React 101
Codecademy
Hooks

Function Components

Stateless Function Components

In React we can also define components as JavaScript functions – we call then function components to differentiate them from class components.

In the latest version of React, function components can do everything that class components can do. In most cases, function components offer a more elegant, concise way of creating React components.

```
// A component class written in the usual way:
export class MyComponentClass extends React.Component {
  render() {
    return <h1>Hello world</h1>;
  }
}

// The same component class, written as a stateless functional component:
export const MyComponentClass = () => {
  return <h1>Hello world</h1>;
}

// Works the same either way:
ReactDOM.render(
      <MyComponentClass />,
      document.getElementById('app')
);
```

Function Components and Props

Like any component, function components can receive information via props.

To access these props, give your function component a parameter named props. Within the function body, you can access the props using this pattern: props.propertyName. You don’t need to use the this keyword.

```
export function YesNoQuestion (props) {
  return (
    <div>
      <p>{props.prompt}</p>
      <input value="Yes" />
      <input value="No" />
    </div>
  );
}

ReactDOM.render(
  <YesNoQuestion prompt="Have you eaten an apple today?" />,
  document.getElementById('app');
);
```

The State Hook

Why Use Hooks?

Classes are:

- Difficult to reuse between components
- Tricky and time consuming to test
- Have confused many developers and caused lots of bugs

React 16.8+ supports Hooks. With Hooks, we can use simple function components to do lots of the fancy things that we could only do with class components in the past.

React Hooks, plainly put, are functions that let us manage the internal state of components and handle post-rendering side effects directly from our function components. Hooks don’t work inside classes – they let us use fancy React features without classes.

Function components and React Hooks do not replace class components. They are just a tool we can take advantage of.

Update Function Component State

State Hook is the most common Hook used for building React components. The State Hook is a named export from the React library, so we import it like this:

```
import React, { useState } from ‘react’;
```

`useState()` is a JS function defined in React Library. When we call this function it returns an array with two values:

- current state – the current value of this state
- state setter – a function that we can use to update the value of this state

React returns these two values in an array, so we can assign them to local variables, naming them whatever we like.

```
const [toggle, setToggle] = useState();
```

To rerender a component with a new value, all we need to do is call the `setToggle()` function with the next state value as an argument.

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

function Toggle() {
  const [toggle, setToggle] = useState();

  return (
    <div>
      <p>The toggle is {toggle}</p>
      <button onClick={() => setToggle("On")}>On</button>
      <button onClick={() => setToggle("Off")}>Off</button>
    </div>
  );
}
```

We need not worry about binding functions to class instances, working with constructors, or dealing with the this keyword. With the State Hook, updating state is as simple as calling a state setter function.

Calling a state setter signals to React that the component needs to re-render, so the whole function defining the component is called again. The magic of `useState()` is that it allows React to keep track of the current value of state from one render to the next!

Initialize State

Just like we used the State Hook to manage a variable with string values, we can use the State Hook to manage the value of any primitive data type and even data collections like arrays and objects!

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

function ToggleLoading() {
  const [isLoading, setIsLoading] = useState();
  return (
    <div>
      <p>The data is {isLoading ? 'Loading' : 'Not Loading'}</p>
      <button onClick={() => setIsLoading(true)}>
        Turn Loading On
      </button>
      <button onClick={() => setIsLoading(false)}>
        Turn Loading Off
      </button>
    </div>
  );
}
```

To initialize our state with any value we want, we simply pass the initial value as an argument to the `useState()` function call.

```
const[isLoading, setIsLoading] = useState(true);
```

There are three ways in which this code affects our component:

1. During the first render, the initial state argument is used.
2. When the state setter is called, React ignores the initial state argument and uses the new value.
3. When the component re-renders for any other reason, React continues to use the same value from the previous render.

If we don't pass an initial value when calling useState(), then the current value of the state during the first render will be undefined. Often, this is perfectly fine for the machines, but can be unclear to the humans reading the code. So, we prefer to explicitly initialize our state. If we don't have the value needed during the first render, we can explicitly pass `null` instead of just passively leaving the value as `undefined`.

Use State Setter Outside of JSX

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

export default function EmailTextInput() {
  const [email, setEmail] = useState('');
  const handleChange = (event) => {
    const updatedEmail = event.target.value;
    setEmail(updatedEmail);
  }

  return (
    <input value={email} onChange={handleChange} />
  );
}
```

Let’s break down how this code works!

- The square brackets on the left side of the assignment operator signal array destructuring.
- The local variable named email is assigned the current state value at index 0 from the array returned by useState().
- The local variable named setEmail() is assigned a reference to the state setter function at index 1 from the array returned by useState()
- It’s convention to name this variable using the current state variable (email) with “set” prepended.

The JSX input tag has an event listener called onChange. This event listener calls an event handler each time the user types something in this element.

React makes it easy to simplify this:

```
const handleChange = (event) => {
  const newEmail = event.target.value;
  setEmail(newEmail);
}
```

To this:

```
const handleChange = (event) => setEmail(event.target.value);
```

Or even, use object destructuring to write this:

```
const handleChange = ({target}) => setEmail(target.value);
```

All these code snippets behave the same way, so there really isn’t a right and wrong between these different ways of doing this.

Set From Previous State

Often, the next value of our state is calculated using the current state. In this case, it is best practice to update state with a callback function. If we do not, we risk capturing outdated, or “stale”, state values.

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

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

  const increment = () => setCount(prevCount => prevCount + 1);

  return (
    <div>
      <p>Wow, you've clicked that button: {count} times</p>
      <button onClick={increment}>Click here!</button>
    </div>
  );
}
```

When the button is pressed, the increment() event handler is called. Inside of this function, we use our setCount() state setter in a new way! Because the next value of count depends on the previous value of count, we pass a callback function as the argument setCount() instead of a value.

When our state setter calls the callback function, this state setter callback function takes our previous count as an argument. The value returned by this state setter callback function is used as the next value of count. We can also use setCount(count+1) and it will work the same but it is safer to use the callback method.

Arrays in State

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

const options = ["Bell Pepper", "Sausage", "Pepperoni", "Pineapple"];

export default function PersonalPizza() {
  const [selected, setSelected] = useState([]);

  const toggleTopping = ({target}) => {
    const clickedTopping = target.value;
    setSelected((prev) => {
     // check if clicked topping is already selected
      if (prev.includes(clickedTopping)) {
        // filter the clicked topping out of state
        return prev.filter(t => t !== clickedTopping);
      } else {
        // add the clicked topping to our state
        return [clickedTopping, ...prev];
      }
    });
  };

  return (
    <div>
      {options.map(option => (
        <button value={option} onClick={toggleTopping} key={option}>
          {selected.includes(option) ? "Remove " : "Add "}
          {option}
        </button>
      ))}
      <p>Order a {selected.join(", ")} pizza</p>
    </div>
  );
}
```

JS arrays are the best data model for managing and rendering JSX lists. In the above example, we are using two arrays:

- options is an array that contains the names of all of the pizza toppings available
- selected is an array representing the selected toppings for our personal pizza

The options array contains static data, meaning that it does not change. We like to define static data models outside of our function components since they don't need to recreated each time our component re-renders. In our JSX, we use the map method to render a button for each of the toppings in our options array.

The selected array contains dynamic data, which changes usually based on a user's actions. We initialize selected as an empty array. When a button is clicked, the toggleTopping event handler is called. Notice how this event handler uses information from the event object to determine which topping was clicked.

When updating an array in state, we donot just add new data to the previous array. We replace the previous array with a brand new array. This means that any information that we want to save from the previous array needs to be explicitly copied over to the new array. That's what this spread syntax does for us: ...prev

Objects in State

When we work with a set of related variables, it can be very helpful to group them in an object.

```
export default function Login() {
  const [formState, setFormState] = useState({});

  const handleChange = ({ target }) => {
    const { name, value } = target;
    setFormState((prev) => ({
      ...prev,
      [name]: value
    }));
  };

  return (
    <form>
      <input
        value={formState.firstName}
        onChange={handleChange}
        name="firstName"
        type="text"
      />
      <input
        value={formState.password}
        onChange={handleChange}
        type="password"
        name="password"
      />
    </form>
  );
}
```

A few things to notice:

- we use a state setter callback function to update state based on the previous value.
- The spread syntax is the same for objects as for arrays: `{...oldObject, newKey: newValue}`
- We reuse our event handler across multiple inputs by using the input tag's name attribute to identify which input the change event came from.

Again when updating the state with setFromState() inside a function component, we do not modify the same object. We must copy over the values from the previous object when setting the next value of state. The spread syntax makes this super easy to do!

Anytime one of the input values is updated, the handleChange() function will be called. Inside of this event handler, we use object destructuring to unpack the target property from our event object, then we use object destructuring again to unpack the name and value properties from the target object.

Inside of our state setter callback function, we wrap our curly brackets in parentheses like so:

```
setFormState((prev) => ({ ...prev }))
```

This tells JS that our curly brackets refer to a new object to be returned. We use ..., the spread operator, to fill in the corresponding fields from our previous state. Finally, we overwrite the appropriate key with its updated value. This computed property name allows us to use the string value stored by the name variable as a property key!

Separate Hooks for Separate States

While there are times when it can be helpful to store related data in a data collection like an array or object, it can also be helpful to separate data that changes separately into completely different state variables. Managing dynamic data is much easier when we keep our data models as simple as possible.

For example, if we have a single object that held state for a subject you are studying at school, it might look something like this:

```
function Subject() {
  const [state, setState] = useState({
    currentGrade: 'B',
    classmates: ['Hasan', 'Sam', 'Emma'],
    classDetails: {topic: 'Math', teacher: 'Ms. Barry', room: 201};
    exams: [{unit: 1, score: 91}, {unit: 2, score: 88}]);
  });
```

This would work, but think about how messy it could get to copy over all the other values when we need to update something in this big state object. For example, to update the grade on an exam, we would need an event handler that did something like this:

```
setState((prev) => ({
 ...prev,
  exams: prev.exams.map((exam) => {
    if( exam.unit === updatedExam.unit ){
      return {
        ...exam,
        score: updatedExam.score
      };
    } else {
      return exam;
    }
  }),
}));
```

Complex code like this is likely to cause bugs! Luckily, there is another option.. We can make more than one call to the State Hook. In fact, we can make as many calls to useState() as we want! It’s best to split into multiple state variables based on which values tend to change together. We can rewrite the previous example as follows..

```
function Subject() {
  const [currentGrade, setGrade] = useState('B');
  const [classmates, setClassmates] = useState(['Hasan', 'Sam', 'Emma']);
  const [classDetails, setClassDetails] = useState({topic: 'Math', teacher: 'Ms. Barry', room: 201});
  const [exams, setExams] = useState([{unit: 1, score: 91}, {unit: 2, score: 88}]);
  // ...
}
```

Managing dynamic data with separate state variables has many advantages, like making our code more simple to write, read, test, and reuse across components.

Often, we find ourselves packaging up and organizing data in collections to pass between components, then separating that very same data within components where different parts of the data change separately.

Example of how Hooks make State handling easier

// AppClass.js without using hooks

```
import React, { Component } from "react";
import NewTask from "../Presentational/NewTask";
import TasksList from "../Presentational/TasksList";

export default class AppClass extends Component {
  constructor(props) {
    super(props);
    this.state = {
      newTask: {},
      allTasks: []
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handleDelete = this.handleDelete.bind(this);
  }

  handleChange({ target }){
    const { name, value } = target;
    this.setState((prevState) => ({
      ...prevState,
      newTask: {
        ...prevState.newTask,
        [name]: value,
        id: Date.now()
      }
    }));
  }

  handleSubmit(event){
    event.preventDefault();
    if (!this.state.newTask.title) return;
    this.setState((prevState) => ({
      allTasks: [prevState.newTask, ...prevState.allTasks],
      newTask: {}
    }));
  }

  handleDelete(taskIdToRemove){
    this.setState((prevState) => ({
      ...prevState,
      allTasks: prevState.allTasks.filter((task) => task.id !== taskIdToRemove)
    }));
  }

  render() {
    return (
      <main>
        <h1>Tasks</h1>
        <NewTask
          newTask={this.state.newTask}
          handleChange={this.handleChange}
          handleSubmit={this.handleSubmit}
        />
        <TasksList
          allTasks={this.state.allTasks}
          handleDelete={this.handleDelete}
        />
      </main>
    );
  }
}
```

// AppFunction.js using Hooks

```
import React, { useState } from "react";
import NewTask from "../Presentational/NewTask";
import TasksList from "../Presentational/TasksList";

export default function AppFunction() {
  const [newTask, setNewTask] = useState({});
  const handleChange = ({ target }) => {
    const { name, value } = target;
    setNewTask((prev) => ({ ...prev, id: Date.now(), [name]: value }));
  };

  const [allTasks, setAllTasks] = useState([]);
  const handleSubmit = (event) => {
    event.preventDefault();
    if (!newTask.title) return;
    setAllTasks((prev) => [newTask, ...prev]);
    setNewTask({});
  };
  const handleDelete = (taskIdToRemove) => {
    setAllTasks((prev) => prev.filter(
      (task) => task.id !== taskIdToRemove
    ));
  };

  return (
    <main>
      <h1>Tasks</h1>
      <NewTask
        newTask={newTask}
        handleChange={handleChange}
        handleSubmit={handleSubmit}
      />
      <TasksList allTasks={allTasks} handleDelete={handleDelete} />
    </main>
  );
}
```

The Effect Hook

Why Use useEffect?

Before Hooks, function components were only used to accept data in form of props and return some JSX to be rendered. However, as we learned, the State Hook allows us to manage dynamic data, in the form of component state, within our function components.

We use the Effect Hook to run some JS code after each render, such as:

- Fetch data from a backend service
- Subscribing to a stream of data
- Managing timers and intervals
- Reading from and making changed to the DOM

Most interesting components re-render multiple times throughout their lifetime and these key moments present the perfect opportunity to execute these “side effects”.

There are three key moments when the Effect Hook can be utilized:

1. When the component is first added, or mounted, to the DOM and renders
2. When the state or props change, causing the component to re-render
3. When the component is removed, or unmounted from the DOM.

Function component Effects

```
import React, { useState, useEffect } from 'react';

function PageTitle() {
  const [name, setName] = useState('');

  useEffect(() => {
    document.title = `Hi, ${name}`;
  });

  return (
    <div>
      <p>Use the input field below to rename this page!</p>
      <input onChange={({target}) => setName(target.value)} value={name}
type='text' />
    </div>
  );
}
```

Here, first we import the Effect Hook from the react library, like:

```
import { useEffect } from ‘react’;
```

The Effect Hook is used to call another function that does something for us so there is nothing returned when we call the `useEffect()` function.

The first argument passed to the useEffect() function is the callback function that we want React to call after each time this component renders. We will refer to this callback function as our effect.

In our example, the effect is:

```
() => { document.title = name; }
```

In our effect, we assign the value of the name variable to the document.title.

Notice how we use the current state inside of our effect. Even though our effect is called after the component renders, we still have access to the variables in the scope of our function component! When React renders our component, it will update the DOM as usual, and then run our effect after the DOM has been updated. This happens for every render, including the first and last one.

Clean Up Effects

Some effects require cleanup. For example, we might want to add event listeners to some element in the DOM, beyond the JSX in our component. When we add event listeners to the DOM, it is important to remove those event listeners when we are done with them to avoid memory leaks!

```
useEffect(()=>{
  document.addEventListener('keydown', handleKeyPress);
  return () => {
    document.removeEventListener('keydown', handleKeyPress);
  };
})
```

If our effect didn’t return a cleanup function, then a new event listener would be added to the DOM’s document object every time that our component re-renders. Not only would this cause bugs, but it could cause our application performance to diminish and maybe even crash!

Control When Effects Are Called

The useEffect() function calls its first argument (the effect) after each time a component renders. We’ve learned how to return a cleanup function so that we don’t create performance issues and other bugs, but sometimes we want to skip calling our effect on re-rnders altogether.

It is common, when defining function components, to run an effect only when the component mounts (renders the first time), but not when the component re-renders. The Effect Hook makes this very easy for us to do! If we want to only call our effect after the first render, we pass an empty array to useEffect() as the second argument. This second argument is called the dependency array.

The dependency array is used to tell the useEffect() method when to call our effect and when to skip it. Our effect is always called after the first render but only called again if something in our dependency array has changed values between renders.

```
useEffect(() => {
  alert("component rendered for the first time");
  return () => {
    alert("component is being removed from the DOM");
  };
}, []);
```

Without passing an empty array as the second argument to the useEffect() above, those alerts would be displayed before and after every render of our component, which is clearly not when those messages are meant to be displayed. Simply, passing [] to the useEffect() function is enough when the effect and cleanup functions are called!

Fetch Data

When building software, we often start with default behaviors then modify them to improve performance. We've learned that the default behavior of the Effect Hook is to call the effect function after every single render. Next, we learned that we can pass an empty array as the second argument for useEffect() if we only want our effect to be called after the component's first render.

When our effect is responsible for fetching data from a server, we pay extra close attention to when our effect is called. Unnecessary round trips back and forth between our React components and the server can be costly in terms of:

- Processing
- Performance
- Data usage for mobile users
- API service fees

When the data that our components need to render doesn't change, we can pass an empty dependency array, so that the data is fetched after the first render. When the response is received from the server, we can use a state setter from the State Hook to store the data from the server's response in our local component state for future renders. Using the State Hook and the Effect Hook together in this way is a powerful pattern that saves our components from unnecessarily fetching new data after every render!

An empty dependency array signals to the Effect Hook that our effect never needs to be re-run, that it doesn’t depend on anything. Specifying zero dependencies means that the result of running that effect won’t change and calling our effect once is enough.

A dependency array that is not empty signals to the Effect Hook that it can skip calling our effect after re-renders unless the value of one of the variables in our dependency array has changed. If the value of a dependency has changed, then the Effect Hook will call our effect again!

Rules of Hooks

There are two main rules to keep in mind when using Hooks:

- Only call Hooks at the top level
- Only call Hooks from React functions

When React builds the Virtual DOM, the library calls the functions that define our components over and over again as the user interacts with the user interface. React keeps track of the data and functions that we are managing with Hooks based on their order in the function component’s definition. For this reason, we always call our Hooks at the top level; we never call hooks inside of loops, conditions, or nested functions.

Secondly, Hooks can only be used in React Functions. We cannot use Hooks in class components and we cannot use Hooks in regular JS functions. We’ve been working with useState() and useEffect() in function components, and this is the most common use. The only other place where Hooks can be used is within custom hooks. Custom Hooks are incredibly useful for organizing and reusing stateful logic between function components.