React Side-Effects

Next we'll add a feature to our Search component in the form of another React hook. We'll make the Search component remember the most recent search interaction, so the application opens it in the browser whenever it restarts.

First, use the local storage of the browser to store the `searchTerm` accompanied by an identifier. Next, use the stored value, if there a value exists, to set the initial state of the `searchTerm`. Otherwise, the initial state defaults to our initial state (here "React") as before:

```
const App = () => {
 ...

 const [searchTerm, setSearchTerm] = React.useState(

   localStorage.getItem('search') || 'React'

 );

 const handleSearch = event => {
   setSearchTerm(event.target.value);

   localStorage.setItem('search', event.target.value);

 };

 ...
);
~~
```

src/App.js

When using the input field and refreshing the browser tab, the browser should remember the latest search term. Using the local storage in React can be seen as a **side-effect** because we interact outside of React's domain by using the browser's API.

There is one flaw, though. The handler function should mostly be concerned about updating the state, but now it has a side-effect. If we use the

`setSearchTerm` function elsewhere in our application, we will break the feature we implemented because we can't be sure the local storage will also get updated. Let's fix this by handling the side-effect at a dedicated place. We'll use **React's useEffect Hook** to trigger the side-effect each time the `searchTerm` changes:

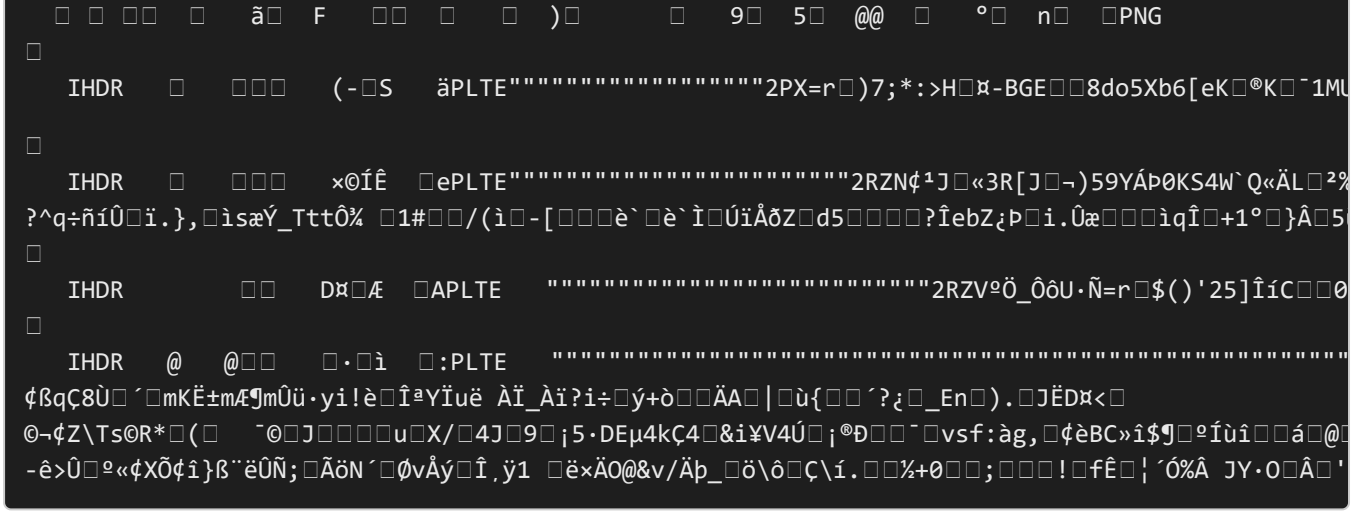
```
const App = () => {
 ...

 const [searchTerm, setSearchTerm] = React.useState(
   localStorage.getItem('search') || 'React'
 );

 React.useEffect(() => {
   localStorage.setItem('search', searchTerm);
 }, [searchTerm]);

 const handleSearch = event => {
   setSearchTerm(event.target.value);
 };


 ...
);
```

src/App.js

React's useEffect Hook takes two arguments: The first argument is a function where the side-effect occurs. In our case, the side-effect is when the user types the `searchTerm` into the browser's local storage. The second argument is a dependency array of variables. If one variable changes, the function for the side-effect is called. In our case, the function is called every time the `searchTerm` changes; it's called initially when the component renders for the first time.

If the dependency array of React's useEffect is an empty array, the function for the side-effect is only called once, after the component renders for the first time. The hook lets us opt into React's component lifecycle. It can be triggered when the component is first mounted, but also one of its dependencies are updated.

Using React `useEffect` instead of managing the side-effect in the handler has made the application more robust. *Whenever* and *wherever* `searchTerm` is updated via `setSearchTerm`, local storage will always be in sync with it.



Exercises:

- Confirm the changes from the last section.
- Read more about React's useEffect Hook (0, 1).
- Give the first argument's function a `console.log()` and experiment with React's useEffect Hook's dependency array. Check the logs for an empty dependency array too.