THE STATE HOOK

Why Use Hooks?

As React developers, we love breaking down complex problems into simple pieces.

Classes, however, are not simple. They:

- are difficult to reuse between components
- are tricky and time-consuming to test
- have confused many developers and caused lots of bugs

The React core team heard all of this feedback from developers like us, and they engineered an incredible solution for us! 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. Keep in mind that function components and React Hooks do not replace class components. They are completely optional; just a new tool that we can take advantage of.

Note: If you're familiar with lifecycle methods of class components, you could say that Hooks let us "hook into" state and lifecycle features directly from our function components.

React offers a number of built-in Hooks. A few of these include `useState()`, `useEffect()`, `useContext()`, `useReducer()`, and `useRef()`. See the full list in the docs. In this lesson, we'll learn different ways to manage state in a function component.

Instructions

1.
Review the class component defined in the **AppClass.js** file.

Notice how the rendering logic has been delegated to separate presentational function components. This `AppClass` component uses a constructor, its own class methods, as well

as `this.setState()` and `this.render()` methods from React's `Component` class.

Make some predictions about how this code behaves, then run the code to check your predictions!

Checkpoint 2 Passed

2.

Buckle your seat belt. We are about to adventure into new territory.

Open the **AppFunction.js** file. We will learn how this code works in the next few exercises. Don't worry about the details of what is going on here just yet, but take a few moments to read through the definition of this function component and develop some theories about what this code may be doing.

Open the **index.js** file and change where this module imports the App component from, so that we can render the `AppFunction` instead of the `AppClass` component. Press run to see how the code behaves!

Checkpoint 3 Passed

Hint

Default exports can be assigned to any local variable name when they are imported. Switching these two lines of code assigns a different value to the same variable name of `App`.

In **index.js**, replace these lines:

```
import App from "./Container/AppClass";
// import App from "./Container/AppFunction";
```

with these lines:

```
// import App from "./Container/AppClass";
import App from "./Container/AppFunction";
```

Want more of a review on this? Here is an awesome lesson on JavaScript modules

index.js

```
import React from "react";
import ReactDOM from "react-dom";
// import App from "./Container/AppClass";
import App from "./Container/AppFunction";
```

```
ReactDOM.render(
  <App />,
  document.getElementById("app")
);
```

AppClass.js

```
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

```
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={handleDel
ete} />
    </main>
  );
}
```

Update Function Component State

Let’s get started with the State Hook, 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 JavaScript function defined in the 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

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

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

Let's have a look at an example of a function component using the State Hook:

```
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>
  );
}
```

Notice how the state setter function, `setToggle()`, is called by our `onClick` *event listeners*. To update the value of `toggle` and re-render this component with the new value, all we need to do is call the `setToggle()` function with the next state value as an argument.

No need to 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 the 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!

Instructions

1.
Import the default export from the ‘react’ library and call it `React`. We will be using the State Hook, so go ahead and import the named export `useState` from the ‘react’ library as well.

Checkpoint 2 Passed

Hint
To import the default export and named exports from a module, use a comma to separate these and use curly brackets to differentiate named exports from the default export.

```
import defaultExport, { namedExport } from 'react';
```

2.
Use `const` to declare and assign the return values from calling `useState()` to `color` and `setColor`.

Checkpoint 3 Passed

Hint
Recall that the return value of the `useState()` function is an array with two values in it:

```
const [currentState, stateSetter] = useState();
```

For this component, we want to use the `const` keyword to declare our local variables. Give the current state a name of `color` and give the state setter a name of `setColor`.

3.
In the JSX, use `onClick` event listeners to call the `setColor()` state setter function with the appropriate color when the user clicks on each of these buttons.

Checkpoint 4 Passed

Hint
Try using this line from our example above for how to add the `onClick` event listener to a JSX button tag:

```
 <button onClick={() => setToggle('On')}>
```

Notice how the value for the `onClick` attribute is an inline callback function that doesn’t use any arguments. It calls the state setter function with the new value as an argument.

Instead of `setToggle()`, our state setter for this component is `setColor()`, and instead of `'On'`, our argument of the state setter function is the color name associated with each button.

4.

Update the `divStyle` object so that the `backgroundColor` property is set to our current state value instead of always showing purple.

Checkpoint 5 Passed

Hint

There are lots of different ways to add styling to our React components. We'll cover this in more detail in later lessons, for now, let's focus on using a variable instead of the hard-coded string value of "purple" in the line below:

```
const divStyle = {backgroundColor: 'purple'};
```

Replace the `'purple'` string value with our `color` current state variable, then click different buttons to see the background color change with our state variable!

ColorPicker.js

```
// import the default export and the named export `useState`
 from the 'react' library
import React, { useState } from "react";

export default function ColorPicker() {
  // call useState and assign its return values to `color` a
nd `setColor`
  const [color, setColor] = useState()

 const divStyle = {backgroundColor: color};

  return (
    <div style={divStyle}>
      <p>The color is {color}</p>
      <button onClick={() => setColor('Aquamarine')}>
        Aquamarine
      </button>
      <button onClick={() => setColor('BlueViolet')}>
        BlueViolet
      </button>
      <button onClick={() => setColor('Chartreuse')}>
        Chartreuse
```

```
      </button>
      <button onClick={() => setColor('CornflowerBlue')}>
        CornflowerBlue
      </button>
    </div>
  );
}
```

Initialize State

Great work building out your first stateful function component in the last exercise. Just like you 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!

Have a look at the following function component. What data type does this state variable hold?

```
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>
  );
}
```

The `ToggleLoading()` function component above is using the simplest of all data types, a boolean. Booleans are frequently used in React applications to represent whether data has loaded or not. In the example above, we see that `true` and `false` values are passed to the state setter, `setIsLoading()`. This code works just fine as is, but

what if we want our component to start off with `isLoading` set to `true`?

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 it can be unclear to the humans reading our 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`.

Instructions

1.

Professional web development is a team sport. Thankfully, a coworker was able to help refactor the code from your `ColorPicker()` component to support more colors. Now, our product owner wants the app to start off with a color of "Tomato" when it first shows up on the screen.

Modify our current `ColorPicker()` component to initialize state so that "Tomato" is the selected `color` for our component's first render.

Checkpoint 2 Passed

Hint

Pass the string `'Tomato'` as an argument to the `useState()` function, to use this as the initial value of `color`:

```
const [color, setColor] = useState(initialStateValue);
```

If you are curious about this line of code:

```
colorNames.map((colorName)=>(
```

have a look at MDN docs on this map() method of JavaScript arrays.

ColorPicker.js

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

const colorNames = ['Aquamarine', 'BlueViolet', 'Chartreuse'
, 'CornflowerBlue', 'Thistle', 'SpringGreen', 'SaddleBrown',
 'PapayaWhip', 'MistyRose'];

export default function ColorPicker() {
  const [color, setColor] = useState("Tomato");

 const divStyle = {backgroundColor: color};

  return (
    <div style={divStyle}>
      <p>Selected color: {color}</p>
      {colorNames.map((colorName)=>(
        <button
          onClick={() => setColor(colorName)}
          key={colorName}>
             {colorName}
        </button>
      ))}
    </div>
  );
}
```

Use State Setter Outside of JSX

Let's see how to manage the changing value of a string as a user types into a text input field:

```
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. In the example above, our event handler is defined inside of the definition for our function component, but outside of our JSX. Earlier in this lesson, we wrote our event handlers right in our JSX. Those inline event handlers work perfectly fine, but when we want to do something more interesting than just calling the state setter with a static value, it's a good idea to separate that logic from everything else going on in our JSX. This separation of concerns makes our code easier to read, test, and modify.

This is so common in React code, that we can comfortably 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 just write this:

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

All three of these code snippets behave the same way, so there really isn't a right and wrong between these different ways of doing this. We'll use the last, most concise version moving forward.

Instructions

1.

Declare and assign values for our current state and state setter with `useState()`. Use `phone` as the name of our current state variable.

Checkpoint 2 Passed

Hint

A convention used to help other developers understand our code is to give a descriptive variable name for the current state and then prefix that variable name with the word 'set' for our state setter. This convention makes it crystal clear which functions are responsible for updating which values!

Be sure to:

- Name our current state variable: `phone`
- Name our state setter function: `setPhone()`
- Use array destructuring to assign these variables the values returned by calling `useState()`
- Initialize state with the value of an empty string like so: `useState('')`
- Import the State Hook from the 'react' library like so:

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

2.

Add the `value` attribute and the `onChange` event listener to our JSX input tag. Give these attributes the values of our current state variable and event handler function.

Checkpoint 3 Passed

Hint

We want to give our input tag the following attributes:

```
<input value={currentState} onChange={eventHandler} id='phone-input' />
```

- Our currentState is `phone`
- Our eventHandler is `handleChange` (not our state setter, `setPhone`)

3.
Use our state setter to update the state only when the value from the user's change event passes our regular expression test for valid phone number strings.

Curious about the `/^\d{1,10}$/` in our code? Check out this great lesson to learn about regular expressions!

Checkpoint 4 Passed

Hint
We want our event handler to follow this format:

```
const handleChange = ({ target })=> {
  const newPhone = target.value;
  const isValid = validPhoneNumber.test(newPhone);
  if (isValid) {
    stateSetter(nextValueToUseForState);
  }
  // just ignore the event, when new value is invalid
};
```

- Our state setter is `setPhone()`
- Our next value to use for state is stored by our `newPhone` variable

PhoneNumber.js

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

// regex to match numbers between 1 and 10 digits long
const validPhoneNumber = /^\d{1,10}$/;

export default function PhoneNumber() {
  // declare current state and state setter
  const [phone, setPhone] = useState('');

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

```
    const isValid = validPhoneNumber.test(newPhone);
    if (isValid) {
        // update state
        setPhone(target.value);
    }
    // just ignore the event, when new value is invalid
  };

  return (
    <div className='phone'>
      <label for='phone-input'>Phone: </label>
      <input value={phone} onChange={handleChange}id='phone-
input' />
    </div>
  );
}
```

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.

Let’s take a look at the following code:

```
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 for `setCount()` instead of a value (as we've done in previous exercises).

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

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` (in this case `prevCount + 1`). Note: We can just call `setCount(count +1)` and it would work the same in this example… but for reasons that are out of scope for this lesson, it is safer to use the callback method. If you'd like to learn more about why the callback method is safer, this section of the docs is a great place to start.

Instructions

1.

Define a `goBack()` event handler. Because our next value of state depends on the previous state value, this function should call the state setter with a callback function. Our state setter callback function needs to compute the next value of `questionIndex` using an argument named `prevQuestionIndex`. Add an event listener to the "Go Back" button that will call our newly defined event handler.

Checkpoint 2 Passed

Hint

Use the following function definition for this event handler:

```
const goBack = () => setQuestionIndex((prevQuestionIndex) => prevQuestionIndex - 1);
```

Don't forget to use an `onClick` event listener to call this event handler when our users click on the back button!

2.

Define a `goToNext()` event handler. Because our next value of state depends on the previous state value, this function should call the state setter with a callback function. Our state

setter callback function needs to compute the next value of `questionIndex` using an argument named `prevQuestionIndex`. Add an event listener to the “Next Question” button that will call our newly defined event handler.

Checkpoint 3 Passed

Hint

Use the following function definition for this event handler:

```
const goToNext = () => setQuestionIndex((prevQuestionIndex) => prevQuestionIndex + 1);
```

Don’t forget to use an `onClick` event listener to call this event handler when our users click on the next button!

3.

Add an `onFirstQuestion` variable with a boolean value then use that value to toggle the `disabled` attribute of the “Go Back” button on and off.

Checkpoint 4 Passed

Hint

JavaScript arrays use zero-based indexing, so we can determine whether `onFirstQuestion` has a value of `true` or `false` by checking if `questionIndex` currently has a value of `0`!

To use this boolean value, add the following attribute to the “Go Back” button:

```
disabled={onFirstQuestion}
```

QuizNavBar.js

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

export default function QuizNavBar({ questions }) {
  const [questionIndex, setQuestionIndex] = useState(0);

  // define event handlers
  const goBack = () =>
    setQuestionIndex((prevQuestionIndex) => prevQuestionIndex - 1);
```

```
  const goToNext = () =>
    setQuestionIndex((prevQuestionIndex) => prevQuestionInde
x + 1);

  // determine if on the first question or not
  const onFirstQuestion = questionIndex === 0;

  const onLastQuestion = questionIndex === questions.length
- 1;

  return (
    <nav>
      <span>Question #{questionIndex + 1}</span>
      <div>
        <button onClick={goBack} disabled={onFirstQuestion}>
          Go Back
        </button>
        <button onClick={goToNext} disabled={onLastQuestion}
>
          Next Question
        </button>
      </div>
    </nav>
  );
}
```

Arrays in State

Think about the last time that you ordered a pizza online. Mmmmm...

Part of the magical website that brought you tasty food was built with code like this:

```
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>
  );
}
```

JavaScript arrays are the best data model for managing and rendering JSX lists. In this 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 be 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*, meaning that it 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 do not 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 our new array. That's what this spread syntax does for us: `...prev`.

Notice how we use the `includes()`, `filter()`, and `map()` methods of our arrays. If these are new to you, or you just want a refresher, take a minute to review these array methods. We don't need to be full-fledged JavaScript gurus to build React UIs, but know that investing time to strengthen our JavaScript skills, will always help us do more faster (and have a lot more fun doing it) as React developers.

Instructions

1.
Declare and initialize a state variable called `cart` that will keep track of a list of string values.

Each of these string values represents a grocery item that we've added to our shopping cart. We'll add event listeners and event handlers to add and remove items to our cart in the coming steps.

For now, let's get started by initializing our `cart` with the value of an empty array for the first render.

Checkpoint 2 Passed

Hint

Using the following format:

```
const [currentState, stateSetter] = useState(initialStateArgument);
```

- let's name our current state: `cart`
- let's name our state setter : `setCart()`

- let's call the `useState()` function with an argument of `[]` so that current state is an array (not `undefined`) during the first render of this component

2.
Add parameters to our two event handlers. `addItem()` should accept an argument called `item` and `removeItem()` should accept an item called `targetIndex`. Don't worry about writing the function body for these event handlers just yet, we'll do that in the next few steps!

Checkpoint 3 Passed

Hint
Our event handlers should look like this:

```
const addItem = (addItemParameter) => {

};

const removeItem = (removeItemParameter) => {

};
```

And just so that we are all on the same page, let's:

- give `addItemParameter` a name of `item`
- give `removeItemParameter` a name of `targetIndex`

3.
Use array spread syntax to add a new item to our `cart` state when the `addItem()` function is called.

Checkpoint 4 Passed

Hint
Because our next state is derived from our previous state, let's call our state setter with a state setter callback function.

So, our `addItem()` event handler is responsible for calling our `setCart()` state setter with the following argument:

```
(prev) => {
  return [item, ...prev];
}
```

4.
When the `removeItem()` function is called, use the array `filter()` method to remove the item from our state that's located at the index of the item that was clicked in our list.

Why would we want to use the index of the clicked item instead of the item itself when determining what to remove from our data model? Say that we have two of the same item in an array. Using the value to remove the item would remove all items with that value, so we use the index as a unique identifier.

Checkpoint 5 Passed

Hint

The callback function passed to the `filter()` method is called with each item in the array as its first argument and the index of each of these items as its second argument. We want our callback function to return `true` for every item that does not have an index matching the `targetIndex` of the clicked list item that we want to remove.

For some more explanation and examples of `filter()` in action head over here: MDN documentation on the `filter()` method of JavaScript arrays.

GroceryCart.js

```
import React, { useState } from "react";
import ItemList from "./ItemList";
import { produce, pantryItems } from "./storeItems";

export default function GroceryCart() {
  const [cart, setCart] = useState([]);

  const addItem = (item) => {
    setCart((prev) => {
      return [item, ...prev];
    });
  };

  const removeItem = (targetIndex) => {
    setCart((prev) => {
      return prev.filter((item, index) => index !== targetIn
dex);
    });
  };
```

```
  return (
    <div>
      <h1>Grocery Cart</h1>
      <ul>
        {cart.map((item, index) => (
          <li onClick={() => removeItem(index)} key={index}>
            {item}
          </li>
        ))}
      </ul>
      <h2>Produce</h2>
      <ItemList items={produce} onItemClick={addItem} />
      <h2>Pantry Items</h2>
      <ItemList items={pantryItems} onItemClick={addItem} />
    </div>
  );
}
```

ItemList.js

```
import React from "react";

export default function ItemList({ items, onItemClick }) {
  const handleClick = ({ target }) => {
    const item = target.value;
    onItemClick(item);
  };
  return (
    <div>
      {items.map((item, index) => (
        <button value={item} onClick={handleClick} key={inde
x}>
          {item}
        </button>
      ))}
    </div>
  );
}
```

storeItems.js

```
export const produce = [
  "Carrots",
  "Cucumbers",
  "Bell peppers",
  "Avocados",
  "Spinach",
  "Kale",
  "Tomatoes",
  "Bananas",
  "Lemons",
  "Ginger",
  "Onions",
  "Potatoes",
  "Sweet potatoes",
  "Purple cabbage",
  "Broccoli",
  "Mushrooms",
  "Cilantro"
];

export const pantryItems = [
  "Chia",
  "Goji berries",
  "Peanut butter",
  "Bread",
  "Cashews",
  "Pumpkin seeds",
  "Peanuts",
  "Olive oil",
  "Sesame oil",
  "Tamari",
  "Pinto beans",
  "Black beans",
  "Coffee",
  "Rice",
  "Dates",
```

```
  "Quinoa"
];
```

Objects in State

When we work with a set of related variables, it can be very helpful to group them in an object. Let's look at an example!

```
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

Once again, when updating the state with `setFormState()` 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. Thankfully, 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 JavaScript 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. Did you notice the square brackets around the `name`? This Computed Property Name allows us to use the string value stored by the `name` variable as a property key!

Instructions

1.
Throughout our JSX, we are looking up properties stored on the `profile` object. On the first render, this is a problem because attempting to get the value of a property from an object that has not been defined causes JavaScript to throw an error.

To defend against these errors, let's initialize `profile` as an empty object!

Checkpoint 2 Passed

Hint
To initialize state, we call the `useState()` function with the value that we want React to use as state during the first render. Pass `{}` as an argument to our `useState()` function call.

2.

Add the event listeners to our JSX tags to call `handleChange()` whenever a user types in an input field.

Checkpoint 3 Passed

Hint

Add the `onChange` event listener as an attribute of each of our `<input>` tags. Use each of these event listeners to call the same event handler, `handleChange()`.

3.

Let's make our `handleChange()` function a bit easier to read. Use object destructuring to initialize `name` and `value` in a more concise way.

Checkpoint 4 Passed

Hint

Replace the current `name` & `value` declarations with this destructuring statement:

```
const {name, value} = target;
```

4.

There's a bug in our code! Have you noticed it? Try typing in one input, then type in a different input. What happens? Why?

Each time that we call `setProfile()` in our event handler, we give `profile` the value of a new object with the `name` and `value` of the input that most recently changed, but we lose the values that were stored for inputs with any other `name`.

Let's use the spread operator to fix this bug. We want to copy over all of the values from our previous `profile` object whenever we call our state setter function. Use `prevProfile` as the argument for our state setter callback function.

Checkpoint 5 Passed

Hint

- Name the first argument of the callback function: `prevProfile`
- Use the spread operator to copy over the values from `prevProfile` into our new state object:

```
setProfile((prevProfile) => ({
  // use the spread operator here
  [name]: value
}));
```

5.

Add an event listener to call our `handleSubmit()` function when the user submits the form.

Checkpoint 6 Passed

Hint

Much like we use the `onChange` event listener to listen for changes in our `<input>` elements, we use `onSubmit` to listen for submit events in our `<form>` elements.

If you'd like, have a look at some working JSX examples of this!

EditProfile.js

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

export default function EditProfile() {
  const [profile, setProfile] = useState({});

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

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(JSON.stringify(profile, '', 2));
  };

  return (
    <form onSubmit={handleSubmit}>
      <input onChange={handleChange}
        value={profile.firstName || ''}
        name="firstName"
        type="text"
        placeholder="First Name"
      />
      <input onChange={handleChange}
        value={profile.lastName || ''}
```

```
        type="text"
        name="lastName"
        placeholder="Last Name"
      />
      <input onChange={handleChange}
        value={profile.bday || ''}
        type="date"
        name="bday"
      />
      <input onChange={handleChange}
        value={profile.password || ''}
        type="password"
        name="password"
        placeholder="Password"
      />
      <button type="submit">Submit</button>
    </form>

  );
}
```

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 had 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;
    }
  }),
}));
```

Yikes! 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 state 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. The wonderful thing about working with Hooks is that we have the freedom to organize our data the way that makes the most sense to us!

If you'd like, have a look at another example of using multiple State Hooks for managing separate data.

Instructions

1.
Inside of the `MusicalRefactored` component, rewrite the state from `Musical` so that its state object is split into three separate variables: `title`, `actors`, and `locations`, each with their own state setter function.

React's State Hook allows us to name our state setters whatever we want, but you may have noticed a pattern:

- `toggle` & `setToggle()`
- `isLoading` & `setIsLoading()`
- `email` & `setEmail()`

Checkpoint 2 Passed

Hint
Rather than starting from scratch, copy the starter code below, and fill in the missing pieces!

```
function MusicalRefactored() {
  const [/*current state variable name here*/, setTitle] = useState(/*initial state for title here*/);
  const [/*current state variable name here*/, setActors] = useState(/*initial state for actors here*/);
  const [locations, /*following our naming convention, name this state setter here*/] = useState({
    Chicago: {
      dates: ["1/1", "2/2"],
      address: "chicago theater"},
      /* finish the initial state value for locations here*/
      }
);
}
```

Musical.js

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

function Musical() {
   const [state, setState] = useState({
    title: "Best Musical Ever",
    actors: ["George Wilson", "Tim Hughes", "Larry Clements"
],
    locations: {
      Chicago: {
        dates: ["1/1", "2/2"],
        address: "chicago theater"},
      SanFrancisco: {
        dates: ["5/2"],
        address: "sf theater"
      }
    }
  })
 }

function MusicalRefactored() {
const [title, setTitle] = useState("Best Musical Ever");
const [actors, setActors] = useState(["George Wilson", "Tim
Hughes", "Larry Clements"]);
const [locations, setLocations] = useState({
      Chicago: {
        dates: ["1/1", "2/2"],
        address: "chicago theater"},
      SanFrancisco: {
        dates: ["5/2"],
        address: "sf theater"
      }
    })
```

Lesson Review

Awesome work, we can now build “stateful” function components using the `useState` React Hook!

Let's review what we learned and practiced in this lesson:

- With React, we feed static and dynamic data models to JSX to render a view to the screen
- Use Hooks to "hook into" internal component state for managing dynamic data in function components
- We employ the State Hook by using the code below:
  - `currentState` to reference the current value of state
  - `stateSetter` to reference a function used to update the value of this state
  - the `initialState` argument to initialize the value of state for the component's first render

```
const [currentState, stateSetter] = useState( initialState );
```

- Call state setters in event handlers
- Define simple event handlers inline with our JSX event listeners and define complex event handlers outside of our JSX
- Use a state setter callback function when our next value depends on our previous value
- Use arrays and objects to organize and manage related data that tends to change together
- Use the spread syntax on collections of dynamic data to copy the previous state into the next state like so: `setArrayState((prev) => [ ...prev ])` and `setObjectState((prev) => ({ ...prev }))`
- Split state into multiple, simpler variables instead of throwing it all into one state object

Instructions

1.
Remember this class component from the beginning of this lesson? The same component was defined as a function component for your review, but at the time, a lot of that code probably didn't make much sense! Not only would that code make sense to you if you had a look at it now, but after everything you've learned and practiced you can now write that code yourself!

Take a moment to read through this class component defined in the **AppClass.js** file, then switch over to the **AppFunction.js** file. Without any guidance, see if you can define a function component that behaves just like this class

component, but saves us all of the complexity of dealing with JavaScript classes!

When you are ready to start testing your code, change the import statements in the **index.js** file, just like we did at the beginning of this lesson!

Good luck, we believe in you!

Checkpoint 2 Passed

Hint

If you are feeling stuck, refer to earlier exercises in this lesson or even go all of the way back to the first exercise to see our example code of a function component that could be used to replace this class component.

It's completely okay if your variable names are different than the ones in our example, there are lots of different ways to define a component correctly. The key is to be thoughtful about the code you write, making it clear to the machines and other developers who will be reading it!

Index.js

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./Container/AppClass";
// import App from "./Container/AppFunction";

ReactDOM.render(
  <App />,
  document.getElementById("app")
);
```

AppClass.js

```
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

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

export default function AppFunction() {
  // hook your code up here ;)
  const [newTask, setNewTask] = useState({});
  const handleChange = ({ target }) => {
    const { name, value } = target;
    setNewTask((prev) => ({ ...prev, id: Date.now(),     [na
me]: 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={handleDel
ete} />
    </main>
  );

}
```