React State

(https://github.com/learn-co-curriculum/react-hooks-state) (https://github.com/learn-co-curriculum/react-hooks-state/issues/new)

Learning Goals

- Understand what state is
- Explain the difference between state and props
- Initialize and update state in a React component
- Understand how updating state causes components to re-render

Introduction

In this lesson, we'll dive into component **state**, and see how we can make our components respond to change dynamically by working with the React state system.

Working with state is one of the most crucial elements of creating dynamic React applications, so make sure to pay extra attention to this lesson! If you'd like to explore state in more depth, the **new React docs** (https://beta.reactjs.org/learn/managing-state) have a great section on learning about state with example code to explore.

You can fork and clone this lesson's repository if you'd like to see a working version of a component that uses state and explore the code.

What's state?

Let's quickly talk about what **state** is in React. State is data that is **dynamic** in your component: it changes over time as users interact with your application.

A component's state, unlike a component's props, *can* change during the component's life.

Consider the limitations of props: for a component's props to change, its **parent** component needs to send it new props (after which, the component would 're-render' itself with the new props).

State provides us with a way to maintain and update information *within* a component *without* requiring its parent to somehow send updated information.

Imagine that we have a single component which displays an integer. When a user clicks the component, it should increment its integer by 1. If we were to represent this integer value in the component using **state**, the component could increment its own state and automatically re-render whenever state is updated!

useState

All of the state for your application is held in React's internal code. Whenever you have a component that needs to work with state, you must first tell React to create some state for the component.

To do this, we must first import a function from React called `useState`. This special function is a **React Hook** that will let us "hook into" React's internal state inside of our function component.

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

Initializing State

Once we've imported the `useState` hook, we can call it inside of our component, like so:

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

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

  return <button>{count}</button>;
}
```

When we call `useState(0)` inside the function component, we're telling React to create some new internal state for our component with an **initial value** of 0 (or whatever value we pass to `useState` when we call it).

`useState` will return an **array** that has two variables inside of it:

- `count` : a reference to the current value of that state in React's internals
- `setCount` : a *setter* function so we can update that state

We *could* create the variables and set their values by accessing the elements from the array individually, like this:

```
const countState = useState(0);
// => [0, setStateFunction]
const count = countState[0];
const setCount = countState[1];
```

But to clean up the code, React recommends using **array destructuring (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)** to achieve the same result in one line of code instead of three:

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

We can then use the `count` variable to display its current value in the `<button>` element:

```
<button>{count}</button>
// => <button>0</button>
```

Setting State

The setter function we get back from calling `useState` is straightforward in its purpose: it sets/updates state! That's it! That's what it's there for. Whenever we want to update state, we can just call the setter function (in our case, `setCount`):

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

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

  return <button onClick={increment}>{count}</button>;
}
```

Now, when the `<button>` element is clicked, it will run our `increment` function. `increment` calls the `setCount` function to do these two things:

- Update the value of `count` in React's internal state to be `count + 1`
- Re-render our component

The magic of working with **state** is that we don't have to worry about any complex DOM manipulation (like finding the button element and telling it to display the new `count` value) — whenever we call the `setCount` function, React will automatically **re-render** our component, along with any of its child components, and update the DOM based on the new values for state!

You can visualize the steps that happen when we call `setCount` like this:

1. Calling `setCount(1)` tells React that its internal state for our `Counter` component's `count` value must update to 1
2. React updates its internal state
3. React **re-renders** the `Counter` component
4. When the `Counter` component is re-rendered, `useState` will return the current value of React's internal state, which is now 1
5. The value of `count` is now 1 within our `Counter` component
6. Our component's JSX uses this new value to display the number 1 within the button

Using state like this allows React to be very performant: based on which component is updated, React can determine which child components are affected and how the DOM needs to be changed when these components are re-rendered.

Take your time to read through the above code. Work through it line by line and make sure you are comfortable before moving forward.

Setting State is Asynchronous

While using the `setCount` function is straightforward enough, there is one very important caveat about *how* it functions that we need to explore: it sets state **asynchronously**.

In order to understand why this is important, let's look at an example of a state setter function being used in a component. The following gif is of this component (pay close attention to the `console.log()` s:

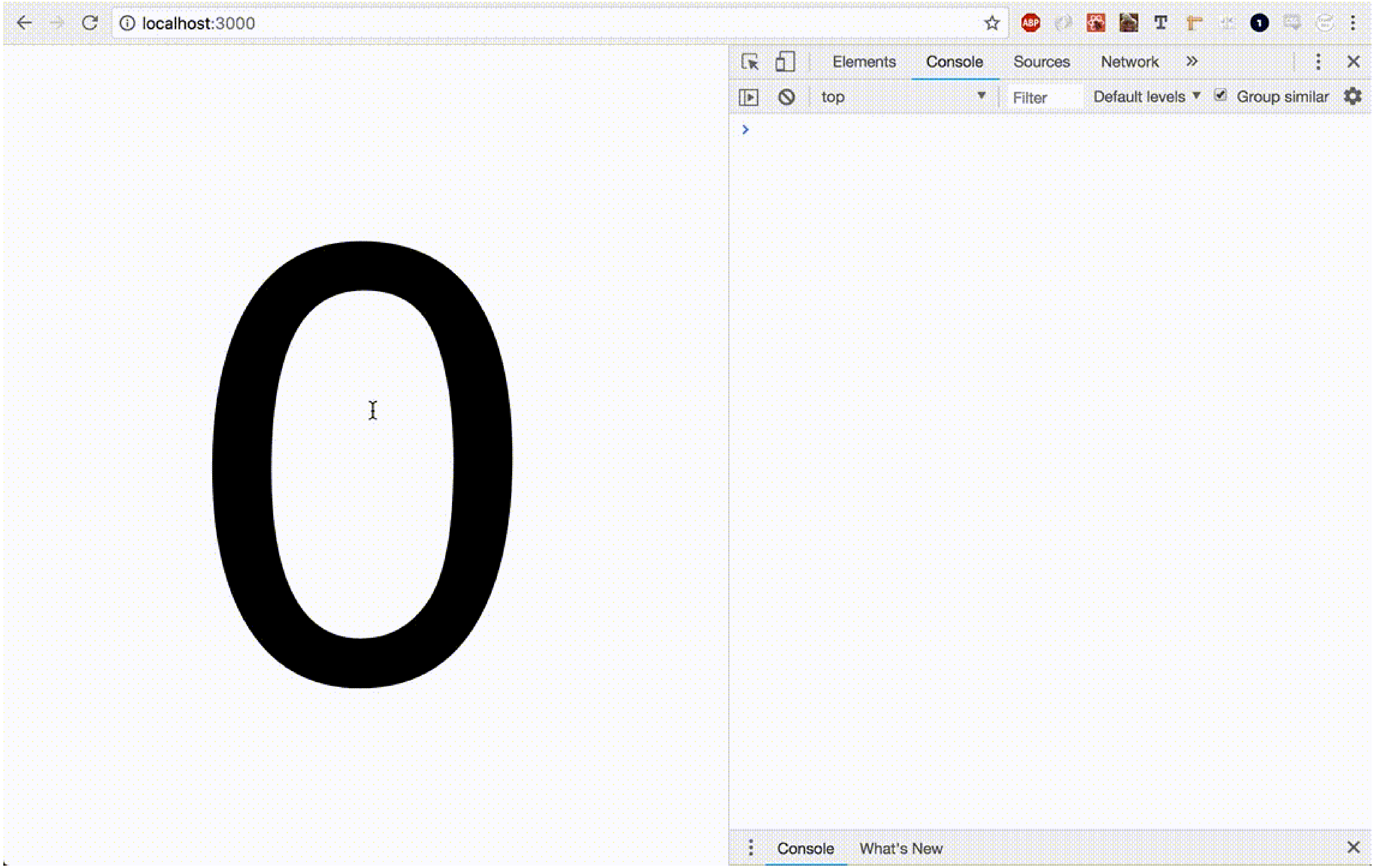
```
function Counter() {
  const [count, setCount] = useState(0);

  function increment() {
    console.log(`before setState: ${count}`);

    setCount(count + 1);

    console.log(`after setState: ${count}`);
  }

  return <div onClick={increment}>{count}</div>;
}
```



What we are seeing is `setCount()` functioning **asynchronously**. When we execute `setCount()` , it is *non-blocking*. It fires off a message to React's internals saying: "Hey, you need to update state to this value when you have a chance." The component finishes doing its current task *before* updating the state. In this case, it finishes executing the `increment` function in full before updating the state.

Since the new value of `count` is only available after React has re-rendered the component, we don't see immediate changes to that variable after `setCount()` has been called.

It's not uncommon for new React developers to get 'bitten' by the asynchronous nature of state setter functions at least once. For this reason, React recommends using a slightly different syntax for setting state when working with values that are calculated based on the previous version of state (like our counter). To demonstrate the issue, consider the following:

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

  function increment() {
    // call setCount twice, to update the counter by two every time we click
    setCount(count + 1);
    setCount(count + 1);
  }

  return <div onClick={increment}>{count}</div>;
}
```

This is a contrived example — we could just as easily have called `setCount(count + 2)` instead of calling `setCount` twice. But if you run this example in your browser, you may be surprised at the result. Instead of seeing the counter incremented by 2, it's still only incremented by 1!

Let's add in some `console.log` s to see what's going on:

```
function increment() {
  console.log(count);
  // => 0

  setCount(count + 1);
  // equivalent to setCount(0 + 1)
```

```
    console.log(count);
    // => 0

    setCount(count + 1);
    // equivalent to setCount(0 + 1)

    console.log(count);
    // => 0
  }
```

Because `setState` is asynchronous — i.e., because the value of `count` isn't updated immediately — when `setCount` is called the second time, `count` is still equal to 0! As a result, we are effectively calling `setCount(1)` in *both* cases.

React actually provides a built in solution for this problem. Rather than passing a new value into `setCount`, we can instead pass a callback function. That function, when called inside `setCount`, will be passed the state variable containing the current state of `count`. This is typically referred to as the *previous state* since it's the state before the current call to `setCount` is executed. With this knowledge, we can rewrite the `increment` function to:

```
function increment() {
  setCount((currentCount) => currentCount + 1);
  setCount((currentCount) => currentCount + 1);
}
```

When using the callback version of `setCount`, React will pass in the current value of `count` before updating it. Now our code works as intended and updates `count` by 2 when the button is clicked.

As a rule of thumb, **any time you need to set state based on the current value of state, you should use the callback syntax**.

Rules of Hooks

Since the `useState` hook is the first of several React Hooks we'll be learning about, now's a good time to review some general **rules for working with hooks (https://reactjs.org/docs/hooks-rules.html)**:

Only Call Hooks at the Top Level

> Don’t call Hooks inside loops, conditions, or nested functions.

When you're using a React Hook such as `useState`, it's important that the hook is called every time your component is rendered. That means this syntax isn't valid:

```
function Counter(props) {
  if (props.shouldHaveCount) {
    // This is wrong -- never call a hook inside a condition
    const [count, setCount] = useState(0);

    // return ...
  }
  // return ...
}
```

The reason for this comes down to how React keeps track of which piece of state each variable is associated with — hooks must always be called in the same order. For a more detailed explanation, check out the **React docs** (https://reactjs.org/docs/hooks-rules.html#explanation).

Only Call Hooks from React Functions

> Don’t call Hooks from regular JavaScript functions.

React Hooks are meant to work specifically with React components, so make sure to only use Hooks inside of React components. We'll see how to create custom hooks later on — custom hooks and React components are the only two places you can use React hooks.

A Word of Caution

While component state is a very powerful feature, it should be used as sparingly as possible. State adds complexity (sometimes unnecessarily) and can be very easy to lose track of. The more state we introduce in our application, the harder it will be to keep track of all of the changes in our data. Remember: **state is only for values that are expected to change during the component's life**.

Before creating any state variables, always think through whether you can derive the information you need from a component's props or from existing state variables. If the answer is yes, do not create additional state!

Conclusion

Whenever we need *dynamic* data in our applications (values that change over time), we should use **state**. We create our initial state by calling the `useState` hook inside of our components.

To update state, we must use the `setState` function returned by `useState`, so that changes to state cause our components to re-render.

Also, because setting state is *asynchronous*, if you need to make any updates to state that are based on the current value of state, you should use the callback syntax.

Resources

- **React Docs (beta): Managing State (https://beta.reactjs.org/learn/managing-state)**
- **React Docs: The useState hook (https://reactjs.org/docs/hooks-state.html)**
- **Props vs. State (https://github.com/uberVU/react-guide/blob/master/props-vs-state.md)**
- **React Docs: Thinking in React (https://reactjs.org/docs/thinking-in-react.html#step-3-identify-the-minimal-but-complete-representation-of-ui-state)**
- **React Docs: Rules of Hooks (https://reactjs.org/docs/hooks-rules.html)**