React State

Prior Knowledge

- Be able to use JSX to create html elements.
- Be able to use JSX to insert JS values into html.
- Understand Array destructuring.

Learning Objectives

- Understand how to declare and update state in react.
- Understand the `useState` hook and what it returns.
- Understand the difference between setting state with a value vs a callback function.

React state

So far, all of the React App examples we have dealt with have been static and had no need to change what's rendered on the page. To make our apps dynamic they must keep track of some information that changes over time. This concept is referred to as `state`.

We can use `state` to keep track of any data in our applications that will need to change.

The useState hook

React provides us with a `hook` called `useState` that allows us to create a `state variable`. This variable can then be updated over time, and React will update our UI and the rendered html, whenever that state value is updated.

`useState`:

- takes 1 argument - the initial value for the state
- returns an array with 2 elements.
  - 0 - The current state value
  - 1 - A function to update the state value.

A common pattern, and the one used by the React docs on hooks, is to use array destructuring to declare the state value and setState function.

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

const App = () => {
  const [name, setName] = useState('Paul');

  return (
    <div>
      <h1>Hello {name}</h1>
    </div>
  );
};
```

In the example above name is set to it's initial value of 'Paul' and can be used just as any other variable would.

We can update this state value using the `setState` function returned to us from `useState`. When a new value is set, React will re-render the component and update the relevant parts of our html with the new state values.

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

const App = () => {
  const [name, setName] = useState('Paul');

  console.log(name, '<< current name in state');

  return (
    <div>
      <h1>Hello {name}</h1>
      <button onClick={() => setName('Izzi')}>Say hi to Izzi</button>
      <button onClick={() => setName('Ant')}>Say hi to Ant</button>
    </div>
  );
};
```

An aside about onClick

`onClick` takes a **function**.

If we pass onClick `{event} => setName('Izzi')`, this function will be called with a synthetic `event` when triggered that is created by React (but for all intended purposes, the same as the 'real' event).

We aren't really interested in this event (we could inspect it to try to find out the `event.target`, but the data we need is already available to us in React!).

If we were to pass `setName('Izzi')` to the `onClick`, the `onClick` is no longer being passed a **function**, but a **function invocation**. So as soon as this component renders for the first time, this function will be called.

Passing the anonymous arrow function allows us to pass arguments to `setName` so that we can change the name in state.

setState updater functions

The `setState` functions can be invoked in 2 ways.

1. Pass the new value of the state directly
2. Pass a function that will return the new state. This function will be invoked with the current state value.

Multiple calls to `setState` can be batched together to improve performance. When updating the state with a completely new value, such as our `name` example we can pass the new value directly.

Sometimes, however, the new state value will be dependant on the current value. Consider the counter below.

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

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

  return (
    <div>
      <h1>Count: {count}</h1>
      <button>Increase Count</button>
    </div>
  );
};
```

When clicking the button, the count should increase by one. In this case `setCount` is invoked with a function, that takes the current count as a parameter.

This function can use the current count to work out what the new state should be and return the new state value.

This guarantees the value of `currCount` is correct and if this button was pressed several times very quickly then our state will always be worked out correctly.

```
<button
  onClick={() =>
    setCount((currCount) => {
      return currCount + 1;
    })
  }
>
```

Writing complicated state updates can complicate our JSX so non-trivial updates can always be extracted out into helper functions.

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

  const incrementCount = (increment) => {
    setCount((currCount) => {
      return currCount + increment;
    });
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => incrementCount(1)}>Increase Count</button>
      <button onClick={() => incrementCount(-1)}>Decrease Count</button>
    </div>
  );
};
```

Never mutate state

React state is designed to be replaced, not mutated. React will make comparisons between state values on re-renders internally and will replace old mutated state with the new values. As per the React docs, we should never mutate the state values and use the `setState` functions to replace old values. Keep in mind how references work and be sure that you are not mutating the current state at any point.

Consider the example below that holds an array in state.

```
const App = () => {
  const [todos, setTodos] = useState(['code on day 0', 'code on day 1']);

  const addTodo = () => {
    setTodos((currTodos) => {
      // correct - returned a new array and used currTodos to create the new array
      return [...currTodos, `code on day ${currTodos.length}`];
    });
  };

  return (
    <div>
      <button onClick={() => addTodo()}>Add Todo</button>
      <ul>
        {todos.map((todo) => {
          return <li key={todo}>{todo}</li>;
        })}
      </ul>
    </div>
  );
};
```

It's easy to forget the currTodos is a reference to the current state and the following method could lead to un-intended and hard to fix bugs

```
const addTodo = () => {
  setTodos((currTodos) => {
    // bad - mutated the currTodos by pushing to it
    currTodos.push(`code on day ${currTodos.length}`);
    return currTodos;
  });
};
```

See Updating Objects in State and Updating Arrays in State for more info on updating a non-primitive state value.

React Hooks

We have used a React hook to manage our state. These were introduced as a way to "hook" into React's lifecycle and access key features of React, such as state from within functional components. You can draw a comparison from our use of Jest hooks like `beforeEach()` and `afterAll()` and how they inject behaviour at key points of a testing suite's lifecycle.

These were introduced in React v16.8 and weren't available before then.

To use state before this version you would have to use a class based component instead, which are covered in their own section of these notes. The underlying principles are the same, but hooks provide a much cleaner syntax.

Naming Conventions

All hooks start with the word `use`. This is what denotes a React hook, meaning that your project will need to be on React version 16.8 or higher in order to use it.

React comes with some inbuilt hooks, such as `useState` and we can write our own custom hooks as well, which will be covered later.

When defining the values returned from `useState`, the function used to update that state starts with the word `set`. This comes from the class based version, generically called `setState`. Our functions should be named with the value of state they update. i.e. `setName`, `setCount` etc.

Rules of Hooks

Another reason for the `use` naming convention is that there are certain restrictions on how hooks can be used. These are summarised in the docs here. The docs give great examples of how these rules work, but to summarise

1. Only Call Hooks at the Top Level

In order for React to use hooks correctly, the same number of hooks must be called in the same order on each re-render. For this reason, hooks cannot be called from within `if` statements, `for` loops or nested functions.

2. Only Call Hooks From React Functions

Hooks can be called from within React components or inside custom hooks. This is for the same reason as above and makes sure React knows about all the hooks you are using.

In practice this means that we use all of our hooks at the top of our components and perform the component's logic below that. This is a pattern that will become second nature and you will likely not have to worry too much about these rules unless you deviate from that pattern.

Passing state to child components

We can pass any value from a parent component to a child component via props. This allows us to extract functionality into separate components. Consider the example below. `App` keeps some todos in state. The logic for rendering a list of todos is contained within the `TodoList` component, a child of the `App`. In order for the TodoList component to do its job, it needs access to the list of todos.

We can pass the todos from `App`'s state as a prop named `todos`.

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

const App = () => {
  const [todos, setTodos] = useState(['code on day 0', 'code on day 1']);

  const clearTodos = () => {
    setTodos([]);
  };

  return (
    <div>
      <Header />
      <button onClick={() => clearTodos()}>Clear All todos</button>
      <TodoList todos={todos} />
    </div>
  );
};

const Header = () => {
  return <h1>My Todo List</h1>;
};

const TodoList = (props) => {
  return (
    <ul>
      {props.todos.map((todo) => {
        return <li key={todo}>{todo}</li>;
      })}
    </ul>
  );
};
```

When `setTodos` is called, `App` (the component responsible for holding the state) is re-rendered as well as `TodoList`. This way any changes to the parent state are propagated down to the child components. Bear in mind that the rules of state still apply and that we should **never mutate state directly**. `props.todos` is a reference to the parents state and should not be mutated.

Child components can update their parents state through the use of the `setState` functions just as their parent components would. Functions are **first class citizens** in JS and can be passed as any other value would.

In the above example, if the button to clear the todos was in a child component called `Controls`. This component will need access to the `setTodos` function, which can be passed as a prop.

```
const App = () => {
  const [todos, setTodos] = useState(['code on day 0', 'code on day 1']);

  return (
    <div>
      <Header />
      <Controls setTodos={setTodos} />
      <TodoList todos={todos} />
    </div>
  );
};

const Controls = (props) => {
  const clearTodos = () => {
    props.setTodos([]);
  };

  return <button onClick={() => clearTodos()}>Clear All todos</button>;
};

// other components omitted
```

copy original markdown to clipboard