React so far

- React function-based components
- state-per-component from `useState` hook
- passing state as props
- altering state in children via callback props
- per-render/init effects from `useEffect` hooks
- changing css classes via state/props for non-structural visual changes

Complex state

`useState` is normally fine

- What if you have multiple state flags that could change at the same time?
- What if your complex state changes based on previous state?

Answer: `useReducer` hook

State as an object

Imagine our todo state as a single object

```
const todoState = {
  isLoading: false,
  isLoggedIn: true,
  username: 'cat',
  todos: {
    asdf: {
      id: 'asdf',
      task: 'Nap',
      done: false,
    },
    hjkl: {
      id: 'hjkl',
      task: 'Knock things off shelves',
      done: true,
    },
  },
};
```

Pros and Cons

- Changes can be made atomically
    - one setter call
    - no risk of partial re-render
- Easy to pass around
    - can pass all as prop or parts as props
- Will trigger rerender of most everything if anything changes
    - but that's mostly true anyway

Actions on the state

With state as a single object

- you can perform actions on the state
- named actions
    - "login", "logout", "toggleTodo", etc
- these actions can be code themselves

```
function logout(state) {
  return {
    ...state,
    IsLoggedIn: false,
    username: '',
    todos: {},
  };
}
```

Many action functions

- each takes state
    - any any params needed for new state
- each returns a new state object

Notice that we aren't CHANGING the state object

- we return a NEW one
- avoids side-effects

A reducer combines these action types

All those action functions are the same pattern:

- accept state
- accept any necessary params
- return new state

You can make one function

- also pass it the action "type"
- it can `switch` that type
- and return the new state

Reducer example

```
function reducer( state, action ) {
  switch(action.type) {
    case 'logout':
      return { ...state, isLoggedIn: false, username: '', todos: {} };
    case 'login':
      return { ...state, isLoggedIn: true, username: action.username };
    case 'toggleTodo':
      return {
        ...state,
        todos: {
          ...state.todos,
          [action.id]: {
            ...state.todos[action.id],
            done: !state.todos[action.id].done,
          }
        },
      };
    default:
      return state;
  }
}
```

A lot there

- but the concept is simple
    - pass the current state
    - pass an action object (below is example)
        - action.type is the name of the action
        - action.(anything else) are params for that action
    - return a new state object
        - often filled with the old values
        - except for parts that change
- Notice there is NO JSX, no React
    - just bland JS

Dispatch function uses the reducer

Imagine a function

- React aware
- knows the current state
- knows the setter for current state
- is passed the action object
- calls the reducer
    - passing state
    - passing action object
- sets the new state to result

useReducer hook

`useReducer(reducer, initialArg);`

- `initialArg` is the initial state
- returns `[ state, dispatch ]`
  - `state` is the current state
  - `dispatch` is the `dispatcher` function

Updates the state (and triggers any re-renders):

- `dispatch({ type:'setTheme', theme:'dark' });`
- You can pass `dispatch` as a prop to descendants
- They can dispatch actions without other callbacks

React Example

Assume `initState` and `reducer` are imported:

```
const App = () => {
  const [state, dispatch] = useReducer(reducer, initState);
  const setTheme = (e) => dispatch({
    type: 'setTheme',
    theme: e.target.value
  });
  return (
    <div className={state.theme}>
      <select value={state.theme} onChange={setTheme}/>
        <option value="light">Light</option>
        <option value="dark">Dark</option>
      </select>
    </div>
  );
};
```

When to useReducer?

`useState` is **not wrong**

use `useReducer` when you:

- need to change many related state values
- have complex state changing logic
    - such as state changing based on state
- state-changing logic that you want
    - to reuse
    - to have testable outside of components

Summary - reducer

A reducer function

- takes the current state and an action
- returns a new state action
- is a pure JS function
    - no react
    - no JSX
    - no outside values
- can be written in a .js file
    - and imported

Summary - dispatcher

Dispatcher function

- is passed the action object
- updates the app state

Summary - useReducer

- Hook takes initial state and reducer
- returns state and a dispatch function

Dispatch function

- can be passed to children
- or wrapped and that wrapper passed to children
    - so children can only "dispatch" certain actions

Summary - when to use a reducer

- useState is perfectly valid
- useReducer when
    - complex state (or part of state)
    - want atomic changes to different parts of state

(Internally, useState is just a simple useReducer!)