Redux

Managing Client State with React

The Problem

- Mutation
- Async Results

Components Downside

- Component architectures spawn distributed state in components - **everywhere — Component Spaghetti**
- React state is determined by
  - Input - Props
  - Internal Component State (Redux eliminates much of this state)
- After clicking around several screens we really don't know what state our application is anymore (or every screen has to re-request everything to know we are consistent)
- *Each new feature introduces new state changes which introduce unpredictable behavior for the UI as a whole*

The Solution

- **Redux** tries to make async state changes predictable
- When **Redux** is used with **React**, it turns React into a simple Data —> DOM transformer (*because the state is essentially removed from the component and placed in a Redux store*)

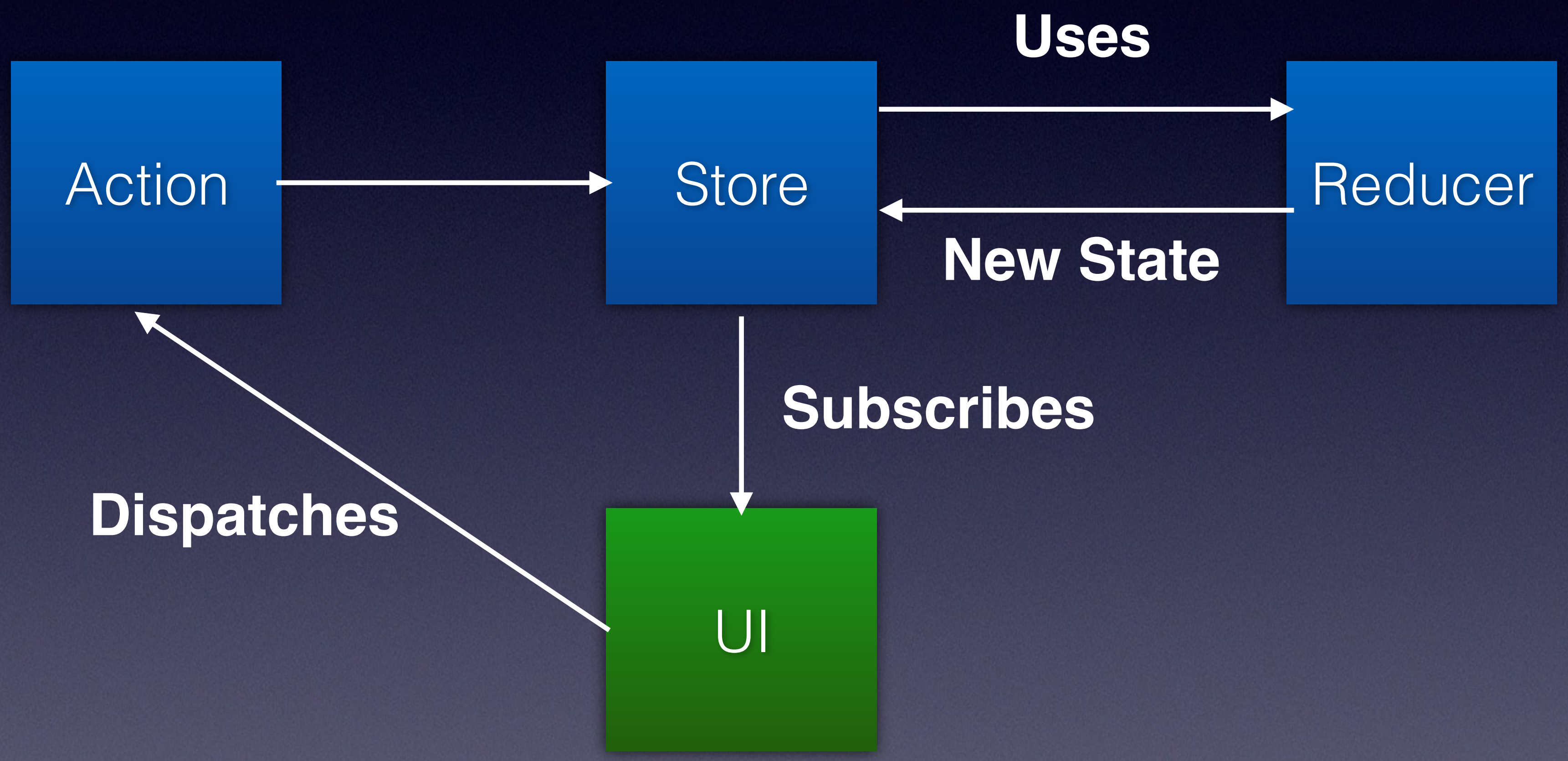
How Can It Help

- UI Framework isolation via common *Actions* versus framework centric *Services*
- *Decouples Component Interaction*
- Consolidate distributed component state is moved into a single immutable store
- Possible to serialize all or parts of the store to a DB or browser DB to resume that state from previous session

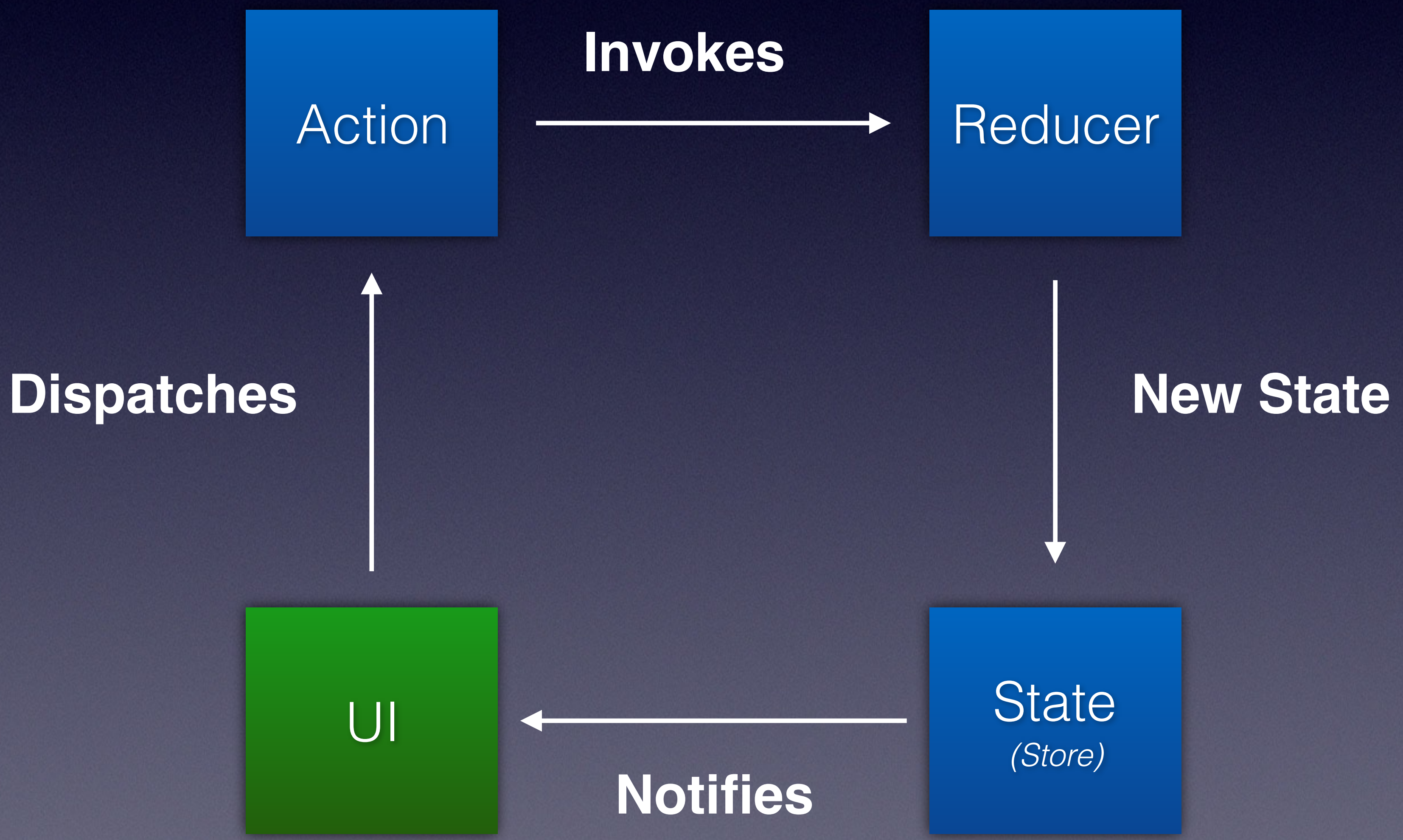
Redux Key Concepts

- Immutable State Management
- Reducers are Pure functions (functional, no side effects)
- Reducers don’t change the state they return a new state: Reducer(state, action) -> Store
- Action has a type and optional payload
- State change can be subscribed

Redux Key Concepts



Redux Key Concepts



Conceptual

- Conceptually the **store** can be pictured as a hierarchical tree of data. And any part of the tree can be subscribed to when it changes

Redux Actions

- Actions consist of
  - Type (Required)
  - Payload (Optional)
  - Anything else you want
- Typically follow this pattern
  - NAMESPACE_REQUEST
  - NAMESPACE_SUCCESS
  - NAMESPACE_FAILURE

Reducer Example

```javascript
const redux = require('redux');
const createStore = redux.createStore;

const initialState = {
  counter: 0
};

// Reducer
const rootReducer = (state = initialState, action) => {
  if (action.type === 'INC_COUNTER') {
    return {
      ...state,
      counter: state.counter + 1
    };
  }
  if (action.type === 'DEC_COUNTER') {
    return {
      ...state,
      counter: state.counter - 1
    };
  }
  return state;
};
```

Store/Subscribe/Dispatch

```
// Store
const store = createStore(rootReducer);
console.log(store.getState());

// Subscription
store.subscribe(() => {
  console.log('[Subscription]', store.getState());
});

// Dispatching Action
store.dispatch({type: 'INC_COUNTER'});
store.dispatch({type: 'INC_COUNTER'});
store.dispatch({type: 'DEC_COUNTER'});
console.log(store.getState());
```

React Redux Setup

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './components/App'
import reducer from './reducers'

const store = createStore(reducer)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Computed Values

- Done via *Reselect* library
    - Computes Dervived data
    - Data is not recomputed unless it changes

Reselect Example

```
import { createSelector } from 'reselect'

const shopItemsSelector = state => state.shop.items
const taxPercentSelector = state => state.shop.taxPercent

const subtotalSelector = createSelector(
  shopItemsSelector,
  items => items.reduce((acc, item) => acc + item.value, 0)
)

const taxSelector = createSelector(
  subtotalSelector,
  taxPercentSelector,
  (subtotal, taxPercent) => subtotal * (taxPercent / 100)
)

export const totalSelector = createSelector(
  subtotalSelector,
  taxSelector,
  (subtotal, tax) => ({ total: subtotal + tax })
)

let exampleState = {
  shop: {
    taxPercent: 8,
    items: [
      { name: 'apple', value: 1.20 },
      { name: 'orange', value: 0.95 },
    ]
  }
}
```

Async

- *Because reducers are pure functions with no side-effects some tricks are needed*
- *Redux-Thunk*
- *Redux-Observable (Rx.js)*
- *Redux-Promise*
- *Redux Saga (ES6 generators)*

Redux Thunk

- *A thunk is a function that wraps an expression to delay its evaluation*

```
// Meet thunks.
// A thunk is a function that returns a function.
// This is a thunk.

function makeASandwichWithSecretSauce(forPerson) {

  // Invert control!
  // Return a function that accepts `dispatch` so we can dispatch later.
  // Thunk middleware knows how to turn thunk async actions into actions.

  return function (dispatch) {
    return fetchSecretSauce().then(
      sauce => dispatch(makeASandwich(forPerson, sauce)),
      error => dispatch(apologize('The Sandwich Shop', forPerson, error))
    );
  };
}
```

•

Debugging

- Time Traveling Debugger (Awesome!)
  - Consistently reproduce your application state for any point in time
- QE can even dump the state.json to reproduce exact state of application when bug occurred
- Testing is much easier without the need for Mocks or spies

Redux Negatives

- Another Layer of code (more code/more moving parts)
- Normally not necessary to start out with Redux
- Different Mental Model to learn

Is Redux Needed

- How much state does my application have and how many ways can that state be changed?
- How much data is shared among other views (think common headers/sidebars in different screens)

Summary

- The Redux Architecture is a big win for **medium/large** Redux applications by organizing state into a single source of truth.
- It allows different sections of the application to share state changes via subscriptions.
- Keeps the application state *predictable* and *reproducible*. Application state can be predictably reproduced at **any** point in time. This is huge for reproducing a bug — QE can send the state json at the time of the bug and the exact application state is resumed.
- Redux is very popular and the ecosystem is huge: https://github.com/xgrommx/awesome-redux

One More Thing

- Jaeger UI uses Redux currently — We could use a shared redux store to communicate to Jaeger — *This is currently under investigation.*

Demo

- Time Travel Debugger
- Undo/Redo