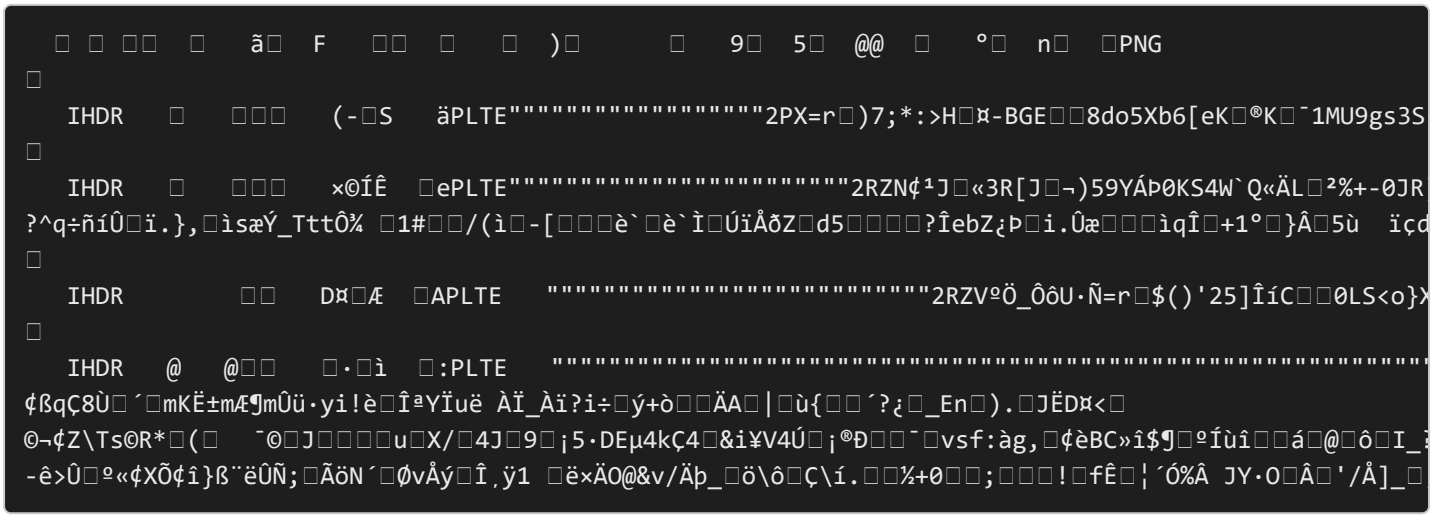
React Controlled Components

Learn how to turn the Search component and its input field into a controlled component.

We'll cover the following

- Exercises:

Controlled components are not necessarily React components, but HTML elements. Let's learn how to turn the `Search` component and its input field into a controlled component. Let's go through a scenario that shows why we should follow the concept of controlled components throughout our React application. After applying the following change – giving the `searchTerm` an initial state – can you spot the mistake in your browser?



While the list has been filtered according to the initial search, the input field doesn't show the initial `searchTerm`. We want the input field to reflect the actual `searchTerm` used from the initial state; but it's only reflected through the filtered list.

We need to convert the Search component with its input field into a controlled component. So far, the input field doesn't know anything about the `searchTerm`. It only uses the change event to inform us of a change. Actually, the input field has a `value` attribute.

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

```
 const [searchTerm, setSearchTerm] = React.useState('React');

 ...

 return (
   <div>
     <h1>My Hacker Stories</h1>

     <Search search={searchTerm} onSearch={handleSearch} />
     ...
   </div>
 );
};

const Search = props => (
 <div>
   <label htmlFor="search">Search: </label>
   <input
     id="search"
     type="text"

     value={props.search}
     onChange={props.onSearch}
   />
 </div>
);
```

src/App.js

Now the input field starts with the correct initial value, using the `searchTerm` from the React state. Also, when we change the `searchTerm`, we force the input field to use the value from React's state (via props). Before, the input field managed its own internal state natively with just HTML.

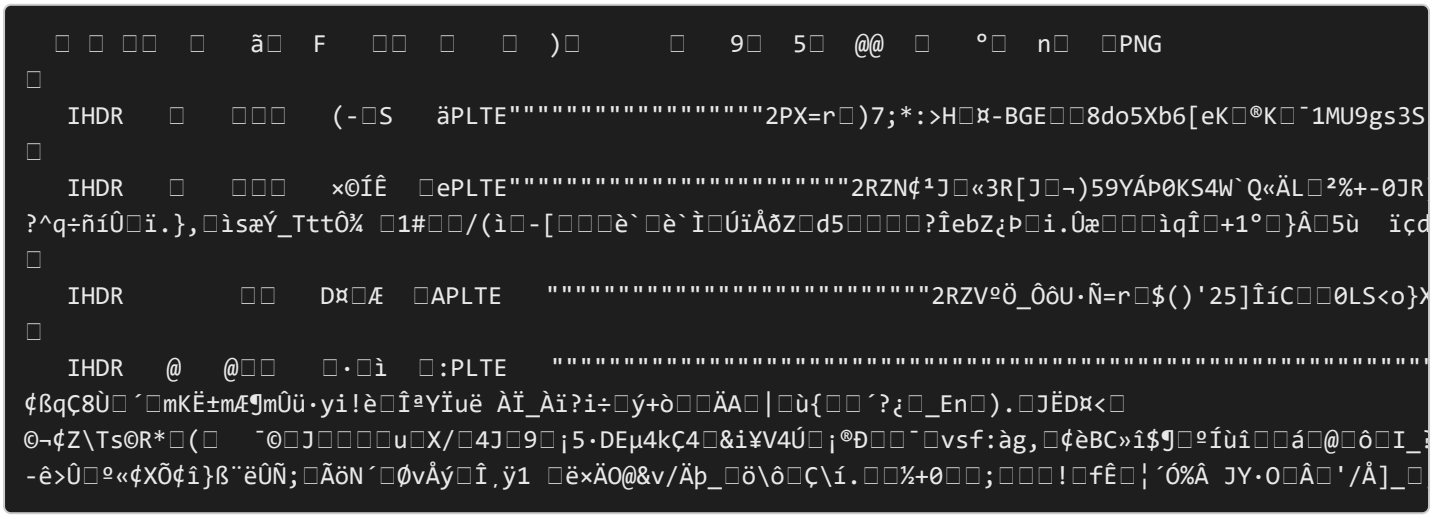
We learned about controlled components in this section, and, taking all the previous sections as learning steps into consideration, discovered another concept called **unidirectional data flow**:

UI -> Side-Effect -> State -> UI -> ...

A React application and its components start with an initial state, which may be passed down as props to other components. It's rendered for the first time as a UI. Once a side-effect occurs, like user input or data loading from a remote API, the change is captured in React's state. Once state has been changed, all the components affected by the modified state or the implicitly modified props are re-rendered (the component functions runs again).

In the previous sections, we also learned about React's **component lifecycle**. At first, all components are instantiated from the top to the bottom of the component hierarchy. This includes all hooks (e.g. `useState`) that are instantiated with their initial values (e.g. initial state). From there, the UI awaits side-effects like user interactions. Once state is changed (e.g. current state changed via state updater function from `useState`), all components affected by modified state/props render again.

Every run through a component's function takes the *recent value* (e.g. current state) from the hooks and *doesn't* reinitialize them again (e.g. initial state). This might seem odd, as one could assume the `useState` hooks function re-initializes again with its initial value, but it doesn't. Hooks initialize only once when the component renders for the first time, after which React tracks them internally with their most recent values.



Exercises:

- Confirm the changes from the last section.
- Read more about controlled components in React.
- Experiment with `console.log()` in your React components and observe how your changes render, both initially and after the input field changes.