React Hooks

Hooks were added to React in version 16.8.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Although Hooks generally replace class components, there are no plans to remove classes from React.

What is a Hook?

Hooks allow us to "hook" into React features such as state and lifecycle methods.

Example:

Here is an example of a Hook. Don't worry if it doesn't make sense. We will go into more detail in the next section.

```
import React, { useState } from "react";
import ReactDOM from "react-dom/client";


function FavoriteColor() {
  const [color, setColor] = useState("red");


  return (
    <>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
```

```
        onClick={() => setColor("blue")}
      >Blue</button>
      <button
        type="button"
        onClick={() => setColor("red")}
      >Red</button>
      <button
        type="button"
        onClick={() => setColor("pink")}
      >Pink</button>
      <button
        type="button"
        onClick={() => setColor("green")}
      >Green</button>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

You must `import` Hooks from `react`.

Here we are using the `useState` Hook to keep track of the application state.

State generally refers to application data or properties that need to be tracked.

Hook Rules

There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

Note: Hooks will not work in React class components.

React `useState` Hook

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

State generally refers to data or properties that need to be tracking in an application.

Import `useState`

To use the `useState` Hook, we first need to `import` it into our component.

Example:

At the top of your component, `import` the `useState` Hook.

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

Notice that we are destructuring `useState` from `react` as it is a named export.

To learn more about destructuring, check out the ES6 section.

Initialize `useState`

We initialize our state by calling `useState` in our function component.

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

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

Example:

Initialize state at the top of the function component.

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


function FavoriteColor() {

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

}
```

Notice that again, we are destructuring the returned values from `useState`.

The first value, `color`, is our current state.

The second value, `setColor`, is the function that is used to update our state.

These names are variables that can be named anything you would like.

Lastly, we set the initial state to an empty string: `useState("")`

Read State

We can now include our state anywhere in our component.

Example:

Use the state variable in the rendered component.

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

```
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return <h1>My favorite color is {color}!</h1>
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

Update State

To update our state, we use our state updater function.

We should never directly update state. Ex: `color = "red"` is not allowed.

Example:

Use a button to update the state:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
```

```
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
    </>
  )
}


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

What Can State Hold

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

We could create multiple state Hooks to track individual values.

Example:

Create multiple state Hooks:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";


function Car() {
  const [brand, setBrand] = useState("Ford");
  const [model, setModel] = useState("Mustang");
  const [year, setYear] = useState("1964");
  const [color, setColor] = useState("red");
```

```
  return (
    <>
      <h1>My {brand}</h1>
      <p>
        It is a {color} {model} from {year}.
      </p>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Or, we can just use one state and include an object instead!

Example:

Create a single Hook that holds an object:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Car() {
  const [car, setCar] = useState({
    brand: "Ford",
    model: "Mustang",
    year: "1964",
```

```
    color: "red"
  });

  return (
    <>
      <h1>My {car.brand}</h1>
      <p>
        It is a {car.color} {car.model} from {car.year}.
      </p>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Since we are now tracking a single object, we need to reference that object and then the property of that object when rendering the component.
(Ex: `car.brand`)

Updating Objects and Arrays in State

When state is updated, the entire state gets overwritten.

What if we only want to update the color of our car?

If we only called `setCar({color: "blue"})`, this would remove the brand, model, and year from our state.

We can use the JavaScript spread operator to help us.

Example:

Use the JavaScript spread operator to update only the color of the car:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Car() {
  const [car, setCar] = useState({
    brand: "Ford",
    model: "Mustang",
    year: "1964",
    color: "red"
  });

  const updateColor = () => {
    setCar(previousState => {
      return { ...previousState, color: "blue" }
    });
  }

  return (
    <>
      <h1>My {car.brand}</h1>
      <p>
        It is a {car.color} {car.model} from {car.year}.
      </p>
      <button
        type="button"
```

```
        onClick={updateColor}
      >Blue</button>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Because we need the current value of state, we pass a function into our `setCar` function. This function receives the previous value.

We then return an object, spreading the `previousState` and overwriting only the color.

React `useEffect` Hooks

he `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.

`useEffect` accepts two arguments. The second argument is optional.

`useEffect(<function>, <dependency>)`

Let's use a timer as an example.

Example:

Use `setTimeout()` to count 1 second after initial render:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

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

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I've rendered {count} times!</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(<Timer />);
```

But wait!! It keeps counting even though it should only count once!

`useEffect` runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

This is not what we want. There are several ways to control when side effects run.

We should always include the second parameter which accepts an array. We can optionally pass dependencies to `useEffect` in this array.

Example

1. No dependency passed:

```
useEffect(() => {
  //Runs on every render
});
```

Example

2. An empty array:

```
useEffect(() => {
  //Runs only on the first render
}, []);
```

Example

3. Props or state values:

```
useEffect(() => {
  //Runs on the first render
  //And any time any dependency value changes
}, [prop, state]);
```

So, to fix this issue, let's only run this effect on the initial render.

Example:

Only run the effect on the initial render:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

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

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  }, []); // <- add empty brackets here

  return <h1>I've rendered {count} times!</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
```

Example:

Here is an example of a `useEffect` Hook that is dependent on a variable. If the `count` variable updates, the effect will run again:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

function Counter() {
```

```
  const [count, setCount] = useState(0);
  const [calculation, setCalculation] = useState(0);

  useEffect(() => {
    setCalculation(() => count * 2);
  }, [count]); // <- add the count variable here

  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount((c) => c + 1)}>+</button>
      <p>Calculation: {calculation}</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Counter />);
```

If there are multiple dependencies, they should be included in the `useEffect` dependency array.

Effect Cleanup

Some effects require cleanup to reduce memory leaks.

Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed.

We do this by including a return function at the end of the `useEffect` Hook.

Example:

Clean up the timer at the end of the `useEffect` Hook:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

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

  useEffect(() => {
    let timer = setTimeout(() => {
    setCount((count) => count + 1);
  }, 1000);

  return () => clearTimeout(timer)
  }, []);

  return <h1>I've rendered {count} times!</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
```

Note: To clear the timer, we had to name it.

React useContext Hook

React Context

React Context is a way to manage state globally.

It can be used together with the `useState` Hook to share state between deeply nested components more easily than with `useState` alone.

The Problem

State should be held by the highest parent component in the stack that requires access to the state.

To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.

To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

Example:

Passing "props" through nested components:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";


function Component1() {
  const [user, setUser] = useState("Jesse Hall");


  return (
    <>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
```

```
    </>
  );
}

function Component2({ user }) {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>
  );
}

function Component3({ user }) {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 user={user} />
    </>
  );
}

function Component4({ user }) {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 user={user} />
```

```
    </>
  );
}

function Component5({ user }) {
  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);
```

Even though components 2-4 did not need the state, they had to pass the state along so that it could reach component 5.

The Solution

The solution is to create context.

Create Context

To create context, you must Import `createContext` and initialize it:

```
import { useState, createContext } from "react";
import ReactDOM from "react-dom/client";
```

```
const UserContext = createContext()
```

Next we'll use the Context Provider to wrap the tree of components that need the state Context.

Context Provider

Wrap child components in the Context Provider and supply the state value.

```
function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </UserContext.Provider>
  );
}
```

Now, all components in this tree will have access to the user Context.

Use the `useContext` Hook

In order to use the Context in a child component, we need to access it using the `useContext` Hook.

First, include the `useContext` in the import statement:

```
import { useState, createContext, useContext } from "react";
```

Then you can access the user Context in all components:

```
function Component5() {
  const user = useContext(UserContext);

  return (
```

```
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

Full Example

Example:

Here is the full example using React Context:

```
import { useState, createContext, useContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext();

function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 />
    </UserContext.Provider>
  );
}
```

```
function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
    </>
  );
}

function Component3() {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 />
    </>
  );
}

function Component4() {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 />
    </>
  );
}
```

```
function Component5() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);
```

React `useRef` Hook

The `useRef` Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.

It can be used to access a DOM element directly.

Does Not Cause Re-renders

If we tried to count how many times our application renders using the `useState` Hook, we would be caught in an infinite loop since this Hook itself causes a re-render.

To avoid this, we can use the `useRef` Hook.

Example:

Use `useRef` to track application renders.

```
import { useState, useEffect, useRef } from "react";
import ReactDOM from "react-dom/client";

function App() {
  const [inputValue, setInputValue] = useState("");
  const count = useRef(0);

  useEffect(() => {
    count.current = count.current + 1;
  });

  return (
```

```
    <>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
      <h1>Render Count: {count.current}</h1>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

`useRef()` only returns one item. It returns an Object called `current`.

When we initialize `useRef` we set the initial value: `useRef(0)`.

It's like doing this: `const count = {current: 0}`. We can access the count by using `count.current`.

Run this on your computer and try typing in the input to see the application render count increase.

Accessing DOM Elements

In general, we want to let React handle all DOM manipulation.

But there are some instances where `useRef` can be used without causing issues.

In React, we can add a `ref` attribute to an element to access it directly in the DOM.

Example:

Use `useRef` to focus the input:

```
import { useRef } from "react";
import ReactDOM from "react-dom/client";

function App() {
  const inputElement = useRef();

  const focusInput = () => {
    inputElement.current.focus();
  };

  return (
    <>
      <input type="text" ref={inputElement} />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Tracking State Changes

The `useRef` Hook can also be used to keep track of previous state values.

This is because we are able to persist `useRef` values between renders.

Example:

Use `useRef` to keep track of previous state values:

```
import { useState, useEffect, useRef } from "react";
import ReactDOM from "react-dom/client";

function App() {
  const [inputValue, setInputValue] = useState("");
  const previousInputValue = useRef("");

  useEffect(() => {
    previousInputValue.current = inputValue;
  }, [inputValue]);

  return (
    <>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
      <h2>Current Value: {inputValue}</h2>
      <h2>Previous Value: {previousInputValue.current}</h2>
    </>
  );
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);
```

This time we use a combination of `useState`, `useEffect`, and `useRef` to keep track of the previous state.

In the `useEffect`, we are updating the `useRef` current value each time the `inputValue` is updated by entering text into the input field.

React `useReducer` Hook

The `useReducer` Hook is similar to the `useState` Hook.

It allows for custom state logic.

If you find yourself keeping track of multiple pieces of state that rely on complex logic, `useReducer` may be useful.

Syntax

The useReducer Hook accepts two arguments.

```
useReducer(<reducer>, <initialState>)
```

The `reducer` function contains your custom state logic and the `initialState`can be a simple value but generally will contain an object.

The `useReducer` Hook returns the current `state`and a `dispatch`method.

Here is an example of `useReducer` in a counter app:

```
import { useReducer } from "react";
import ReactDOM from "react-dom/client";


const initialTodos = [
  {
    id: 1,
    title: "Todo 1",
    complete: false,
  },
  {
    id: 2,
    title: "Todo 2",
```

```
    complete: false,
  },
];

const reducer = (state, action) => {
  switch (action.type) {
    case "COMPLETE":
      return state.map((todo) => {
        if (todo.id === action.id) {
          return { ...todo, complete: !todo.complete };
        } else {
          return todo;
        }
      });
    default:
      return state;
  }
};

function Todos() {
  const [todos, dispatch] = useReducer(reducer, initialTodos);

  const handleComplete = (todo) => {
    dispatch({ type: "COMPLETE", id: todo.id });
  };

  return (
```

```
    <>
      {todos.map((todo) => (
        <div key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleComplete(todo)}
            />
            {todo.title}
          </label>
        </div>
      ))}
    </>
  );
}


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Todos />);
```

This is just the logic to keep track of the todo complete status.

All of the logic to add, delete, and complete a todo could be contained within a single `useReducer` Hook by adding more actions.

React `useCallback` Hook

The React `useCallback` Hook returns a memoized callback function.

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

This allows us to isolate resource intensive functions so that they will not automatically run on every render.

The `useCallback` Hook only runs when one of its dependencies update.

This can improve performance.

The `useCallback` and `useMemo` Hooks are similar. The main difference is that `useMemo` returns a memoized *value* and `useCallback` returns a memoized *function*. You can learn more about useMemo in the useMemo chapter.

Problem

One reason to use `useCallback` is to prevent a component from re-rendering unless its props have changed.

In this example, you might think that the `Todos` component will not re-render unless the `todos` change:

This is a similar example to the one in the React.memo section.

Example:

index.js

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";


const App = () => {
```

```
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);

  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = () => {
    setTodos((t) => [...t, "New Todo"]);
  };

  return (
    <>
      <Todos todos={todos} addTodo={addTodo} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Todos.js

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


const Todos = ({ todos, addTodo }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
      <button onClick={addTodo}>Add Todo</button>
    </>
  );
};


export default memo(Todos);
```

Try running this and click the count increment button.

You will notice that the `Todos` component re-renders even when the `todos` do not change.

Why does this not work? We are using `memo`, so the `Todos` component should not re-render since neither the `todos` state nor the `addTodo` function are changing when the count is incremented.

This is because of something called "referential equality".

Every time a component re-renders, its functions get recreated. Because of this, the `addTodo` function has actually changed.

Solution

To fix this, we can use the `useCallback` hook to prevent the function from being recreated unless necessary.

Use the `useCallback` Hook to prevent the `Todos` component from re-rendering needlessly:

Example:

index.js

```
import { useState, useCallback } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);

  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = useCallback(() => {
    setTodos((t) => [...t, "New Todo"]);
  }, [todos]);

  return (
    <>
      <Todos todos={todos} addTodo={addTodo} />
      <hr />
      <div>
```

```
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Todos.js

```
import { memo } from "react";
const Todos = ({ todos, addTodo }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
      <button onClick={addTodo}>Add Todo</button>
    </>
  );
};
export default memo(Todos);
```

Now the `Todos` component will only re-render when the `todos` prop changes.

React `useMemo` Hook

The React `useMemo` Hook returns a memoized value.

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

The `useMemo` Hook only runs when one of its dependencies update.

This can improve performance.

The `useMemo` and `useCallback` Hooks are similar. The main difference is that `useMemo` returns a memoized value and `useCallback` returns a memoized function. You can learn more about `useCallback` in the useCallback chapter.

Performance

The `useMemo` Hook can be used to keep expensive, resource intensive functions from needlessly running.

In this example, we have an expensive function that runs on every render.

When changing the count or adding a todo, you will notice a delay in execution.

Example:

A poor performing function. The `expensiveCalculation` function runs on every render:

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

import ReactDOM from "react-dom/client";


const App = () => {

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

  const [todos, setTodos] = useState([]);

  const calculation = expensiveCalculation(count);
```

```
  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = () => {
    setTodos((t) => [...t, "New Todo"]);
  };

  return (
    <div>
      <div>
        <h2>My Todos</h2>
        {todos.map((todo, index) => {
          return <p key={index}>{todo}</p>;
        })}
        <button onClick={addTodo}>Add Todo</button>
      </div>
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
        <h2>Expensive Calculation</h2>
        {calculation}
      </div>
    </div>
  );
};
```

```
const expensiveCalculation = (num) => {
  console.log("Calculating...");
  for (let i = 0; i < 1000000000; i++) {
    num += 1;
  }
  return num;
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Use useMemo

To fix this performance issue, we can use the `useMemo` Hook to memoize the `expensiveCalculation` function. This will cause the function to only run when needed.

We can wrap the expensive function call with `useMemo`.

The `useMemo`Hook accepts a second parameter to declare dependencies. The expensive function will only run when its dependencies have changed.

In the following example, the expensive function will only run when `count` is changed and not when todo's are added.

Example:

Performance example using the `useMemo` Hook:

```
import { useState, useMemo } from "react";
import ReactDOM from "react-dom/client";
```

```
const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);
  const calculation = useMemo(() => expensiveCalculation(count),
[count]);

  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = () => {
    setTodos((t) => [...t, "New Todo"]);
  };

  return (
    <div>
      <div>
        <h2>My Todos</h2>
        {todos.map((todo, index) => {
          return <p key={index}>{todo}</p>;
        })}
        <button onClick={addTodo}>Add Todo</button>
      </div>
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
        <h2>Expensive Calculation</h2>
```

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

const expensiveCalculation = (num) => {
  console.log("Calculating...");
  for (let i = 0; i < 1000000000; i++) {
    num += 1;
  }
  return num;
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

React Custom Hooks

Hooks are reusable functions.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

Custom Hooks start with "use". Example: `useFetch`.

Build a Hook

In the following code, we are fetching data in our `Home` component and displaying it.

We will use the JSONPlaceholder service to fetch fake data. This service is great for testing applications when there is no existing data.

To learn more, check out the JavaScript Fetch API section.

Use the JSONPlaceholder service to fetch fake "todo" items and display the titles on the page:

Example:

index.js:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";


const Home = () => {
  const [data, setData] = useState(null);


  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos")
      .then((res) => res.json())
```

```
      .then((data) => setData(data));
  }, []);

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Home />);
```

The fetch logic may be needed in other components as well, so we will extract that into a custom Hook.

Move the fetch logic to a new file to be used as a custom Hook:

Example:

useFetch.js:

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

const useFetch = (url) => {
  const [data, setData] = useState(null);

  useEffect(() => {
```

```
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```

index.js:

```
import ReactDOM from "react-dom/client";
import useFetch from "./useFetch";

const Home = () => {
  const [data] =
useFetch("https://jsonplaceholder.typicode.com/todos");

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>
  );
};
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Home />);
```

Example Explained

We have created a new file called `useFetch.js` containing a function called `useFetch` which contains all of the logic needed to fetch our data.

We removed the hard-coded URL and replaced it with a `url` variable that can be passed to the custom Hook.

Lastly, we are returning our data from our Hook.

In `index.js`, we are importing our `useFetch` Hook and utilizing it like any other Hook. This is where we pass in the URL to fetch data from.

Now we can reuse this custom Hook in any component to fetch data from any URL.