React Advanced State

All state management in this application makes heavy use of React's useState Hook. More sophisticated state management gives you **React's useReducer Hook**, though. Since the concept of reducers in JavaScript splits the community in half, we won't cover it extensively here, but the exercises at the end of this section should give you plenty of practice.

We'll move the `stories` state management from the `useState` hook to a new `useReducer` hook. First, introduce a reducer function outside of your components. A reducer function always receives `state` and `action`. Based on these two arguments, a reducer always returns a new state:

```
const storiesReducer = (state, action) => {
  if (action.type === 'SET_STORIES') {
    return action.payload;
  } else {
    throw new Error();
  }
};
```

src/App.js

A reducer `action` is often associated with a `type`. If this type matches a condition in the reducer, do something. If it isn't covered by the reducer, throw an error to remind yourself the implementation isn't covered. The `storiesReducer` function covers one `type`, and then returns the `payload` of the incoming action without using the current state to compute the new state. The new state is simply the `payload`.

In the App component, exchange `useState` for `useReducer` for managing the `stories`. The new hook receives a reducer function and an initial state as arguments and returns an array with two items. The first item is the *current state*; the second item is the *state updater function* (also called *dispatch function*):

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

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    []
  );

  ...
};
```

src/App.js

The new dispatch function can be used instead of the `setStories` function, which was returned from `useState`. Instead of setting state explicitly with the state updater function from `useState`, the `useReducer` state updater function dispatches an action for the reducer. The action comes with a `type` and an optional payload:

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

  React.useEffect(() => {
    setIsLoading(true);

    getAsyncStories()
      .then(result => {
        dispatchStories({
          type: 'SET_STORIES',
          payload: result.data.stories,
        });

        setIsLoading(false);
      })
      .catch(() => setIsError(true));
  }, []);

  ...

  const handleRemoveStory = item => {
    const newStories = stories.filter(
      story => item.objectID !== story.objectID
    );

    dispatchStories({
      type: 'SET_STORIES',
      payload: newStories,
    });
  };

  ...
};
```

src/App.js

The application appears the same in the browser, though a reducer and React's `useReducer` hook are managing the state for the stories now. Let's bring the concept of a reducer to a minimal version by handling more than one state transition.

So far, the `handleRemoveStory` handler computes the new stories. It's valid to move this logic into the reducer function and manage the reducer with an action, which is another case for moving from imperative to declarative programming. Instead of doing it ourselves by saying *how it should be done*, we are telling the reducer *what to do*. Everything else is hidden in the reducer.

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

  const handleRemoveStory = item => {
    dispatchStories({
      type: 'REMOVE_STORY',
      payload: item,
    });
  };

  ...
};
```

src/App.js

Now the reducer function has to cover this new case in a new conditional state transition. If the condition for removing a story is met, the reducer has all the implementation details needed to remove the story. The action gives all the necessary information, an item's identifier to remove the story from the current state and return a new list of filtered stories as state.

```
const storiesReducer = (state, action) => {
  if (action.type === 'SET_STORIES') {
    return action.payload;

  } else if (action.type === 'REMOVE_STORY') {
    return state.filter(
      story => action.payload.objectID !== story.objectID
    );
  } else {

    throw new Error();
  }
```

```
};
```

src/App.js

All these if else statements will eventually clutter when adding more state transitions into one reducer function. Refactoring it to a switch statement for all the state transitions makes it more readable:

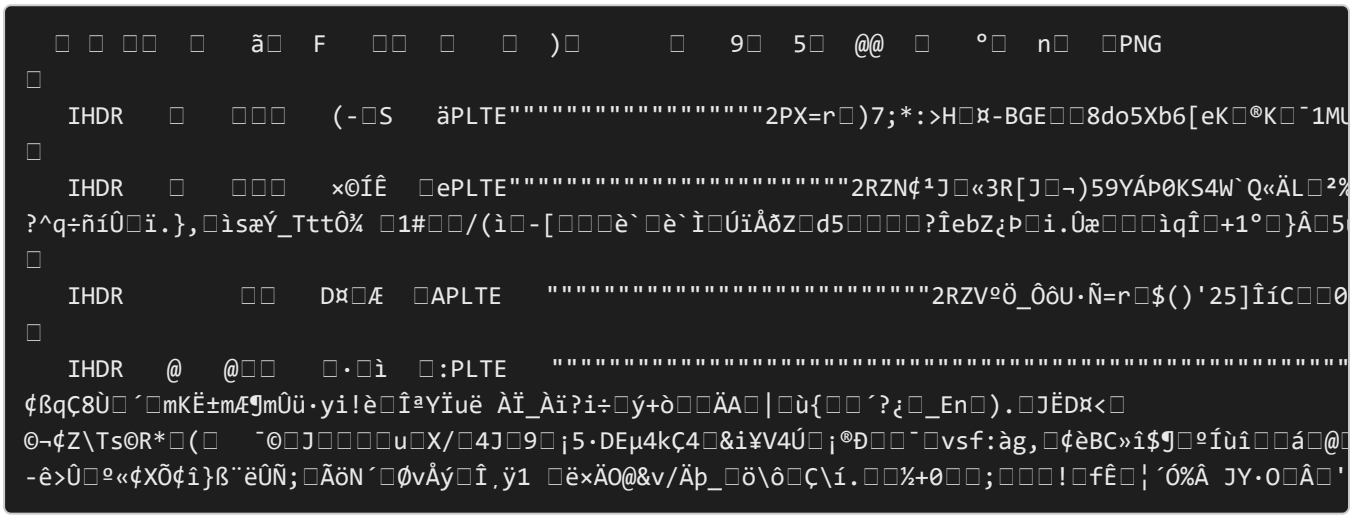
```
const storiesReducer = (state, action) => {

  switch (action.type) {
    case 'SET_STORIES':
      return action.payload;
    case 'REMOVE_STORY':
      return state.filter(
        story => action.payload.objectID !== story.objectID
      );
    default:
      throw new Error();
  }

};
```

src/App.js

What we've covered is a minimal version of a reducer in JavaScript. It covers two state transitions, shows how to compute current state and action into a new state, and uses some business logic (removal of a story). Now we can set a list of stories as state for the asynchronously arriving data, and remove a story from the list of stories, with just one state managing reducer and its associated `useReducer` hook.



To fully grasp the concept of reducers in JavaScript and the usage of React's useReducer Hook, visit the linked resources in the exercises.

Exercises:

- Confirm the changes from the last section.
- Read more about reducers in JavaScript.
- Read more about reducers and useReducer in React (0, 1).