THE EFFECT HOOK

Why Use useEffect?

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

In this lesson, we'll use the Effect Hook to run some JavaScript code after each render, such as:

- fetching data from a backend service
- subscribing to a stream of data
- managing timers and intervals
- reading from and making changes to the DOM

Why after each render?
Most interesting components will 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.

Later on in this lesson, we'll learn how to further fine-tune exactly when this code executes.

Instructions

In the editor, we've defined a component as both a class and a function, each with the same "side effects".

Even if you are unfamiliar with class component lifecycle methods, start by having a look at both implementations, just to get a sense of both options.

Note: Understanding lifecycle methods in class components is not a pre-requisite for this lesson.
In the class component in **PageTitleClass.js**, the logic for setting the document title is split between two functions – `componentDidMount()` and `componentDidUpdate()`.

Compare this to the function component in **PageTitleFunction.js**, where the logic is written in one place – `useEffect()`.

Both component implementations have the same behavior, but reading and maintaining the function component will be easier.

Ready to start using the Effect Hook? Great! We'll dive into the details of how to use this Hook throughout this lesson!

PageTitleClass.js

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

export default class PageTitle extends Component {
  constructor(props) {
    super(props);
    this.state = {
      name: ''
    };
  }

  componentDidMount() {
    document.title = this.state.name;
  }

  componentDidUpdate() {
    document.title == `Hi, ${this.state.name}`;
  }

  render() {
    return (
      <div>
        <p>Use the input field below to rename this page!</p
>
        <input
```

```
          onChange={({target}) => this.setState({ name: targ
et.value })}
          value={this.state.name}
          type='text' />
      </div>
    );
  }
}
```

PageTitleFunction.js

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

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

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

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

Function Component Effects

Let's break down how our `PageTitle()` function component is using the Effect Hook to execute some code after each render!

```
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>
  );
}
```

First, we import the Effect Hook from the `react` library, like so:

```
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`. For more on this syntax, have a look at this explanation of the document’s title property.

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.

Instructions

1.

Import the Effect Hook, as well as State Hook and React from the `'react'` library.

Make sure to import everything on one line.

Checkpoint 2 Passed

Hint

If we only needed `useEffect()` in this module, then we could just write:

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

But we need to combine this with the variables that we are already importing:

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

To import the default export and two named exports from `'react'`, we combine these like so:

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

2.

Call `useEffect()` with a callback function that creates an [alert]() with the current value of `count`. Start clicking the button to see when our `alert()` function is called and be sure that it is logging the values that we'd expect!

Checkpoint 3 Passed

Hint

Call `useEffect()`, with the following effect as its first argument:

```
()                               =>                              {
  alert(count)
}
```

Be sure to use the Effect Hook after we call the State Hook, so that we have access to `count` and before the `handleClick()` function. Hooks must be used at the beginning of a component definition (more on this later in the lesson).

3.

Use a template literal so that the message in our alert dialog reads: “Count: 0”, then “Count: 1”, then “Count: 2”, etc.

Checkpoint 4 Passed

Hint

Update our effect so that, rather than calling `alert()` with `count`, it is called with a template literal like so: `Count: ${count}`, like this:

```
()                              =>                             {
  alert(`Count:                                        ${count}`);
}
```

Counter.js

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

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

   useEffect(() => {
    alert(`Count: ${count}`);
  });

  const handleClick = () => {
    setCount((prevCount) =>  prevCount + 1);
  };

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>
        Click me
      </button>
    </div>
  );
}
```

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!

Let's consider the following effect:

```
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!

Because effects run after every render and not just once, React calls our cleanup function before each re-render and before unmounting to clean up each effect call.

If our effect returns a function, then the `useEffect()` Hook always treats that as a cleanup function. React will call this cleanup function before the component re-renders or unmounts. Since this cleanup function is optional, it is our responsibility to return a cleanup function from our effect when our effect code could create memory leaks.

Instructions

1.

Write an event handler named `increment()`. Define this function so that it calls `setClickCount()` with a state setter callback function, adding `1` to the previous value of `clickCount`.

Checkpoint 2 Passed

Hint

Our state setter callback function should accept a single argument, add 1 to it, then return that value, like so: `(prev) => prev + 1`.

This callback function function is called by our state setter, which is called by our event handler, like so:

```
const                  increment                  = ()                  =>
setClickCount(stateSetterCallbackFunction);
```

2.

Import the `useEffect()` hook and call it with an effect that adds an event listener for `'mousedown'` events on the `document` object. When a `"mousedown"` event occurs anywhere on the `document`, we want our `increment()` event handler to be called.

Checkpoint 3 Passed

Hint

Don’t worry about a cleanup function yet. For now, just focus on adding an event listener to your effect, like this:

```
    document.addEventListener('mousedown', increment);
```

3.

If you haven’t already, run our code and click around the browser window. What is happening? Why is this happening?

Each time that our component renders, our effect is called, adding another event listener. With just a few clicks and rerenders, we have attached a lot of event listeners to the DOM! We need to clean up after ourselves!

Update our effect so that it returns a cleanup function that will remove our last event listener from the DOM.

Checkpoint 4 Passed

Hint

Our cleanup function is a new function that our effect returns. This cleanup function needs to use the `removeEventListener()` like this:

```
()                              =>                              {
    document.removeEventListener('mousedown',      increment);
}
```

Counter.js

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

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

  const increment = () => setClickCount((prev) => prev + 1);

  useEffect(() => {
    document.addEventListener('mousedown', increment);
    return () => {
      document.removeEventListener('mousedown', increment);
    };
  });

  return (
      <h1>Document Clicks: {clickCount}</h1>
  );
}
```

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-renders 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.

We will continue to learn more about this second argument over the next few exercises, but for now, we’ll focus on using an empty dependency array to call an effect when a component first mounts, and if a cleanup function is returned by our effect, calling that when the component unmounts.

```
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 to configure when the effect and cleanup functions are called!

Instructions

1.

Let’s get started by using four functions to advance the number stored by `time` each second:

- `useEffect()` - the Effect Hook, imported from the ‘react’ library
- `setInterval()` - W3Schools Documentation
- `setTime()` - our state setter function
- state setter callback function - used by `setTime()` because we want to calculate the next value of `time` based on the previous value of `time`

Add an effect that uses the `setInterval()` function to call `setTime()` every second (or 1000 ms).

Checkpoint 2 Passed

Hint

- `useEffect()` - calls our effect
- `setInterval()` - takes two arguments, a function to call repeatedly and the number of milliseconds to wait between each time it calls that function
- `setTime()` - our state setter function
- state setter callback function - call `setTime()` with this function as its only argument: `(prev) => prev + 1)`

When we put all of these together, we get:

```
useEffect(()                  =>                       {
  setInterval(()                  =>                   {
    setTime((prev)         =>            prev          + 1);
  },                                                1000);
});
```

2.

Well that doesn’t look quite right now does it? Our `time` value is updating way too quickly! Any idea why that may be happening?

By default, the Effect Hook calls our effect after every render. Our effect is creating a new interval that updates the value of `time` each second. We keep adding more and more intervals that keep updating the same `time` variable. We need to clean up our old intervals before adding new ones!

In your effect, use the `return` keyword to return a cleanup function. Our cleanup function will use the `clearInterval()` function. For more on how to use this JavaScript timer function have a quick look at W3Schools’s explanation.

Checkpoint 3 Passed

Hint

Add the following cleanup function to our effect:

```
return                 ()                  =>                  {
  clearInterval(intervalId);
};
```

But where does `intervalId` come from? This value is returned by `setInterval()` and it is used as the argument for `clearInterval()`. Use `const intervalId =` to capture this value in a local variable when it is returned by our `setInterval()` call!

3.

Well, that seems to have solved our way-too-many-intervals-all-updating-the-same-variable bug!

Let’s create a new text input element so that the user can type a message while the timer is counting up.

- Add an `input` tag to our JSX whose `value` is managed by a current state variable called `name` with a state setter called `setName()`.

- Define an event handler named handleChange() and call that event handler with the input's onChange event listener attribute.

Checkpoint 4 Passed

Hint

Add each of the following following lines of code in appropriate places to our Timer component:

```
const handleChange = ({ target }) => setName(target.value);
<input value={name} onChange={handleChange} type='text' />
const [name, setName] = useState('');
```

4.

Uh oh. More bugs. Did you notice it yet? Type your full name in the text input field. See how the timer seems to stop counting while you are typing? That's not what we want!

What is going on here? We are creating a new interval after each render, that interval will call our state setter to update time exactly one second after each render. When we type in the input field, our component keeps re-rendering, cleaning up old intervals, and starting new ones… but our state setter never gets called until one second after we are done typing!

Let's fix this once and for all! We really want to use a single interval. We want that interval to start ticking away after our first render and we want it to be cleaned up after the final render. To accomplish this, use an empty dependency array!

Checkpoint 5 Passed

Hint

After our effect, pass a second argument to the Effect Hook. So that our effect is only called on the first render and our cleanup function is only called after our last render, pass an empty array as the second argument like so:

```
useEffect(effect, [])
```

Timer.js

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

export default function Timer() {
  const [time, setTime] = useState(0);
  const [name, setName] = useState("");

  useEffect(() => {
    const intervalId = setInterval(() => {
      setTime((prev) => prev + 1);
    }, 1000);

    return () => {
      clearInterval(intervalId);
    };
  }, []);

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

  return (
    <>
      <h1>Time: {time}</h1>
      <input value={name} onChange={handleChange} type='text
' />
    </>
  );
}
```

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. In this exercise, we'll learn to use the dependency array to further configure exactly when we want our effect to be called!

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!

Here's a nice example from the official React docs:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if the value stored by count changes
```

Instructions

1.
We've started building a weather planner app that will fetch data about the weather and allow our users to write some notes next to the forecast. A lot of good code has already been written, but there currently isn't anything rendering to the screen.

Let’s read through the code and start to wrap our heads around what is going on here. What part of our code do we think is keeping the component from rendering?

In our JSX, we are trying to map over an array stored by the `data` state variable, but our effect that fetches this data doesn’t get called until after the first render. So during the first render, data is `undefined` and attempting to call `map()` on `undefined` is causing our error!

Let’s prevent this error by checking to see if data has loaded yet. If it hasn’t, then we want our function component to just return a paragraph tag with the text “Loading…”. If data is no longer undefined, then the data has been loaded, and we can go ahead and render the full JSX!

Checkpoint 2 Passed

Hint

Just before the existing JSX, use an `if` block to check if data is falsey.

If it is, return the loading paragraph like so:

```
if (!data) {
  return <p>Loading...</p>;
}
```

2.

Our data fetching is being done in our effect. Notice how we are currently just using `alert()` messages to keep track of requesting and receiving data from our server. Instead of just stringifying the response data and showing it in an alert message, let’s store that data in our state.

When the data has been fetched, use our state setter function to store that data in our component’s state!

P.S. Remember that we want to store an array in our `data` state variable, not the whole response object.

Checkpoint 3 Passed

Hint

Calling the `setData()` function with the `data` property of the `response` object will store the array that we received from the server to our component’s local state. Call our state setter just after calling `alert()` with the response:

```
setData(response.data);
```

For more on asynchronous JavaScript, head over to this awesome lesson on JavaScript Promises.

3.

Type in each of the notes' input fields in our table. What do you notice? Why do you think this is happening?

Each time that we type in an input field, the component re-renders to show the new value of that field. Even though we don't need any new data from the backend, our component is fetching new data after every render!

Use an empty dependency array to ensure that data is only fetched after our component's first render.

Checkpoint 4 Passed

Hint

A small change in our code can make a big difference in our app's performance! Pass `[]` as the second argument to the `useEffect()` function.

```
useEffect(effect, []);
```

4.

Wow, that small code change made a huge difference in the performance of our weather planner app!

Let's make one more improvement. Did you notice the buttons at the top of our app? We want our users to be able to choose between planning around daily weather forecasts and weekly weather forecasts. Clicking on these buttons currently doesn't change anything. Let's fix that!

The server has two different endpoints called: `/daily` and `/hourly`. Let's use the value of the `forecastType` state variable to determine which endpoint our effect should request data from.

After making this change, our effect behaves differently based on the value of `forecastType`. You could say that how we use our effect depends on it! Add this variable to our dependency array so that the effect is called again, updating `data` appropriately, after re-renders where the user has selected a different forecast type.

Checkpoint 5 Passed

Hint

Add `forecastType` to the array that is passed as the second argument to `useEffect()`:

```
useEffect(effect, [forecastType]);
```

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

As we have been practicing with the State Hook and the Effect Hook, we've been following these rules with ease, but it is helpful to keep these two rules in mind as you take your new understanding of Hooks out into the wild and begin using more Hooks in your React applications.

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.

Instead of confusing React with code like this:

```
if (userName !== '') {
  useEffect(() => {
    localStorage.setItem('savedUserName', userName);
  });
}
```

We can accomplish the same goal, while consistently calling our Hook every time:

```
useEffect(() => {
  if (userName !== '') {
    localStorage.setItem('savedUserName', userName);
  }
});
```

Secondly, Hooks can only be used in React Functions. We cannot use Hooks in class components and we cannot use Hooks in regular JavaScript 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. For more on this topic, head to the React Docs.

Instructions

1.
The code that we are starting with has a lot of good ideas, but there are some bugs that we need to help sort out. Let's get started by refactoring the code so that the State Hook is always called at the top level.

It looks like the developers that wrote this code wanted to hold off on using the `selectedCategory` and `items` state variables until after the `categories` have been fetched. Conceptually this makes sense, but React requires that all hooks be called on every render, so nesting these `useState()` calls is not a valid option.

1. Let's bring all of our State Hook calls to the top level.
2. To be clear about initial values, let's explicitly set the initial state value for `categories` and `selectedCategory` to `null`.

Checkpoint 2 Passed
Hint
We don't need to check if `categories` is truthy before initializing our other state values. Delete the lines `if (categories) {` and `}` so that all of our State Hook calls are at the top level of the function component.

Call `useState()` with an argument of `null` to explicitly set its initial value as falsy instead of leaving it as `undefined`.
2.
It looks like the idea behind using this expression: `if (!categories)` was to only fetch the categories data from the server once. Nesting a call to the Effect Hook inside of a condition like this will cause different hooks to be called on different re-renders, resulting in errors. Luckily, we know a better way!

Refactor this code so that the effect responsible for fetching the `categories` data from the backend and saving it to local state follows the rules for Hooks and only fetches the categories data once.

Checkpoint 3 Passed

Hint

When using the Effect Hook, passing a dependency array as the second argument for `useEffect()` is the best way to determine when our effect is and is not called.

Remove the `if (!categories)` condition, and pass an empty dependency array so that this effect is only called after the first render.

3.

Whew, we're making great progress! It's such a nice feeling to turn error screens into working code, isn't it?

Now that we are fetching the list of categories from the backend and successfully rendering buttons for each of these to the screen, we are ready to use another effect to fetch the items for each of these categories, when the user clicks on each of them!

Uncomment the block of code that was attempting to do this, and refactor it so that we follow the rules of Hooks. To optimize performance, only call the backend for data when we don't yet have it stored in the component's state like this code was trying to do.

Checkpoint 4 Passed

Hint

The behavior of this effect depends on the values of two variables: `items` and `selectedCategory`. While we can't nest the whole `useEffect()` function call inside of the if block, this code…

```
if (selectedCategory && !items[selectedCategory]) { }
```

…is still very helpful to us. If `selectedCategory` and `!items[selectedCategory]` are both truthy, then we know that the user has clicked a button to see the items in a category that we don't yet have the data for, so we want to fetch them from the backend and store them in local state, otherwise, we don't need to fetch anything from the backend.

We already have most of the correct code, we just need to rearrange it a bit:

```
useEffect(() => {
    if (selectedCategory && !items[selectedCategory]) {
     /* fetch data and store it to local state */
    }
  }, [ /* list the two variables that this effect depends on here */]);
```

Shop.js

```
import React, { useState, useEffect } from 'react';
import { get } from './mockBackend/fetch';

export default function Shop() {
  const [categories, setCategories] = useState(null);
  const [selectedCategory, setSelectedCategory] = useState(null);
  const [items, setItems] = useState({});

  useEffect(() => {
    get('/categories').then((response) => {
      setCategories(response.data);
    });
  }, []);

  useEffect(() => {
    if (selectedCategory && !items[selectedCategory]) {
      get(`/items?category=${selectedCategory}`).then((response) => {
        setItems((prev) => ({ ...prev, [selectedCategory]: response.data }));
      });
    }
  }, [items, selectedCategory]);

  if (!categories) {
    return <p>Loading..</p>;
  }

  return (
    <div className='App'>
      <h1>Clothes 'n Things</h1>
      <nav>
        {categories.map((category) => (
          <button key={category} onClick={() => setSelectedC
ategory(category)}>
            {category}
          </button>
        ))}
      </nav>
      <h2>{selectedCategory}</h2>
      <ul>
        {!items[selectedCategory]
          ? null
          : items[selectedCategory].map((item) => <li key={i
tem}>{item}</li>)}
      </ul>
    </div>
  );
}
```

Separate Hooks for Separate Effects
When multiple values are closely related and change at the same time, it can make sense to group these values in a collection like an object or array. Packaging data together can also add complexity to the code responsible for managing that data. Therefore, it is a good idea to separate concerns by managing different data with different Hooks.

Compare the complexity here, where data is bundled up into a single object:

```
// Handle both position and menuItems with one useEffect hook.
const [data, setData] = useState({ position: { x: 0, y: 0
} });
useEffect(() => {
  get('/menu').then((response) => {
    setData((prev) => ({ ...prev, menuItems: response.data
}));
  });
  const handleMove = (event) =>
    setData((prev) => ({
      ...prev,
      position: { x: event.clientX, y: event.clientY }
    }));
  window.addEventListener('mousemove', handleMove);
  return () => window.removeEventListener('mousemove',
handleMove);
}, []);
```

To the simplicity here, where we have separated concerns:

```
// Handle menuItems with one useEffect hook.
const [menuItems, setMenuItems] = useState(null);
useEffect(() => {
  get('/menu').then((response) =>
setMenuItems(response.data));
}, []);

// Handle position with a separate useEffect hook.
const [position, setPosition] = useState({ x: 0, y: 0 });
useEffect(() => {
  const handleMove = (event) =>
    setPosition({ x: event.clientX, y: event.clientY });
  window.addEventListener('mousemove', handleMove);
  return () => window.removeEventListener('mousemove',
handleMove);
}, []);
```

It is not always obvious whether to bundle data together or separate it, but with practice, we get better at organizing our code so that it is easier to understand, add to, reuse, and test!

Instructions

1.
At the moment, this code seems to work just fine. There are three different network requests being made in a single effect and lots of different, unrelated data being managed in a single state variable. Let's get to work breaking these single `useState()` and `useEffect()` calls into separate, simpler hooks. Doing so will make this code easier to

understand, build on top of, and reuse as we continue to improve our application!

Begin refactoring this component:

- Use a separate State Hook for `menu`, `newsFeed`, and `friends`.
- Use these new state setters instead of `setData()` in the effect.
- Simplify our JSX to use these new state variables instead of `data`.

Checkpoint 2 Passed

Hint

Replace this:

```
const [data, setData] = useState(null);
```

With this:

```
const [menu, setMenu] = useState(null);
const [newsFeed, setNewsFeed] = useState(null);
const [friends, setFriends] = useState(null);
```

Replace the `setData()` call in the effect with this:

```
setMenu(menuResponse.data);
setNewsFeed(newsFeedResponse.data);
setFriends(friendsResponse.data);
```

And switch to using each of our new state variables in the JSX, for example:

- replace `!data || !data.menu` with `!menu`
- replace `data.menu.map()` with `menu.map()`

2.

`Promise.all()` was helpful to us when we had all of our data bundled up in a single object. It called all backend services, and when they all sent back responses, we could then call our state setters with the responses.

Because we are now managing these values separately, we can remove this complexity! In our effect, call the `get()` function three times for the three different data collections that our component wants to render, without using `Promise.all()` any more.

Checkpoint 3 Passed

Hint
These three endpoints are unrelated, so breaking them apart will help simplify our code!

Replace the `Promise.all()` function call with a separate `get()` and `then()` for each endpoint.

For example, our fetch of the menu data looks like this:

```
get('/menu').then((response) => {
  setMenu(response.data);
});
```

Use this same pattern to fetch data from the other two endpoints and call the appropriate state setter functions to store the response data in the local state.

3.

Now that we have three separate backend calls, let’s continue to separate concerns by splitting each of these into three separate Effect Hooks!

Checkpoint 4 Passed

Hint
Each API call is its own effect, so let’s place each in its own `useEffect()` call. For example, our Effect Hook for the menu data looks like this:

```
  useEffect(() => {
    get('/menu').then((response) => {
      setMenu(response.data);
    });
  }, []);
```

Use this same pattern to fetch data from the other two endpoints!

4.

Each `useEffect()` call is working with a corresponding `useState()` call. Let’s reorganize our code to show this relationship more clearly, making the logic easier to read and reuse!

For each of these three data collections, group the related State Hook and the Effect Hook next to one. This will help to make it clear which Hooks are working together to manage each separate data model.

Checkpoint 5 Passed

Hint
For example, our Hooks for the menu data looks like this:

```
const [menu, setMenu] = useState(null);
useEffect(() => {
  get('/menu').then((response) => {
    setMenu(response.data);
  });
}, []);
```

Use this same pattern to fetch data from the other two endpoints!

SocialNetwork.js

```
import React, { useState, useEffect } from 'react';
import { get } from './mockBackend/fetch';

export default function SocialNetwork() {
  const [menu, setMenu] = useState(null);
  useEffect(() => {
    get('/menu').then((response) => {
      setMenu(response.data);
    });
  }, []);

  const [newsFeed, setNewsFeed] = useState(null);
  useEffect(() => {
    get('/news-feed').then((response) => {
      setNewsFeed(response.data);
    });
   }, []);

  const [friends, setFriends] = useState(null);
  useEffect(() => {
    get('/friends').then(response) => {
      setFriends(response.data);
    });
   }, []);


  return (
    <div className='App'>
      <h1>My Network</h1>
      {!menu ? <p>Loading..</p> : (
        <nav>
          {menu.map((menuItem) => (
            <button key={menuItem}>{menuItem}</button>
          ))}
        </nav>
      )}
      <div className='content'>
        {!newsFeed ? <p>Loading..</p> : (
          <section>
            {newsFeed.map(({ id, title, message, imgSrc }) =
> (
              <article key={id}>
                <h3>{title}</h3>
                <p>{message}</p>
                <img src={imgSrc} alt='' />
              </article>
            ))}
          </section>
        )}
        {!friends ? <p>Loading..</p> : (
          <aside>
            <ul>
              {friends
                .sort((a, b) => (a.isOnline && !b.isOnline ?
 -1 : 0))
                .map(({ id, name, isOnline }) => (
                  <li key={id} className={isOnline ? 'online
' : 'offline'}>
                    {name}
                  </li>
                ))}
            </ul>
          </aside>
        )}
      </div>
    </div>
  );
}
```

Lesson Review
In this lesson, we learned how to write effects that manage timers, manipulate the DOM, and fetch data from a server. In earlier versions of React, we could only have executed this type of code in the lifecycle methods of class components, but with the Effect Hook, we can perform these types of actions in function components with ease!

Let's review the main concepts from this lesson:

- `useEffect()` - we can import this function from the 'react' library and call it in our function components
- *effect* - refers to a function that we pass as the first argument of the `useEffect()` function. By default, the Effect Hook calls this effect after each render
- *cleanup function* - the function that is optionally returned by the effect. If the effect does anything that needs to be cleaned up to prevent memory leaks, then the effect returns a cleanup function, then the Effect Hook will call this cleanup function before calling the effect again as well as when the component is being unmounted
- *dependency array* - this is the optional second argument that the `useEffect()` function can be called with in order to prevent repeatedly calling the effect when this is not needed. This array should consist of all variables that the effect depends on.

The Effect Hook is all about scheduling when our effect's code gets executed. We can use the dependency array to configure when our effect is called in the following ways:

Dependency Array	**Effect called after first render & …**
undefined	every re-render
Empty array	no re-renders
Non-empty array	when any value in the dependency array changes

Hooks gives us the flexibility to organize our code in different ways, grouping related data as well as separating concerns to keep code simple, error-free, reusable, and testable!

Instructions

Congratulations on finishing this lesson! With the new skills and information in this lesson, there's so much functionality that you can now add to the function components in your own React apps!

PageTitleFunction.js

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

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

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

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

PageTitleClass.js

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

export default class PageTitle extends Component {
  constructor(props) {
    super(props);
    this.state = {
      name: ''
    };
  }

  componentDidMount() {
    document.title = this.state.name;
  }

  componentDidUpdate() {
    document.title == `Hi, ${this.state.name}`;
  }

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