# 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
- Structural changes w/conditional rendering
- Non-structural changes w/CSS classes

# Complex application state

`useState` is normally fine, but state can be complex

- Complex state changes multiple fields at once
- Creates chances for bugs
    - Easy to overlook a field in one place
- A set of changes can have multiple triggers

Answer: `useReducer` hook

**My view: Most important abstraction in webdev**

# 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 large rerender if anything changes
    - But that's mostly true anyway
    - React will only change DOM when needed

# Actions on the state

With state as a single object

- Can perform named actions on the state
    - "login", "logout", "toggleTodo", etc
    - Named for the **event happening to the state**
        - NOT the page it is happening on
- These actions can be code themselves

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

# Many action functions

- Each takes state
    - And 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
- Also don't mutate any state values!

# 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

- Pass state + action "type" (name)
- It can `switch` that type
- Return the new state

# Reducer Example (simplified)

```
function reducer( state, action ) {
  switch(action.type) {
    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 concept isn't as complex as it seems:
    - Pass the **current state**
    - Pass an **action object** (below is example)
        - `action.type` is the name of the action
        - `action.(anything else)` are needed data
    - **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 - easy to test!

# Dispatch function uses the reducer

Imagine a function that has the reducer

- React aware
    - Has an object for state
    - Knows the setter for that state
- Is passed the action object
    - Calls the reducer
        - passing reducer the state
        - passing reducer the action object
    - Sets the new state to result from reducer

# `useReducer` hook

```
useReducer(reducer, initialState);
```

- `initialState` is a default value
  - Like with `useState()`
- 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:

```
function 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
- Want to abstract complicated state changes
- State-changing logic that you want
    - To reuse
    - To have testable outside of components

# useReducer Alternatives

Many **state management** libraries exist

- `react-query`
- `Redux`
- etc

Some wrap `useReducer`

- Others are separate decisions

All are about state management

- Still have and use state

# Summary - reducer

A **reducer** function

- Takes the current state + an action object
- Returns a new state object
- 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
- "knows" the state
- Updates the app state
    - Triggers render
- How you "use" an action

# Summary - useReducer

- Hook takes reducer + initial state
- returns state + dispatch function

```
const [state, dispatch] = useReducer(reducer, initState);
```

Dispatch function

- Can be passed to children
- Can be wrapped
    - Wrapper passed to children
    - Children can only "dispatch" via wrapper
        - Decouples children from state
        - Like `<Login onLogin={}/>`

# Summary - when to use a reducer

- `useState` is perfectly valid
- `useReducer` when you want
    - Abstracted sets of state changes
    - Reusable actions

Internally, `useState` is just a simple `useReducer`!

- (I have not recently confirmed this statement)