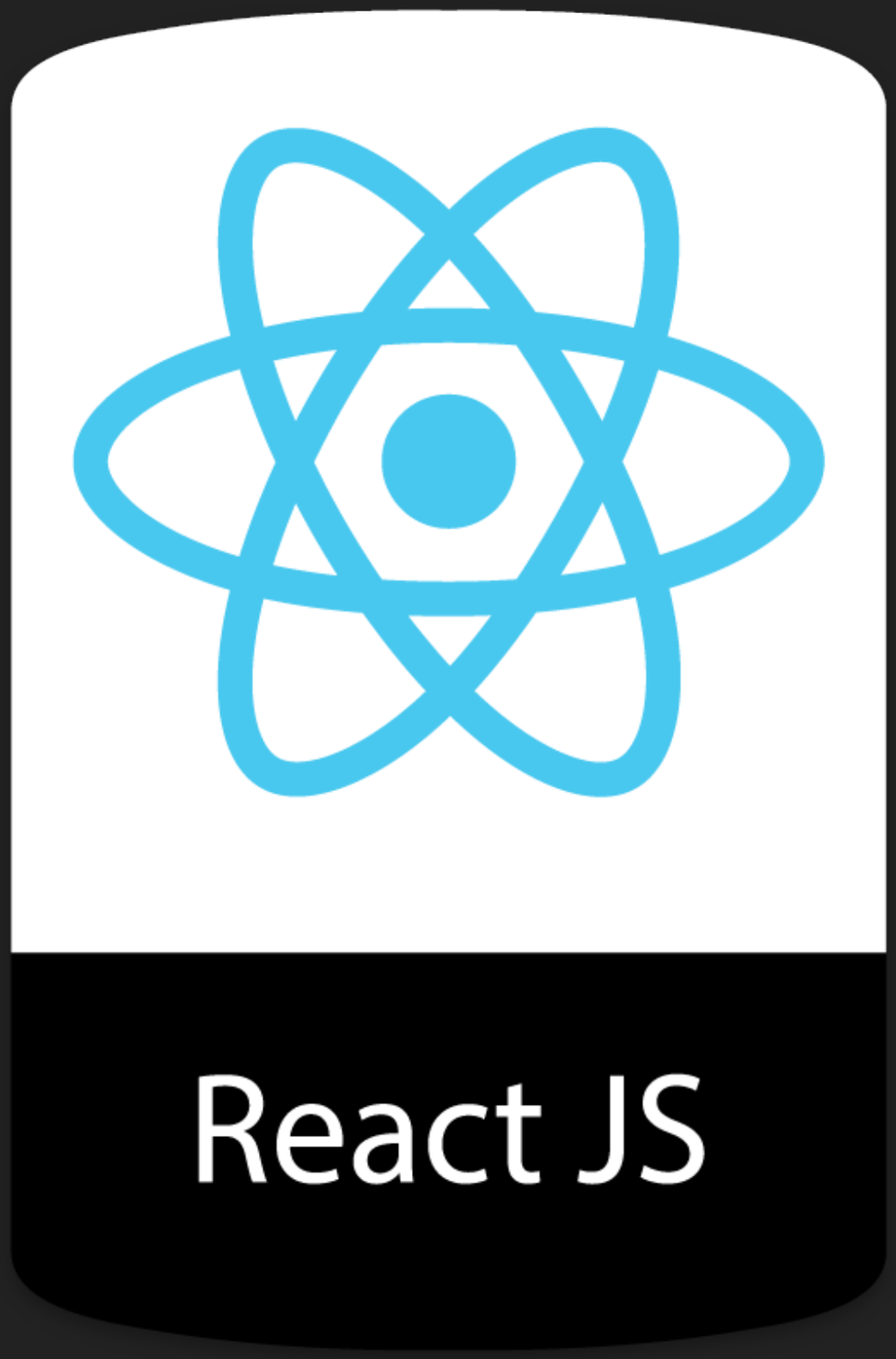
TOP-DOWN STATE MANAGEMENT AND GLOBAL EVENT BUS

# REACT WITH REDUX

# WHAT IS REACT?

- The View layer of MVC
- That's it
- Nothing else
- No AJAX stuff
- Not a data store
- Just chain-able components that accept props as input and can manage their own state



# KEY CONCEPTS

- Top-down data flow - Data should be managed as high as possible in the component hierarchy.  Requests to change data should be sent from the child components up to whatever component is managing the applications state.
- Virtual DOM - changes to the applications state are not made directly to the DOM.  Instead, changes are made to a Virtual DOM which is then diffed against the current DOM tree.  Only DOM elements that are dirty will be changes and re-rendered.  This minimizes changes to the DOM and lowers the amount of repaints the browser has to make (CSS styling is actually the most expensive part of rendering a web page)
- JSX - Syntactical sugar to allow writing html like markup that actually represent chained function calls.

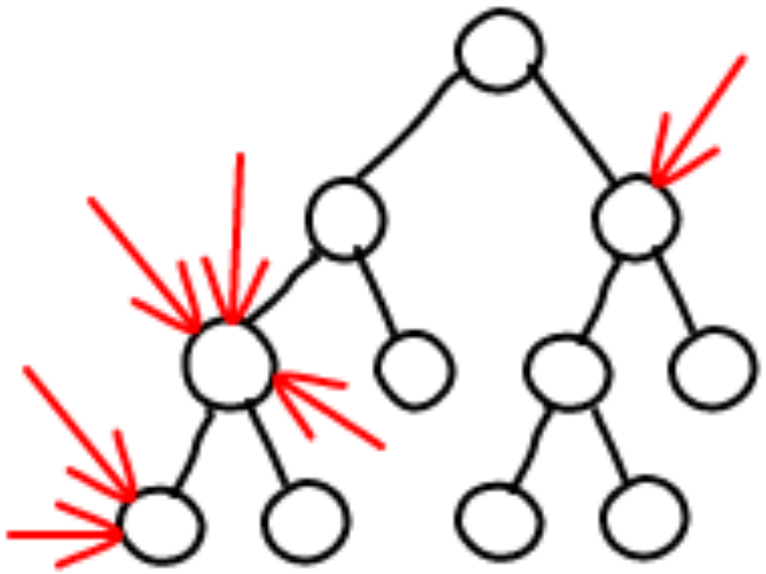
# TOP-DOWN DATA FLOW

- Learning where to manage state can be one of the hardest parts about learning ReactJS
- State should be managed as high as possible in the component hierarchy and then passed down to children components.  There are a number of ways to pass props such as directly passing props, using the component context, or using a state management solution such as Redux which implements the FLUX architecture pattern

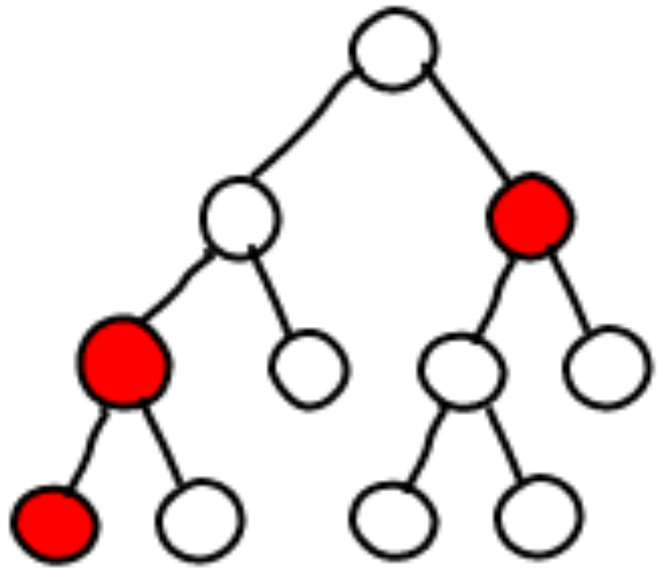
# VIRTUAL DOM

- To make changes to a web UI, the Document Object Model (DOM) must be updated
- Most UI frameworks (jQuery-UI, Angular 2, etc) frequently make direct changes to the DOM, often time updating the DOM multiple times for a single application state change
- The Virtual DOM solves a lot of these issues by first making changes to a fast, in-memory, copy of the DOM.  This is then diffed against the current DOM tree.  Only DOM nodes that are dirty and their children will be changed and thus repainte

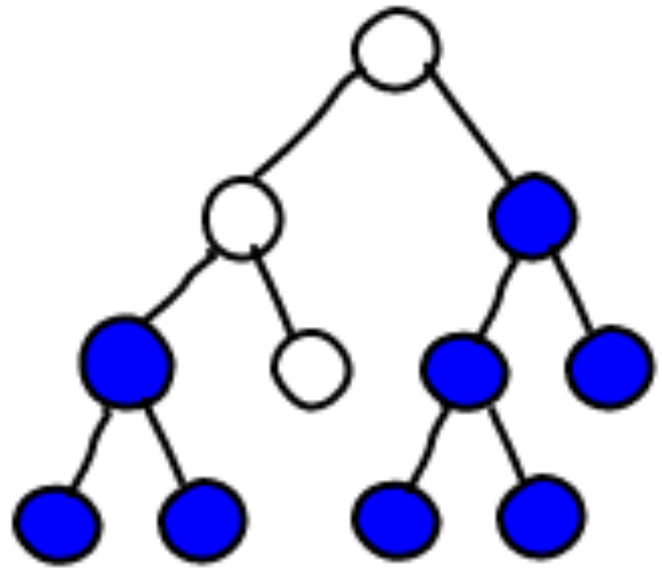
setState



Dirty



Re-rendered



# JSX

- JSX is not a templating tool like Angular 1 templates
- JSX is actually syntactic sugar with each "DOM node" actually representing a part of a function call chain
- JSX allows us to easily represent complex function chains and property passing in an easy to read and understand visual format

# JSX

```
<form>
  <div className={`form-group}`}>
    <label htmlFor='code' className={'required'}>Code</label>
    <input type='text' value={this.state.code} onChange={this.updateNodeCode}
      className={`form-control ${codeAlreadyExists ? 'is-invalid' : ''}`}
      id='code' aria-describedby='codeHelp' placeholder='Enter Scenario Node Code'
    />
    <small id='codeHelp'
      className={`form-text ${codeAlreadyExists ? 'invalid-feedback' : 'text-muted'}`}
    >
      Code must be unique for the current initiative date.
    </small>
  </div>
</form>
```

# PURE FUNCTION CALLS

```
React.createElement(
  'form',
  null,
  React.createElement(
    'div',
    { className: 'form-group}' },
    React.createElement(
      'label',
      { htmlFor: 'code', className: 'required' },
      'Code'
    ),
    React.createElement('input', { type: 'text', value: undefined.state.code, onChange: undefined.updateNodeCo
de,
      className: 'form-control ' + (codeAlreadyExists ? 'is-invalid' : ''),
      id: 'code', 'aria-describedby': 'codeHelp', placeholder: 'Enter Scenario Node Code'
    }),
    React.createElement(
      'small',
      { id: 'codeHelp',
        className: 'form-text ' + (codeAlreadyExists ? 'invalid-feedback' : 'text-muted')
      },
      'Code must be unique for the current initiative date.'
    )
  )
);
```

# REDUX

- Represents the Model and Controller aspects of MVC
- Owns and controls global state for your application
- State is divided into stores, which are each managed by a set of reducers
- Changes to application state of done my emitting actions on Redux's global event bus

# THREE PRINCIPLES

- Single source of truth - The state of your application is stored as a single object tree
- State is read only - All changes to state must be emitted as actions.  Changing the state in-place will cause unexpected and inconsistent results
- Changes are made with pure functions - Reducers are just pure functions that take the previous state and an action. A new state is created using the previous state as a base and making the changes requested in the action

# STORE

- Holds the application state
- Provides access to the state via `getState()`
- Allows state update via `dispatch(action)`
- Registers listeners via `subscribe(listener)`
- Is created by combining 1...n reducers which handle dispatched actions

# REDUCERS

- Handles actions and updates the part of the store the reducer controls accordingly
- Generally the main method of a reducer is a single switch statement that takes the actions type and passes the current state and the action onto the corresponding function to process the action
- The default section of the switch statement simply returns the current state

# ACTIONS

- Actions are the information payload that informs the store of changes that need to be made to the current state
- All actions should have a type property.  This type property is what reducers use to decide if they should handle the action or not
- Any sort of data can be passed via an action
- Actions should not dispatch themselves in Redux.  Instead actions should be created and then passed to dispatch
- To allow for async action creation, we generally rely on the Redux middleware Thunk