React UI So Far

- Rendered UI based on state
- Conditionally render components
- Conditionally render attributes
    - Esp. `className` for changed styling
- Conditionally render text and elements

React State So Far

- State defined in different components
- Some state scoped to lower components
- Shared state in higher level components
- State passed as props to children
- Setters and wrapped setters passed as props

Complex Application State Changes

Complex state changes with `useState`

- Changing multiple fields at once
- Repetitive if reason for change happens often
- Easy to make a mistake
- May have to make changes in many places

Answer: `useReducer` hook

State as an object

Imagine our Todo state as a single object

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

Pros and Cons

Pro:

- Changes can be made **atomically**
  - One setter call
- Easy to pass around
  - Can pass all as prop or parts as props

Con:

- Will trigger large rerender if anything changes
  - "Rerender" === component functions run
  - React will only change DOM when needed

Actions on the state

Action === A set of state changes for a purpose

- `login`, `logout`, `toggleTodo`, etc
- Named for the **event happening to the state**
  - NOT the page it is happening on

Imagine: The change in state due to an action:

```
function logout(state) {
  return {     // New object, not mutation
    ...state,  // Copies current state
    IsLoggedIn: false,  // Overwrites this property
    username: '',
    todos: {},
  };
}
```

We Can Imagine Many Such Action Functions

- Each takes object representing current state
    - And any params needed for new state
- Each returns a new object showing resulting state

```
// Example: login(state, { username: 'Jean' });

function login(state, details) {
  return {     // New object, not mutation
    ...state,  // Copies current state
    IsLoggedIn: true,  // Overwrites this property
    username: details.username, // New values passed in
  };
}
```

We Aren't Mutating the State Object

- Copy previous state properties
    - Can explicitly copy nested values
- Only focus on properties impacted by action
- No side-effects

```
// Example: toggleTodos(state, { id: 'asdf' })

function toggleTodo(state, details) {
  return {     // New object, not mutation
    ...state,  // Copies current state
    todos: {   // Overwrites this property
      ...state.todos,   // Copies this current state
      [details.id]: { // ...but overwrite this one
        ...state.todos[details.id], // as a copy of current
        done: !state.todos[details.id].done, // Except here
      },
    },
  };
}
```

Why Was That So Messy?

- We're avoiding mutation in nested objects

Consider:

```
function nestedStateDemo(state, details) {
  const id = details.id;

  const newState = { ...state }; // Shallow copy

  console.log(state === newState); // false
  console.log(state.todos === newState.todos); // true! ⁉️
  console.log(state.todos[id] === newState.todos[id]);// true

  newState.todos[id].done = !state.todos[id].done;

  console.log(
    state.todos[id].done === newState.todos[id].done
  ); // true - mutated state, React gets confused 😿
}
```

Alternate version with deep copy

```
// Example: toggleTodos(state, { id: 'asdf' })

function toggleTodo(state, details) {
  const newState = structuredClone(state); // Deep copy!
  const id = details.id;

  console.log(state === newState); // fals
  console.log(state.todos === newState.todos); // false 😻

  newState.todos[id].done = !state.todos[id].done;

  console.log(state.todos[id] === newState.todos[id]);//false

  return newState;
}
```

Both approaches work, different Pros/Cons

```javascript
function toggleTodo(state, details) {
  return {
    ...state,
    todos: {
      ...state.todos,
      [details.id]: {
        ...state.todos[details.id],
        done: !state.todos[details.id].done,
      },
    },
  };
}
```

```javascript
function toggleTodo(state, details) {
  const newState = structuredClone(state);
  const id = details.id;

  newState.todos[id].done = !state.todos[id].done;
  return newState;
}
```

No React, no UI in deciding changes to state

This is all pure state logic

We aren't using React or HTML!

- No JSX, no Presentation
- Pure vanilla JS

Not even setting actual state

- Just calculating what the new state would be

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

```
// Not the actual code, but captures concept
function dispatcher( action ) {
  const newState = reducer(state, action);
  setState(newState);
}
```

- You define and pass an `action` object
    - `action.type` is the action type
    - `action.`(everything else) are needed params
- `dispatcher`
    - Calculates new state using reducer
    - Actually sets the state

```
dispatcher({ type: 'logout' });
dispatcher({ type: 'login', username: 'Jorts'});
dispatcher({ type: 'toggleTodo', id: 'asdf' });
```

`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

Multiple **state management** libraries exist

- `react-query` (`@tanstack/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