Custom React Hooks

Prior Knowledge

- Understand the naming conventions of hooks.
- Be able to write component logic with `useState` and `useEffect`.

Learning Objectives

- Understand how to use custom hooks written by 3rd party components.
- Understand how the rules of hooks must be followed when writing custom hooks.
- Be able to write a hook that uses React's in-built hooks.

Custom Hooks

In React we have been using some in built `hooks` to add functionality to our components. React also supports writing our own custom hooks to extract component logic into re-useable functions.

In React, a custom hook is a function

1. Whose name starts with "use".
2. May call other hooks within itself.

When we are calling hooks from within our components we must follow the rules of hooks and we must do the same with our custom hooks.

Custom hooks work exactly like components, the major difference being that they are not responsible for **rendering** data. Where our components will usually return some `JSX` (or potentially null) our custom hooks can return data in any form we would like.

Consider a component that renders a user's profile from an api.

```
import { fetchProfileById } from '../utils/api';

const UserProfile = (userId) => {
  const [profile, setProfile] = useState({});

  useEffect(() => {
    fetchProfileById(userId).then((profileFromApi) => {
      setProfile(profileFromApi);
    });
  }, [userId]);

  return (
    <div>
      <h2>{profile.username}</h2>
      // etc
    </div>
  );
};
```

We can extract the logic of contacting the api to a custom hook and call that hook from our component.

```
// src/hooks/useProfile.js
export const useProfile = (userId) => {
  const [profile, setProfile] = useState({});

  useEffect(() => {
    fetchProfileById(userId).then((profileFromApi) => {
      setProfile(profileFromApi);
    });
  }, [userId]);

  return profile;
};
```

The logic of keeping the `profile` in state and the side effect of fetching the data is still the same as it was before but it is extracted into our hook.

```
const UserProfile = (userId) => {
  const profile = useProfile(userId);

  return (
    <div>
      <h2>{profile.username}</h2>
      // etc
    </div>
  );
};
```

In this case we are returning the profile. We can return any data structure we like however. For example we could expand the hook to keep track of whether or not the data has loaded and return an object with both the profile and the loading state.

```
export const useProfile = (userId) => {
  const [profile, setProfile] = useState({});
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    setIsLoading(true);
    fetchProfileById(userId).then((profileFromApi) => {
      setProfile(profileFromApi);
      setIsLoading(false);
    });
  }, [userId]);

  return { profile, isLoading };
};
```

This hook is re-useable by any component and can be called multiple times. Each invocation of the hook will get it's own state and effects just as if we had written the logic multiple times. The hooks allow us to re-use stateful logic. To share state we should use another mechanism such as Context or a composed component.

The React docs have an excellent guide on writing your own hooks and here a list of examples of custom hooks that someone has kindly shared in a Medium post.