React Impossible States

Learn how conditional states can lead to impossible states and undesired behavior in the UI.

We'll cover the following

- Exercises:

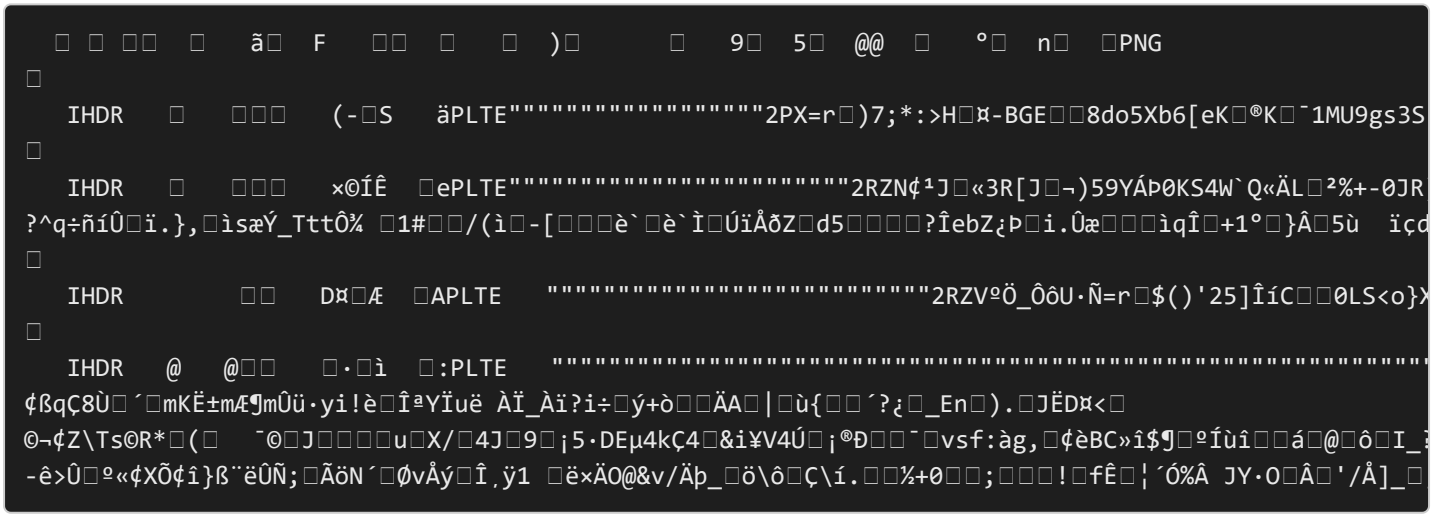
Perhaps you've noticed a disconnect between the single states in the App component, which seem to belong together because of the `useState` hooks. Technically, all the states related to the asynchronous data belong together, which doesn't only include the stories as actual data, but also their loading and error states.

There is nothing wrong with multiple `useState` hooks in one React component. Be wary once you see multiple state updater functions in a row, however. These conditional states can lead to **impossible states**, and undesired behavior in the UI. Try changing your pseudo data fetching function to the following to simulate the error handling:

```
const getAsyncStories = () =>
  new Promise((resolve, reject) => setTimeout(reject, 2000));
```

src/App.js

Practice this right here:



The impossible state happens when an error occurs for the asynchronous data. The state for the error is set, but the state for the loading indicator isn't revoked. In the UI, this would lead to an infinite loading indicator and an error message, though it may be better to show the error message only and hide the loading indicator. Impossible states are not easy to spot, which makes them infamous for causing bugs in the UI.

Fortunately, we can improve our chances by moving states that belong together from multiple `useState` and `useReducer` hooks into a single `useReducer` hook. Take the following `useState` hooks:

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

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    []
  );
  const [isLoading, setIsLoading] = React.useState(false);
  const [isError, setIsError] = React.useState(false);

  ...
};
```

src/App.js

Merge them into one `useReducer` hook for a unified state management and a more complex state object:

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

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    { data: [], isLoading: false, isError: false }
  );

  ...
};
```

src/App.js

Everything related to asynchronous data fetching must use the new dispatch function for state transitions:

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

  const [stories, dispatchStories] = React.useReducer(
```

```
    storiesReducer,
    { data: [], isLoading: false, isError: false }
  );


  React.useEffect(() => {

    dispatchStories({ type: 'STORIES_FETCH_INIT' });
    getAsyncStories()
      .then(result => {
        dispatchStories({

          type: 'STORIES_FETCH_SUCCESS',

          payload: result.data.stories,
        });
      })
      .catch(() =>

        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })

      );
  }, []);

  ...
};
```

src/App.js

Since we introduced new types for state transitions, we must handle them in the `storiesReducer` reducer function:

```
const storiesReducer = (state, action) => {
  switch (action.type) {

    case 'STORIES_FETCH_INIT':
      return {
        ...state,
        isLoading: true,
        isError: false,
      };
    case 'STORIES_FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,
        data: action.payload,
      };
    case 'STORIES_FETCH_FAILURE':
      return {
        ...state,
        isLoading: false,
        isError: true,
      };
    case 'REMOVE_STORY':
      return {
        ...state,
        data: state.data.filter(
          story => action.payload.objectID !== story.objectID
        ),
```

```
      };

    default:
      throw new Error();
  }
};
```

src/App.js

For every state transition, we return a *new state* object which contains all the key/value pairs from the *current state* object (via JavaScript's spread operator) and the new overwriting properties. For instance, `STORIES_FETCH_FAILURE` resets the `isLoading`, but sets the `isError` boolean flags yet keeps all the other state instact (e.g. `stories`). That's how we get around the bug introduced earlier since an error should remove the loading state.

Observe how the `REMOVE_STORY` action changed as well. It operates on the `state.data` , no longer just the plain `state`. The state is a complex object with data, loading and error states rather than just a list of stories. This has to be solved in the remaining code too:

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

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    { data: [], isLoading: false, isError: false }
  );

  ...

  const searchedStories = stories.data.filter(story =>

    story.title.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      ...

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List
          list={searchedStories}
          onRemoveItem={handleRemoveStory}
        />
      )}
    </div>
  );
};
```

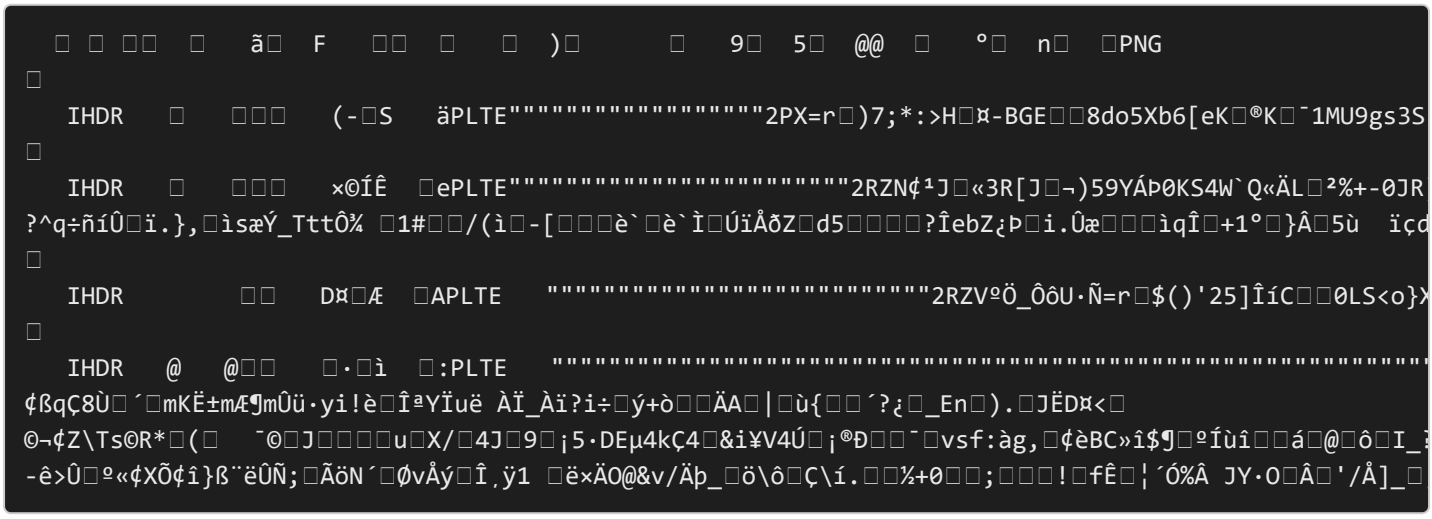
src/App.js

Try to use the erroneous data fetching function again and check whether everything works as expected now:

```
const getAsyncStories = () =>
  new Promise((resolve, reject) => setTimeout(reject, 2000));
```

src/App.js

We moved from unreliable state transitions with multiple `useState` hooks to predictable state transitions with React's useReducer Hook. The state object managed by the reducer encapsulates everything related to the stories, including loading and error state, but also implementation details like removing a story from the list of stories. We didn't get fully rid of impossible states, because it's still possible to leave out a crucial boolean flag like before, but we moved one step closer towards more predictable state management.



Exercises:

- Confirm the changes from the last section.
- Read over the previously linked tutorials about reducers in JavaScript and React.
- Read more about when to use useState or useReducer in React.