useEffect

Prior Knowledge

- Be able to manage React state.
- Be able to plan and implement React apps.

Learning Objectives

- Understand how to perform side effects in React using the `useEffect` hook.
- Understand how to control when effects are ran using `useEffect dependencies`.
- Understand loading patterns in React.
- Understand error patterns in React.

useEffect hook

When writing React apps we will need to perform some side-effects. These include things like contacting apis to retrieve data, subscribing to services or manipulating the DOM (attaching a global listener for example.)

These actions need to be performed at certain times and not at others. If we think about a GET request to /api/users to list all of the users of a site, this would need to happen once on the page load.

React hooks allow us to `hook` into React's lifecycle and tell React when to run these side-effects.

To do so we use the `useEffect` hook. `useEffect` takes 2 arguments:

0. The effect to run - A `function` containing the logic of our side-effect.
1. Dependencies - An `array of variables`. When any variable in the array changes value, the effect will be run.

```
useEffect(() => {}, []);
// () => {} - effect
// [] - dependencies
```

If we think about the users example from above, we want to make a request to the api as soon as the component is first rendered (this is referred to as `mounting` the component). We could implement that functionality as below:

```
import { useEffect } from 'react';
// a util function that deals with the http logic of contacting the api
import { fetchUsers } from '../utils/api.js'

const Users = () => {
  const [users, setUsers] = useState([]);

  // fetch the users data when the component is first rendered
  useEffect(() => {
    fetchUsers().then((usersFromApi) => {
      setUsers(usersFromApi);
    });
  }, []);

  return <ul>
    {users.map((user) => {
    //  render each user
    })
};
```

The effect here is to invoke `fetchUsers` which will contact our api and return a promise, resolving to an array of users. Then to set those users in the component's state.

There are no dependencies for this effect so the array is blank. This effectively means that the request will happen the first time the component is rendered, and then never again.

Caution The dependency array is optional. If it is omitted the effect will be ran on every render of the component. However the effect includes setting the state, which will cause a re-render. This next render will then run the effect again, causing another render, and so on in an infinite loop.

Make sure that you are running your effects at the right time using appropriate dependencies.

```
// Infinite loop :(
useEffect(() => {
  fetchUsers().then((usersFromApi) => {
    setUsers(usersFromApi);
  });
});
```

useEffect dependencies

The second argument to `useEffect` is a list of dependencies. A dependency is a value for React to check to see if the effect should run.

Each time the component is rendered, React will compare *each value* in the dependency array to the value it contained on the previous render. If any of those values are different, the effect will be run. If they are all the same, the effect will not be run.

For example, if we were building a page to display information about a user's profile and had an endpoint of GET `/api/users/:user_id` to retrieve their profile information, we would need an effect to fetch that data.

Our effect would look similar to the above example but now we have a dependency.

```
import { useEffect } from 'react';
// a util function that deals with the http logic of contacting the api
import { fetchUserById } from '../utils/api.js'

const UserProfile = ({ userId }) => {
  const [user, setUser] = useState({});

  useEffect(() => {
    fetchUserById(userId).then((userFromApi) => {
      setUser(userFromApi);
    });
  }, [userId]);

  return <section>
    <h2>{user.username}</h2>
    //  render the users profile
};
```

The component takes the `userId` as a prop and will display the information for that user. As this prop can change we don't just want to fetch the user's profile on the first render, but also any time that the id changes.

By adding it to our array, React will compare the current `userId` to the `userId` from the **previous** render. If the values are different the effect will be re-run and the correct users data will be fetched.

nb The `userId` being a prop is unrelated to how `useEffect` works. Props are very common dependencies but a dependency can be any value, it could have been a state value from this component or any other value and it would work the same way.

Cleanup effects

Sometimes we will be writing side effects that require some cleanup. Consider a messaging app in which our users can open and close a chat. When that chat is opened our app will need to subscribe to that chat and start listening to new posted messages.

A pseudo-code implementation might look like the following:

```
const Chat = () => {
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    // every time a message is posted, add it to state
    subscribeToNewMessages().on('newMessage', (newMessage) => {
      setMessages((currMsgs) => [...currMsgs, newMessage]);
    });
    // subscribe when this component is mounted
  }, []);

  return <ul>
    // list of messages in the chat
};
```

Here we subscribe to our new messages when the component is mounted and write the logic for adding a new message to our state. At the moment, we never unsubscribe from these messages however. If the user closes our Chat and removes the component, we are still trying to listen.

This is where we need to perform some cleanup. We can return a function from useEffect that will be *run just before the component is removed (unmounted)* from the page. Inside this function we will write our cleanup logic, in this case, unsubscribing from messages.

```
useEffect(() => {
  subscribeToNewMessages().on('newMessage', (newMessage) => {
    setMessages((currMsgs) => [...currMsgs, newMessage]);
  });
  // a cleanup function to be ran before unmount
  return () => {
    // logic to unsub from messages
    unSubscribeFromMessages();
  };
}, []);
```

Loading patterns

When dealing with http requests, they will be asynchronous and will take some time to complete. Depending on the speed of the user's connection and the amount of traffic going through our servers, these requests may take a noticeable amount of time to complete.

A common practice is to give users an indication that the page is loading whilst waiting for these requests to complete so they are aware that something is happening and they should wait for it to finish.

Consider the users example from before. When our component first mounts, there is no data to display so there will just be a blank page until the request finishes. We can track whether the data is loaded or not in state and update our UI to give the user some meaningful feedback to let them know we're still loading data.

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

const Users = () => {
  const [users, setUsers] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

useEffect(() => {
    fetchUsers()
    .then((usersFromApi) => {
      setUsers(usersFromApi);
      setIsLoading(false)
    });
  }, []);

  if (isLoading) return <p>Loading...</p>
  return <ul>
    {users.map((user) => {
      //  render each user
    })}
  </ul>
};
```

Here we keep a boolean in state to keep track of whether or not the users have been loaded yet. If they are still loading we can render a different UI, in this case just a `p` tag. This could be a dedicated `Loading` component that shows a nice loading spinner or similar however. Once the data has been received and set in our state, we set loading to false and render the list of users.

Rendering different JSX depending on some information in our components is a technique called conditional rendering.

The React docs have some lovely examples of different ways of implementing this and when it would be useful.

See useEffect for more info on the `useEffect` hook.