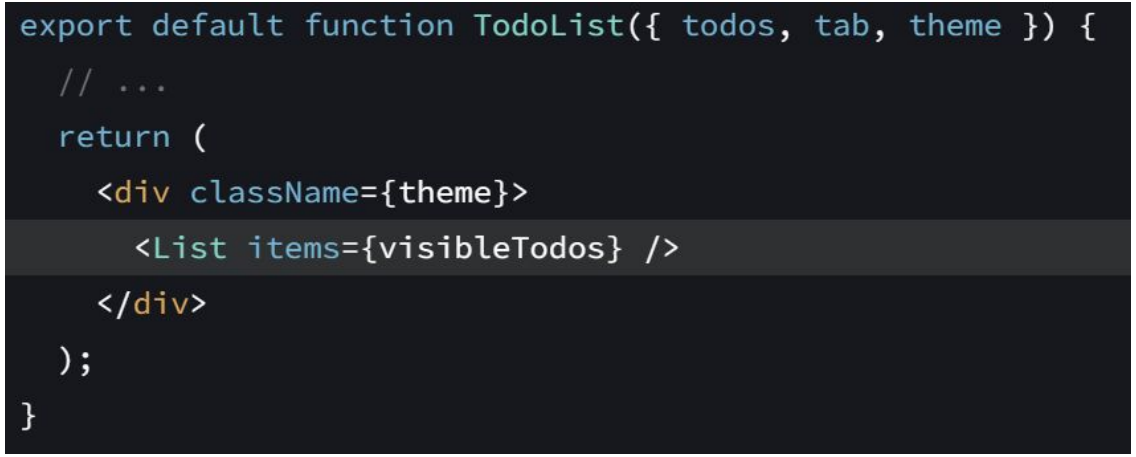
useMemo caches a calculation result between re-renders until its dependencies change



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

function TodoList({ todos, tab, theme }) {
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
}
```

useMemo

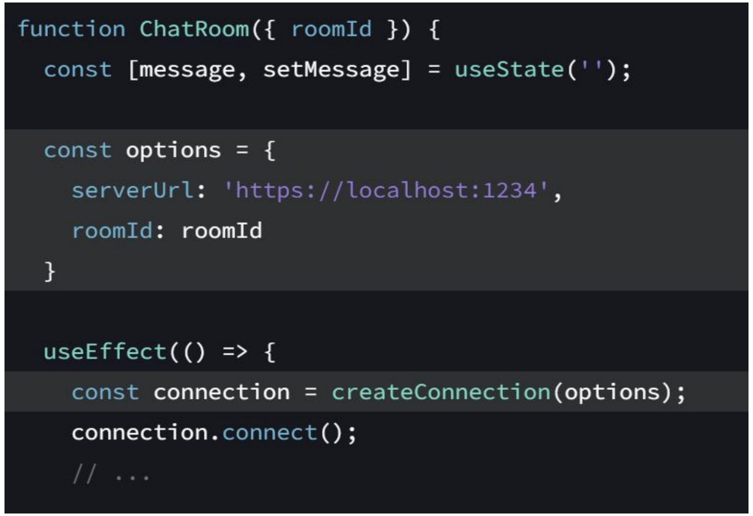
Skipping re-rendering of components



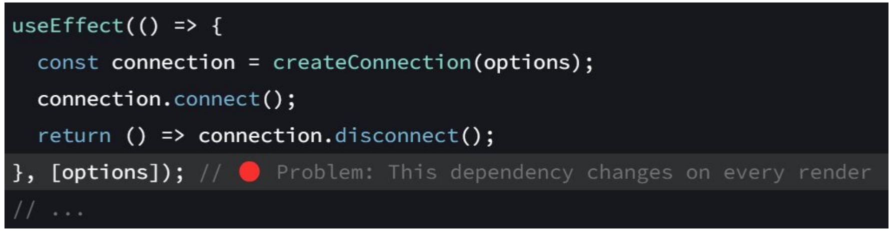
```
export default function TodoList({ todos, tab, theme }) {
  // ...
  return (
    <div className={theme}>
      <List items={visibleTodos} />
    </div>
  );
}
```

useMemo

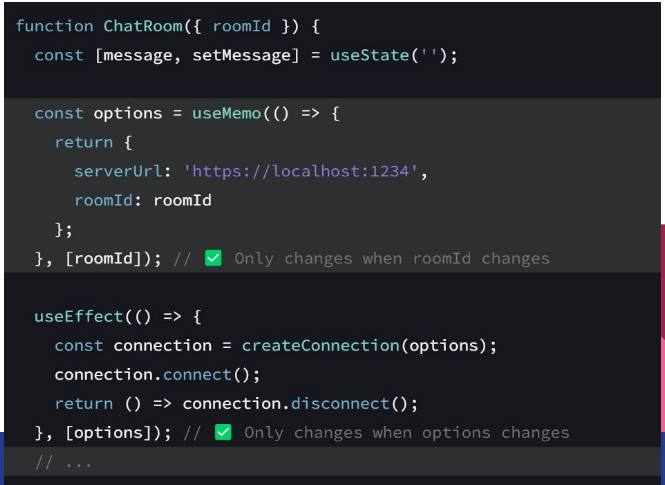
Preventing an Effect from firing too often



```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const options = {
    serverUrl: 'https://localhost:1234',
    roomId: roomId
  }

  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    // ...
```



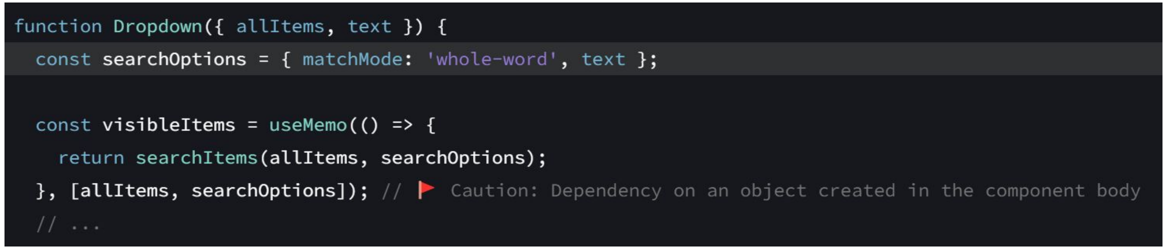
```
useEffect(() => {
  const connection = createConnection(options);
  connection.connect();
  return () => connection.disconnect();
}, [options]); // 🔴 Problem: This dependency changes on every render
// ...
```



```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const options = useMemo(() => {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }, [roomId]); // ✅ Only changes when roomId changes

  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [options]); // ✅ Only changes when options changes
  // ...
```

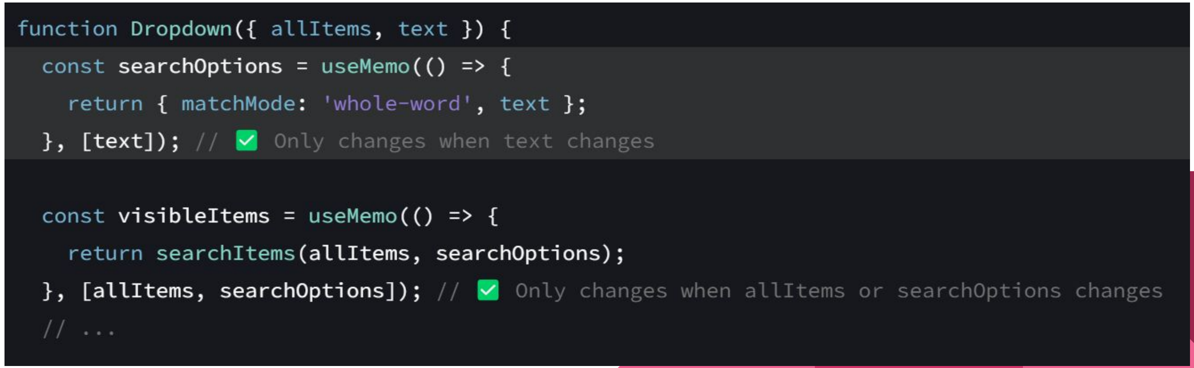
useMemo

Memoizing a dependency of another Hook



```
function Dropdown({ allItems, text }) {
  const searchOptions = { matchMode: 'whole-word', text };

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]); // 🚩 Caution: Dependency on an object created in the component body
  // ...
```

Depending on an object like this defeats the point of memoization

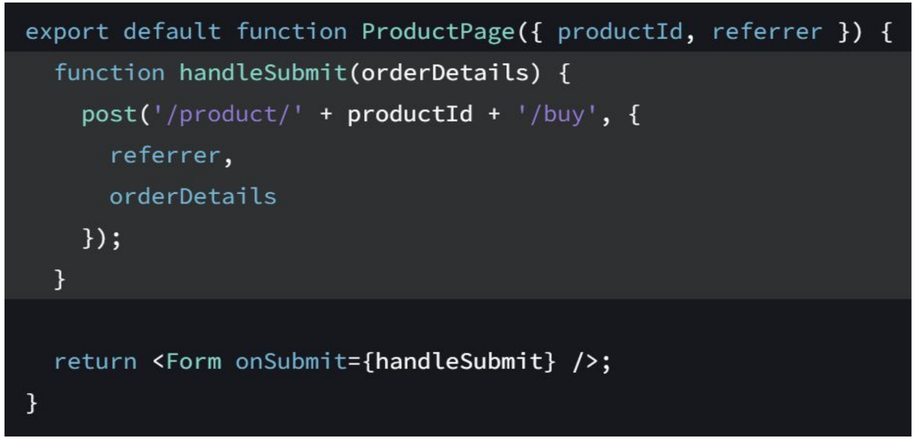


```
function Dropdown({ allItems, text }) {
  const searchOptions = useMemo(() => {
    return { matchMode: 'whole-word', text };
  }, [text]); // ✅ Only changes when text changes

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]); // ✅ Only changes when allItems or searchOptions changes
  // ...
```

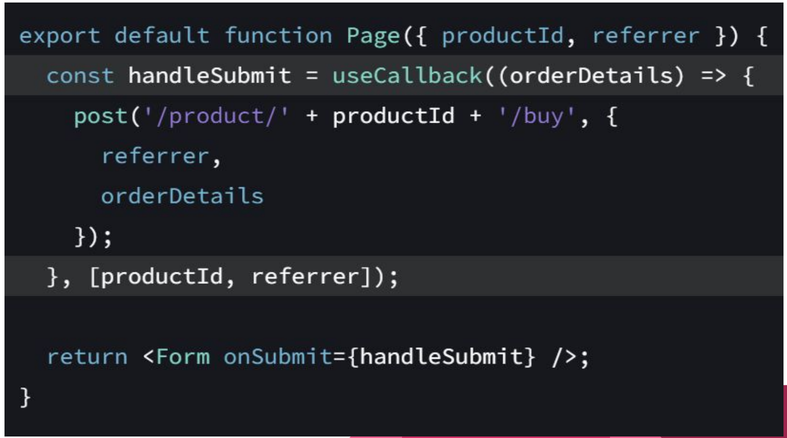
useMemo

Memoizing a function

Suppose the Form component is wrapped in memo. You want to pass a function to it as a prop:



```
export default function ProductPage({ productId, referrer }) {
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails
    });
  }

  return <Form onSubmit={handleSubmit} />;
}
```



```
export default function Page({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails
    });
  }, [productId, referrer]);

  return <Form onSubmit={handleSubmit} />;
}
```

Wrap your functions into `useCallback` instead of `useMemo` to avoid having to write an extra nested function

Handling Events in React

Conditional Rendering

Lists and Keys in React

Forms in React

React Router

API Calls with React

React Performance Optimization

- Use React.memo for memoization.
- Lazy loading with `React.lazy`.
- Avoid unnecessary re-renders.
- Optimize state and prop drilling.

Common Mistakes in React

- Mutating state directly.
- Not using keys in lists.
- Overusing state.
- Ignoring performance optimizations.

Conclusion

- React is powerful and widely used for building dynamic web applications.
- Component-based architecture promotes reusability and maintainability.
- Continuous improvements with a strong developer community.
- Start building with React today!