React Custom Hooks (Advanced)

Learn to build the hook yourself.

We'll cover the following

- Exercises:

Thus far we've covered the two most popular hooks in React: `useState` and `useEffect`. `useState` is used to make your application interactive; `useEffect` is used to opt into the lifecycle of your components.

We'll eventually cover more hooks that come with React – though we won't cover all of them here. Next, we'll tackle **React custom Hooks**; that is, building a hook yourself.

We will use the two hooks we already possess to create a new custom hook called `useSemiPersistentState`, named as such because it manages state yet synchronizes with the local storage. It's not fully persistent because clearing the local storage of the browser deletes relevant data for this application. Start by extracting all relevant implementation details from the App component into this new custom hook:

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

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


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

src/App.js

So far, it's just a function around our previously used in the `App` component used

`useState` and `useEffect` hooks. Before we can use it, let's return the values that are needed in our App component from this custom hook:

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

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


 return [searchTerm, setSearchTerm];

};
```

src/App.js

We are following two conventions of React's built-in hooks here. First, the naming convention which puts the "use" prefix in front of every hook name; second, the returned values are returned as an array. Now we can use the custom hook with its returned values in the App component with the usual array destructuring:

```
const App = () => {
 const stories = [ ... ];

 const [searchTerm, setSearchTerm] = useSemiPersistentState();

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

 const searchedStories = stories.filter(story =>
   story.title.toLowerCase().includes(searchTerm.toLowerCase())
 );

 return (
   ...
 );
};
```

src/App.js

Another goal of a custom hook should be reusability. All of this custom hook's internals are about the search domain, but the hook should be for a value that's set in state and synchronized in local storage. Let's adjust the naming therefore:

```
const useSemiPersistentState = () => {
 const [value, setValue] = React.useState(
   localStorage.getItem('value') || ''
 );
```

```
 React.useEffect(() => {

   localStorage.setItem('value', value);

 }, [value]);

 return [value, setValue];

};
```

src/App.js

We handle an abstracted “value” within the custom hook. Using it in the App component, we can name the returned current state and state updater function anything domain-related (e.g. `searchTerm` and `setSearchTerm`) with array destructuring.

There is still one problem with this custom hook. Using the custom hook more than once in a React application leads to an overwrite of the “value”-allocated item in the local storage. To fix this, pass in a key:

```
const useSemiPersistentState = key => {

 const [value, setValue] = React.useState(

   localStorage.getItem(key) || ''
 );

 React.useEffect(() => {

   localStorage.setItem(key, value);
 }, [value, key]);


 return [value, setValue];
};

const App = () => {
 ...

 const [searchTerm, setSearchTerm] = useSemiPersistentState(

   'search'
 );

 ...
};
```

src/App.js

Since the key comes from outside, the custom hook assumes that it could change, so it needs to be included in the dependency array of the `useEffect` hook. Without it, the side-effect may run with an outdated key (also called *stale*) if the key

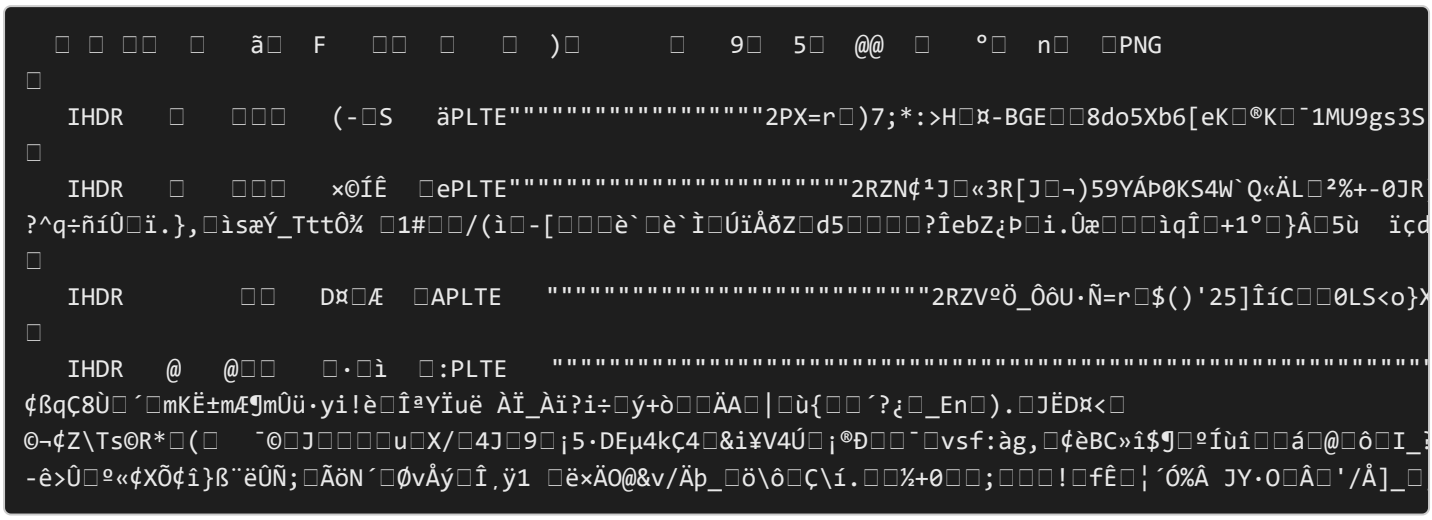
changed between renders.

Another improvement is to give the custom hook the initial state we had from the outside:

```
const useSemiPersistentState = (key, initialState) => {

 const [value, setValue] = React.useState(

   localStorage.getItem(key) || initialState

 );

 ...
};

const App = () => {
 ...

 const [searchTerm, setSearchTerm] = useSemiPersistentState(

   'search',
   'React'

 );

 ...
};
```

src/App.js

Here is the complete demonstration of the above concepts:



You've just created your first custom hook. If you're not comfortable with custom hooks, you can revert the changes and use the `useState` and `useEffect` hook as before, in the App component.

However, knowing more about custom hooks gives you lots of new options. A custom hook can encapsulate non-trivial implementation details that should be

kept away from a component; can be used in more than one React component, and

can even be open-sourced as an external library. Using your favorite search engine, you'll notice there are hundreds of React hooks that could be used in your application without worry over implementation details.

Exercises:

- Confirm the changes from the last section.
- Read more about React Hooks to get a good understanding of them. They are the bread and butter in React function components, so it's important to really understand them (0, 1).