Redux

Predictable State Management

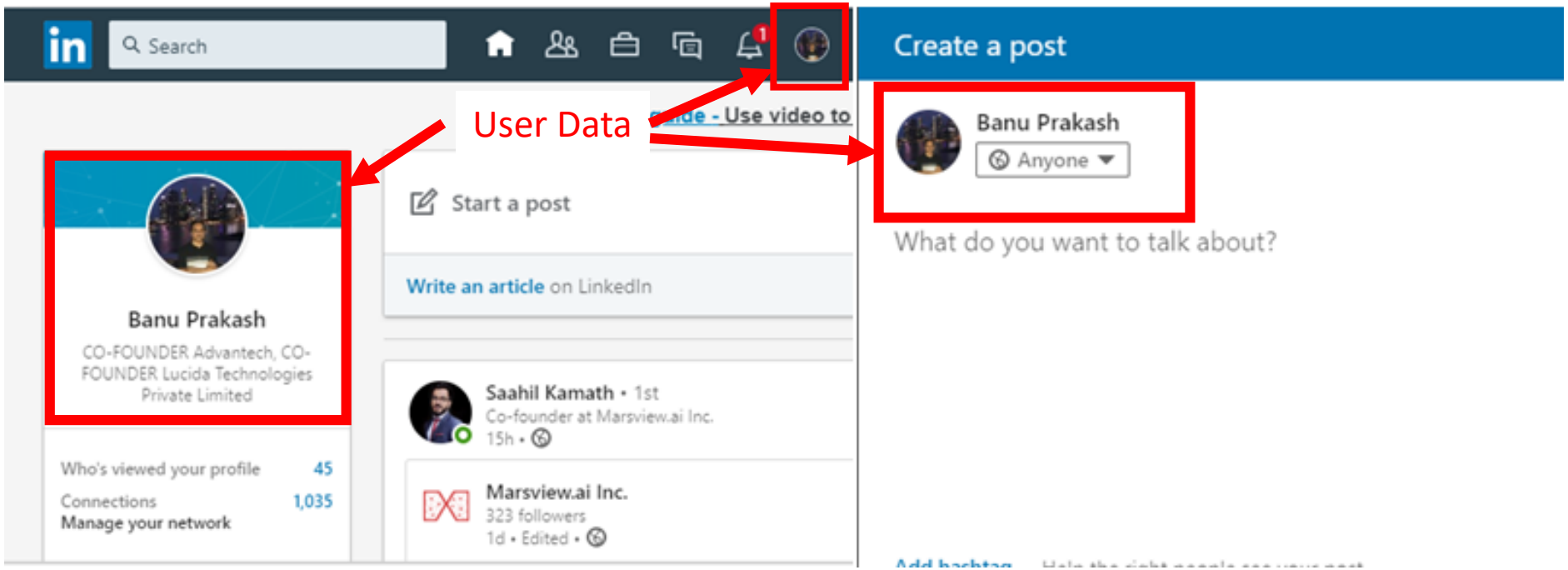
Banuprakash C

Motivation

- JavaScript single-page applications have become increasingly complicated, **our code must manage more state than ever before**.
- If a model can update another model, then a view can update a model, which updates another model, and this, in turn, might cause another view to update.
- At some point, you no longer understand what happens in your app as you have **lost control over the when, why, and how of its state.**

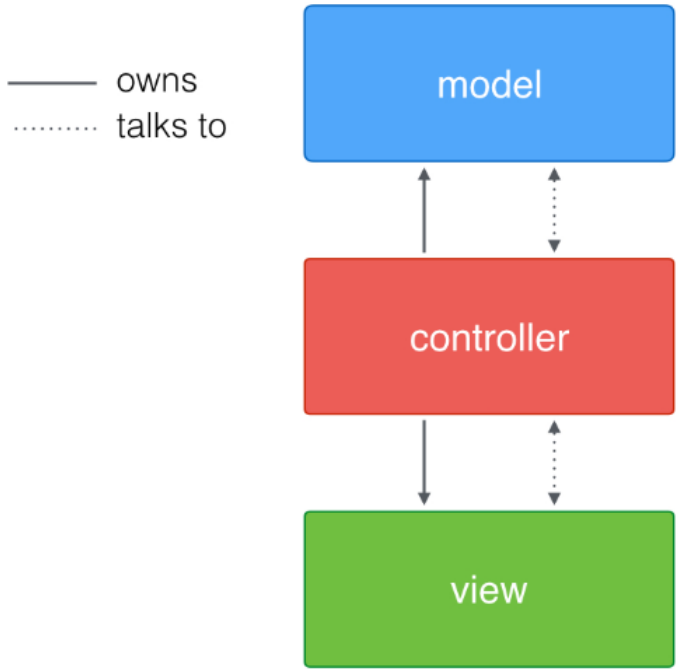
Passing Data Multiple Levels is a Pain

- Top-level container has some data, and a child 4+ levels down needs that data. Here's an example from Linkedin, with all the avatars highlighted:



The MVC pattern

- In general most MVC patterns considers three roles:
  - Model: manages the behavior and data of the application domain
  - View: represents the display of the model in the UI
  - Controller: takes user input, manipulates the model and causes the view to update
- The core ideas of MVC can be formulated as following:
  - Separating the presentation from the model:
  - Enables implementation of different UIs and better testability
- MVC is a legendary pattern that has been used for various projects since 1976 (it was introduced in Smalltalk-76).

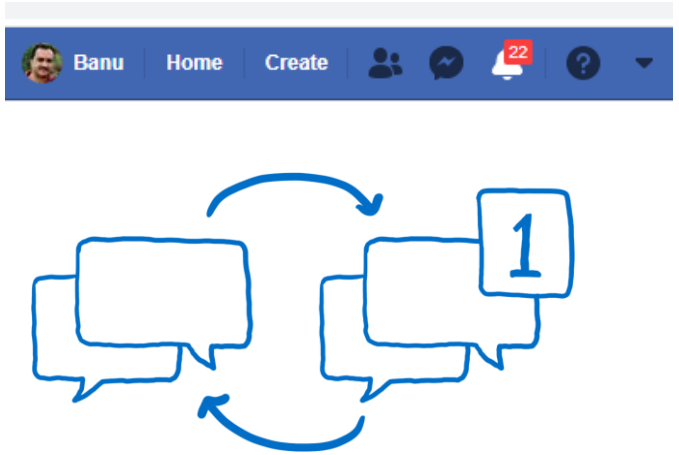


Flux pattern

- Flux is a pattern for handling data in your application.
- Flux and React grew up together at Facebook.
- Many people use them together, though you can use them independently.
- They were developed to address a particular set of problems that Facebook was seeing.

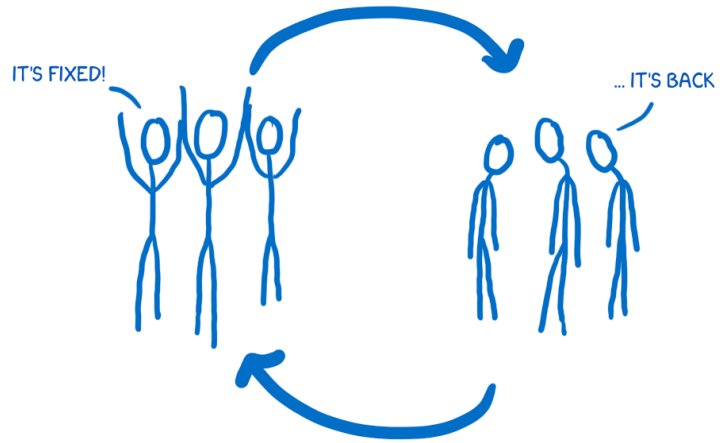
Flux pattern

- In Facebook one of the problems was the notification bug.
- When you logged in to Facebook, you would see a notification over the messages icon.
  - When you clicked on the messages icon, though, there would be no new message. The notification would go away.
  - Then, a few minutes later after a few interactions with the site, the notification would come back.
  - You'd click on the messages icon again… still no new messages. It would just keep going back-and-forth in this cycle.



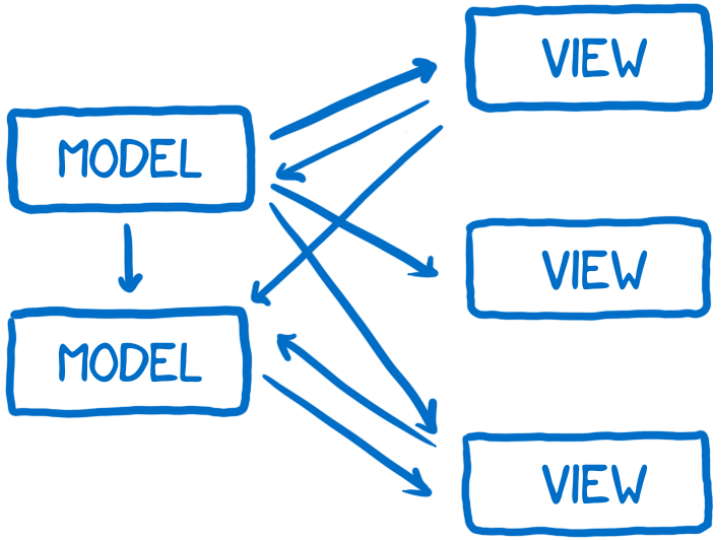
Flux pattern

- There was also a cycle going on for the team at Facebook.
  - They would fix this bug and everything would be fine for a while and then the bug would be back.
  - It would go back-and-forth between being resolved and being an issue again.
- So Facebook wanted to make the system predictable so they could ensure that this problem wouldn’t keep resurfacing.



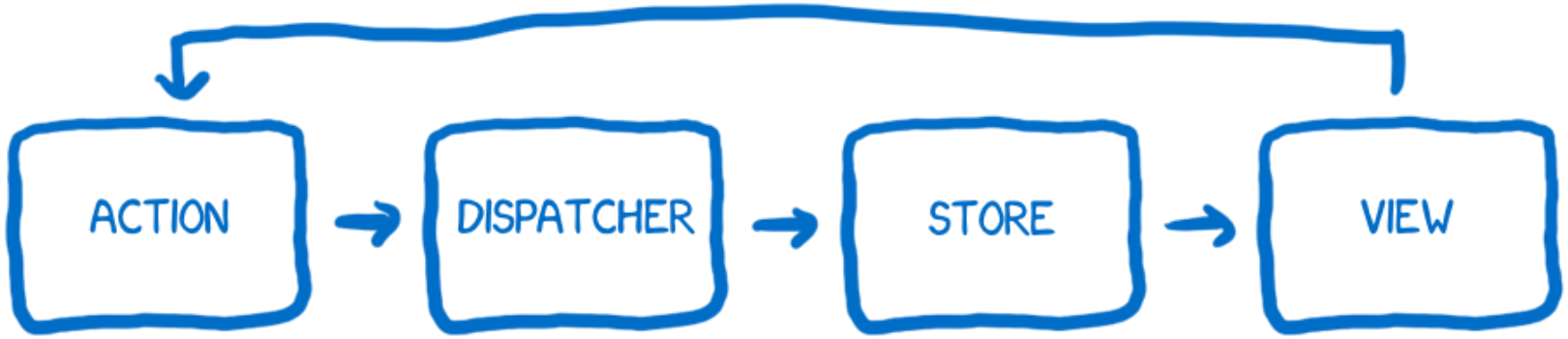
The underlying problem

- The underlying problem that they identified was the way that the data flowed through the application.
- They had models which held the data and would pass data to the view layer to render the data.
- Because user interaction happened through the views, the views sometimes needed to update models based on user input. And sometimes models needed to update other models.
- On top of that, sometimes these actions would trigger a cascade of other changes and these changes could be happening asynchronously.



Flux => The solution: unidirectional data flow

- Facebook decided to try a different kind of architecture, where the data flows in only one direction, and when you need to insert new data, the flow starts all over again at the beginning.
- They called their architecture Flux.



Flux characters

- **The action creator**
  - Whenever you want to change the state of the app or have the view render differently, you shoot off an action.
  - You go to the action creator knowing basically what message you want to send, and then the action creator formats that in a way that the rest of the system can understand.
  - The action creator creates an action with a type and a payload.
    - The type will be one of the types that you have defined as actions in your system (usually a list of constants).
    - An example of an action would be something like MESSAGE_CREATE or MESSAGE_READ

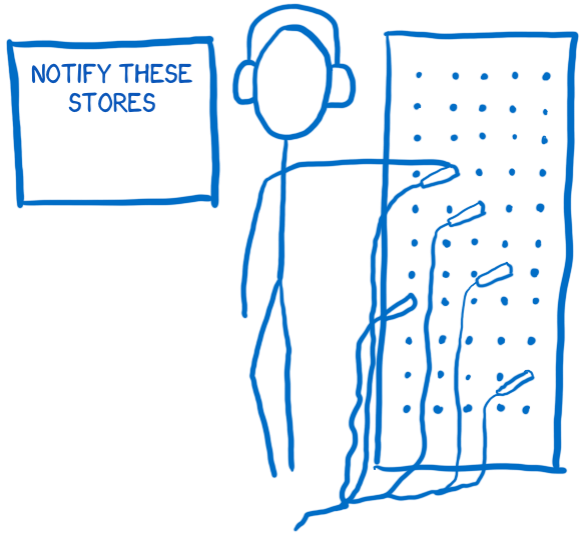
Think of the action creator as a telegraph operator.

Flux characters: The action creator example

```
import dispatcher from "../Dispatcher";

export const COLOR_APP_ACTIONS = {
    CHANGE_COLOR: 'colorAppActions.ChangeColor'
};

export function changeColor(colorName) {
    dispatcher.dispatch({
        type: COLOR_APP_ACTIONS.CHANGE_COLOR,
        value: colorName
    })
}
```

Flux characters

- **The dispatcher**
  - The dispatcher is basically a big registry of callbacks.
  - It keeps a list of all of the stores that it needs to send actions to. When an action comes in from the action creator, it will pass the action around to different stores.
  - The action is sent to all of the registered stores regardless of what the action type is. This means the store doesn't just subscribe to some actions. It hears about all actions and filters out what it cares about and doesn't.



Dispatcher is like a telephone operator at a phone switchboard

Flux characters: The dispatcher

- import {Dispatcher} from "flux";
- export default new Dispatcher();

- This is the dispatcher for our app which will dispatch actions invoked in components to the registered stores.

Flux characters

- **The store**
  - The store holds on to all state in the application, and all of the state changing logic lives inside of the stores.
  - All state changes must be made by it personally. And you can't directly request that it change the state.
  - To request a state change, you must follow proper procedure.
    - You must submit an action via the action creator/dispatcher pipeline

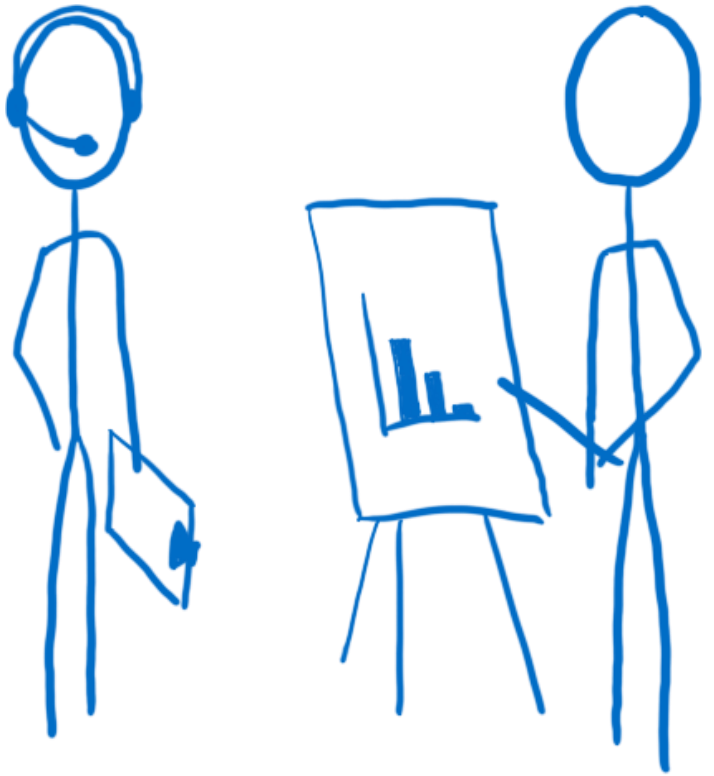
Think of the store as an over-controlling bureaucrat

Flux characters: The store

```
class ColorAppStore extends EventEmitter {

    constructor() {
        super();
        this.activeColor = "lightgrey";
    }

    handleActions(action) {
        switch (action.type) {
            case ColorAppActions.COLOR_APP_ACTIONS.CHANGE_COLOR: {
                this.activeColor = action.value;
                this.emit("storeUpdated");
                break;
            }
            default: {
            }
        }
    }

    getActiveColor() {
        return this.activeColor;
    }
}

const colorAppStore = new ColorAppStore();
dispatcher.register(colorAppStore.handleActions.bind(colorAppStore));
export default colorAppStore;
```

Flux characters

- **The controller view and the view**
  - The view is a presenter. It isn't aware of anything in the application, it just knows the data that's handed to it and how to format the data into output that people understand.
  - The controller view is like a middle manager between the store and the view. The store tells it when the state has changed. It collects the new state and then passes the updated state along to all of the views under it.

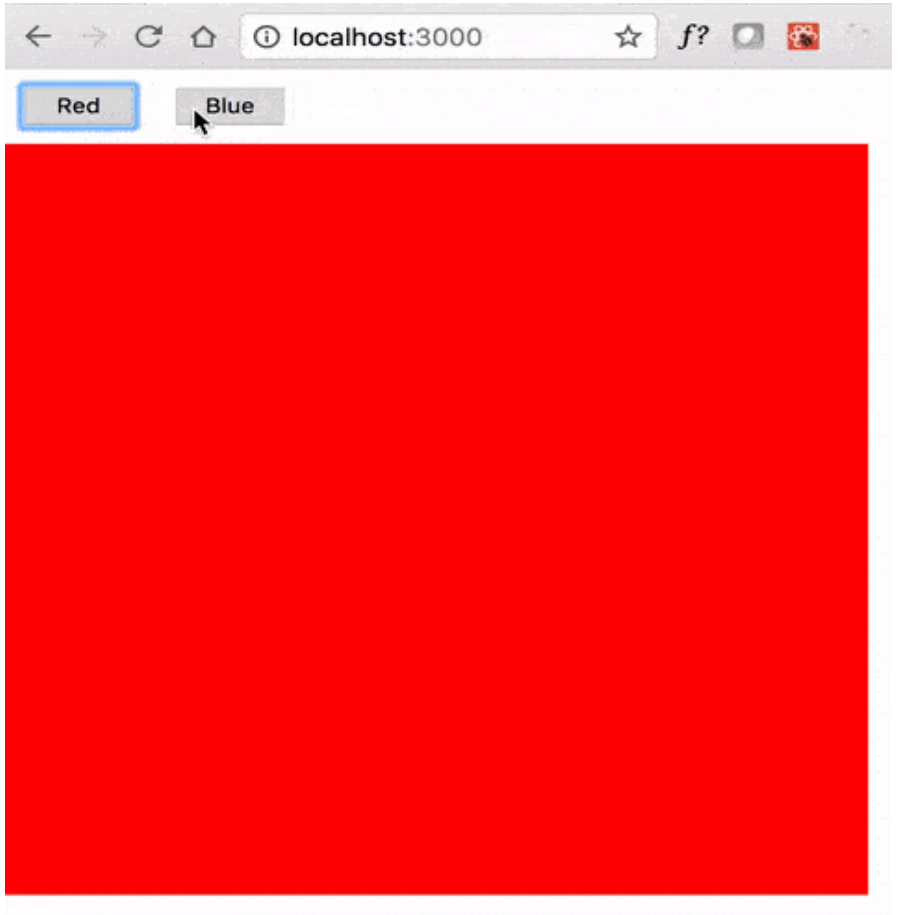


Flux characters: View

```
export default class ColorComponent extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
            color: ColorAppStore.getActiveColor()
        }
    }

    componentDidMount() {
        ColorAppStore.on("storeUpdated", this.updateBackgroundColor);
    }

    componentWillUnmount() {
        ColorAppStore.removeListener("storeUpdated", this.updateBackgroundColor);
    }

    updateBackgroundColor = () => {
        this.setState({color: ColorAppStore.getActiveColor()})
    };

    render() {
        return (
            <div className="color-container" style={{backgroundColor: this.state.color}}/>
        );
    }
}
```

Flux characters: View

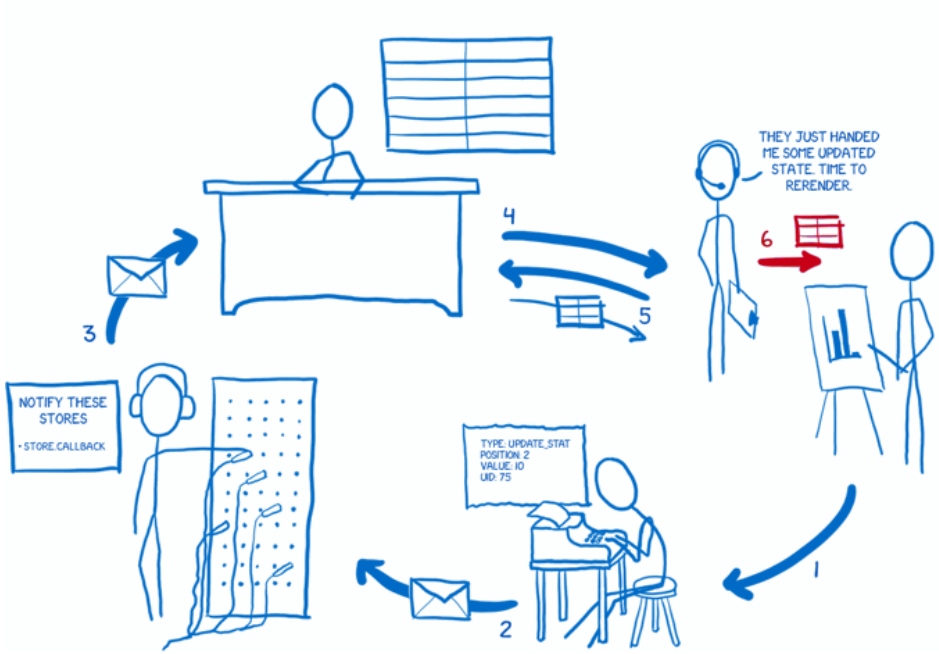
```
import React from "react";
import * as ColorAppActions from "../actions/ColorAppActions";

export default class ButtonComponent extends React.Component {

    onButtonClick = (colorName) => {
        ColorAppActions.changeColor(colorName)
    };

    render() {
        return (
            <div>
                <button onClick={() => this.onButtonClick("red")} className="color-button">Red</button>
                <button onClick={() => this.onButtonClick("blue")} className="color-button">Blue</button>
            </div>

        );
    }
}


export default class App extends React.Component {

    render() {
        return (
            <div>
                <ButtonComponent/>
                <ColorComponent/>
            </div>

        );
    }
}
```



Flux characters

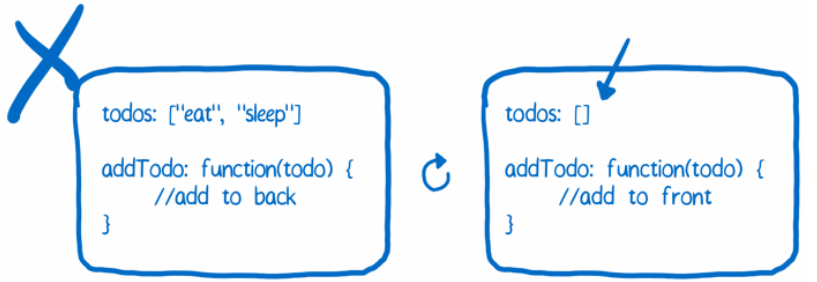
The data flow

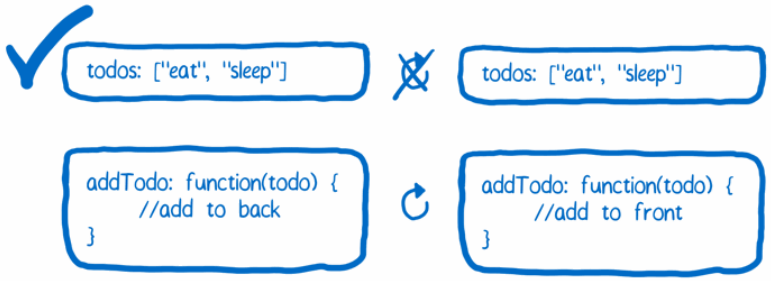
1. The view tells the action creator to prepare an action
2. The action creator formats the action and sends it off to the dispatcher
3. The dispatcher sends the action off to the stores in sequence. Each store gets notified of all actions. Then the store decides whether it cares about this one or not, and changes the state accordingly.
4. Once it's done changing state, the store lets its subscribed view controllers know.
5. Those view controllers will then ask the store to give them the updated state.
6. After the store gives it the state, the view controller will tell its child views to re-render based on the new state.



Why change Flux?
Problem 1

- In Flux, the store contains two things:
  - The state change logic
  - The current state itself
- Having these two on the same object is a problem for hot reloading. When you reload the store object to see the effect that the new state change logic has, you lose the state that the store is holding on to.
- **Solution**
  Separate these two functions. Have one object that holds on to the state. This object doesn't get reloaded. Have another object that contains all of the state change logic. This object can be reloaded because it doesn't have to worry about holding on to any state.

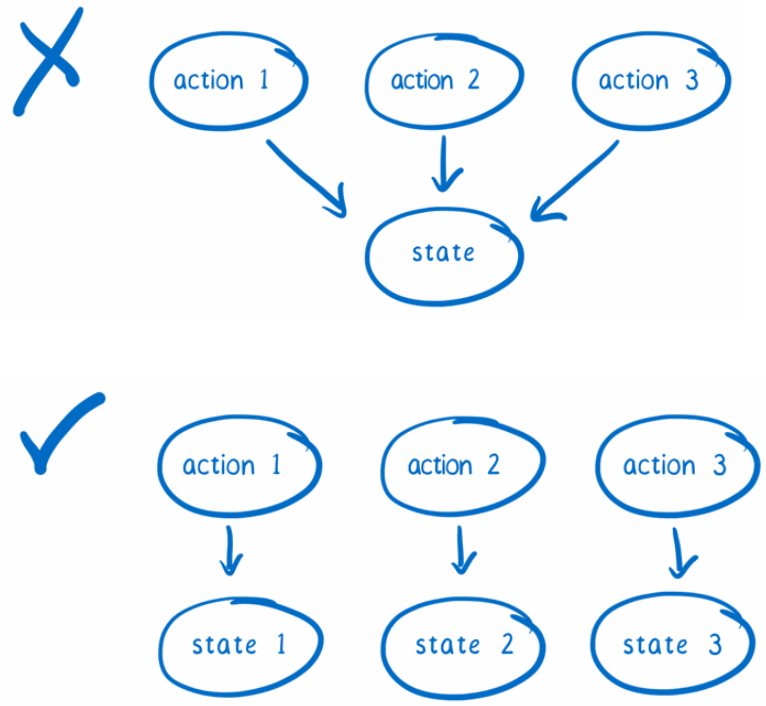




Why change Flux?

Problem 2

- The state is being rewritten with every action
  - In time travel debugging, you keep track of each version of a state object. That way, you can go back to an earlier state.
- **Solution**
  When an action comes in to the store, don't handle it by changing the state. Instead, copy the state and make changes to the copy.



Redux

- **The new cast of characters**
  - **Action creators**
    - Redux keeps the action creator from Flux. Whenever you want to change the state of the application, you shoot off an action. That's the only way that the state should be changed.
    - Unlike Flux, action creators in Redux do not send the action to the dispatcher. Instead, they just return a formatted action object.

Flux

```
function addTodo(text) {
  const action = {
        type: ADD_TODO,
        payload: text
  }
  Dispatcher.dispatch(action)
}
```

Redux

```
function addTodo(text) {
  return {
    type: ADD_TODO,
    payload: text
  }
}
```

Redux

- **The store**
  - In Redux, there is only one store…
  - there's no dispatcher, the store has also taken over dispatching.

```
import { createStore } from 'redux'
//combine several reducers into one
import todoApp from './reducers'
const store = createStore(todoApp)

//This is useful for hydrating the state of the client to match
// the state of a Redux application running on the server.
const store = createStore(todoApp, window.STATE_FROM_SERVER)
```

Redux

- **The reducers**
  - **Reducers** specify how the application's state changes in response to actions sent to the store.
  - The root reducer takes charge and slices the state up based on the state object's keys. It passes each slice of state to the reducer that knows how to handle it.
  - They don't change the state that has been passed in to them. Instead, they make a copy and make all their changes on the copy.
  - The reducers pass their copies back to the root reducer, which pastes the copies together to form the updated state object. Then the root reducer sends the new state object back to the store, and the store makes it the new official state.

Redux: **The reducers**

- todos.js

```
const todos = (state = [], action) => {
    switch (action.type) {
      case 'ADD_TODO':
        return [
          ...state,
          {
            id: action.id,
            text: action.text,
            completed: false
          }
        ]
      case 'TOGGLE_TODO':
        return state.map(todo =>
          todo.id === action.id ? { ...todo, completed: !todo.completed } : todo
        )
      default:
        return state
    }
  }
  export default todos
```

Redux: **The reducers**

- visibilityFilter.js

```
import { VisibilityFilters } from '../actions'
const visibilityFilter = (state = VisibilityFilters.SHOW_ALL, action) => {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}
export default visibilityFilter
```

Redux: Combine **reducers**

- index.js

```
import { combineReducers } from 'redux'
import todos from './todos'
import visibilityFilter from './visibilityFilter'
export default combineReducers({
  todos,
  visibilityFilter
})
```

Redux

- **The views: smart and dumb components**
  - The smart components are the managers.
  - Smart components are in charge of the actions. If a dumb component underneath them needs to trigger an action, the smart component passes a function in via the props. The dumb component can then just treat that as a callback.
  - Smart components rarely emit DOM of their own. Instead, they arrange dumb components, which handle laying out DOM elements.

Redux

- **The view layer binding**
  - To connect the store to the views, it needs something to bind the two together
  - If you're using React, this is react-redux.
- The view layer binding introduces:
  - The Provider component: This is wrapped around the component tree. It makes it easy for the root component's children to hook up to the store using connect().
  - connect(): If a component wants to get state updates, it wraps itself using connect(). Then the connect function will set up all the wiring for it

The view layer binding is kind of like the IT department for the view tree. It makes sure that all of the components can connect to the store.

Redux: Dumb & Smart components

```
const TodoList = ({ todos, toggleTodo }) => (
    <ul>
      {todos.map(todo => (
        <Todo
            key={todo.id} {...todo}
            onClick={() => toggleTodo(todo.id)} />
      ))}
    </ul>
  );
```

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case VisibilityFilters.SHOW_ALL:
      return todos
    case VisibilityFilters.SHOW_COMPLETED:
      return todos.filter(t => t.completed)
    default:
      throw new Error('Unknown filter: ' + filter)
  }
}
const mapStateToProps = state => ({
  todos: getVisibleTodos(state.todos, state.visibilityFilter)
})
const mapDispatchToProps = dispatch => ({
  toggleTodo: id => dispatch(toggleTodo(id))
})
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)
```

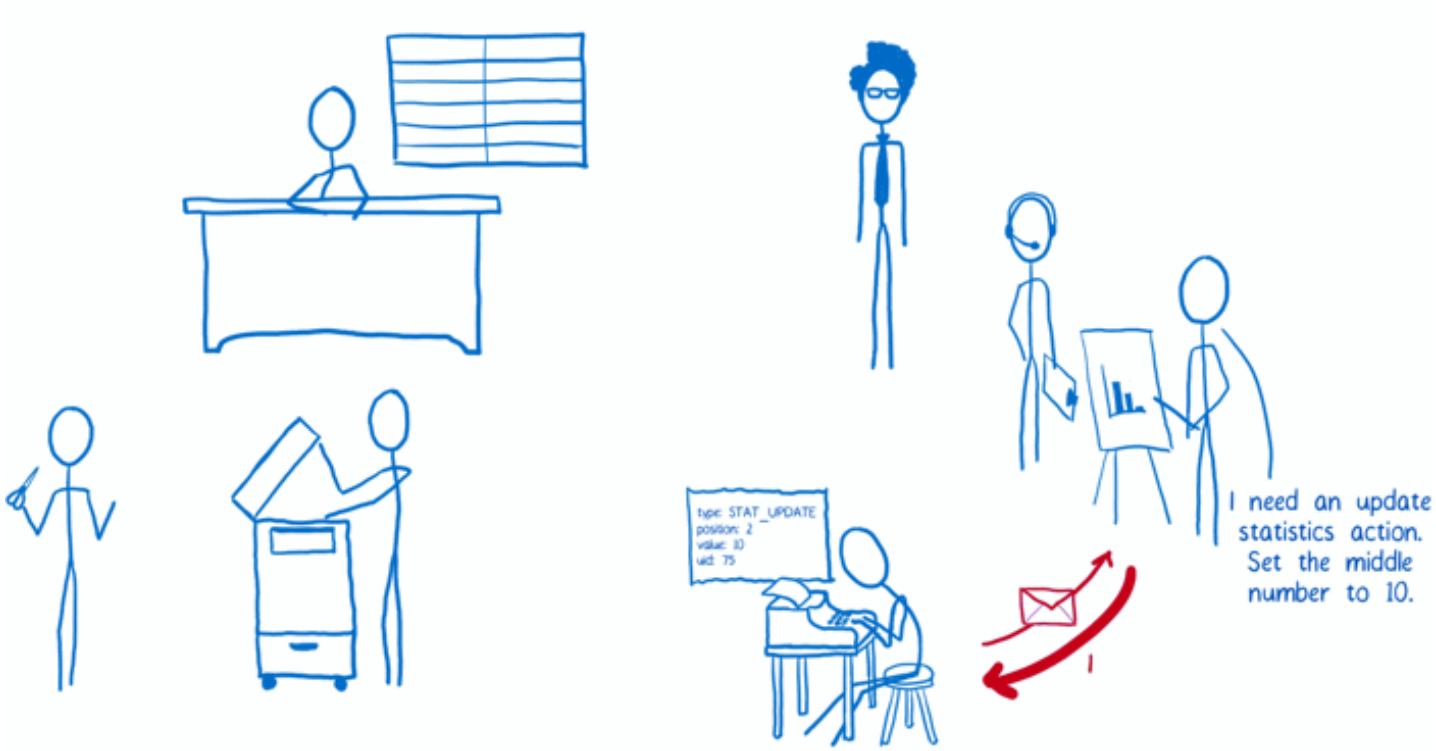
Redux

- **The root component**
  - It creates the store, telling it what reducer to use, and brings together the view layer binding and the views.
  - The root component is pretty hands-off after it initializes the app, though.

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

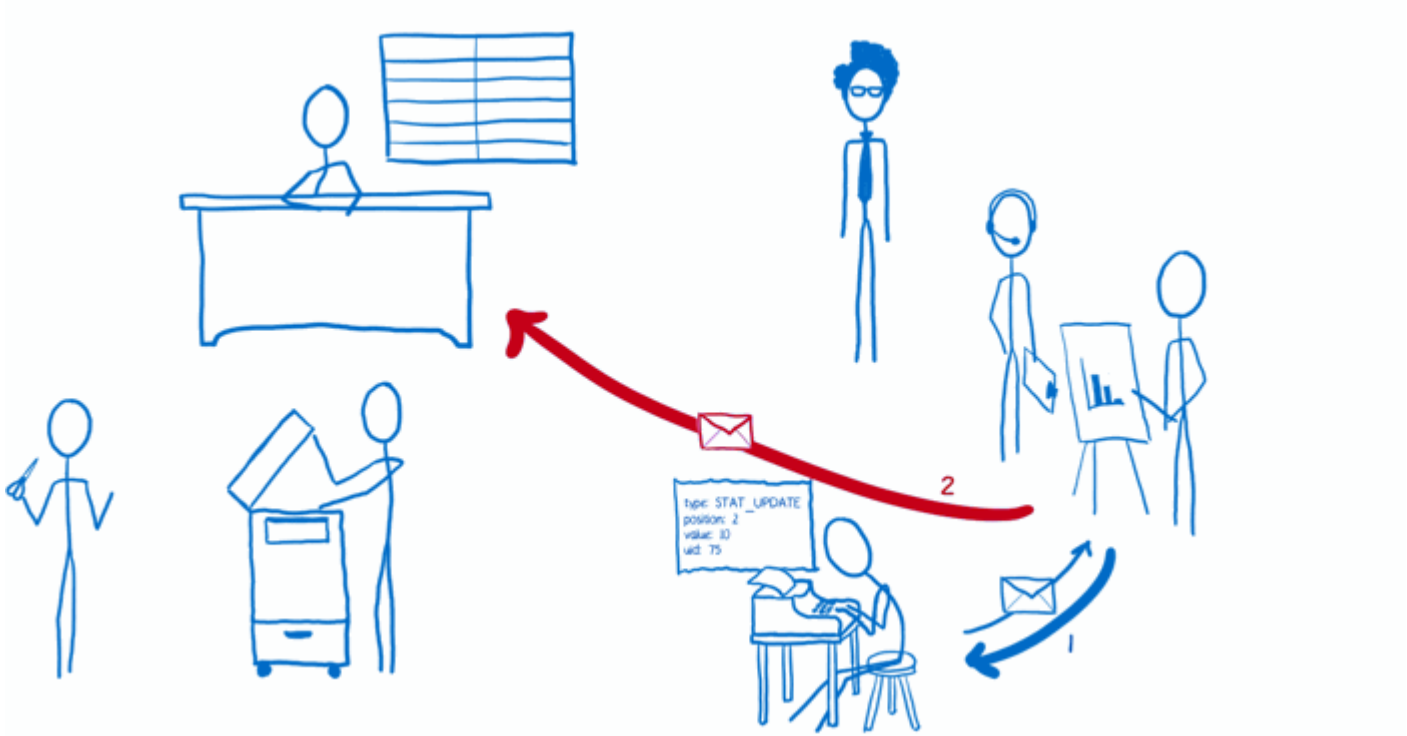
Redux: The data flow

1. The view requests an action. The action creator formats it and returns it.



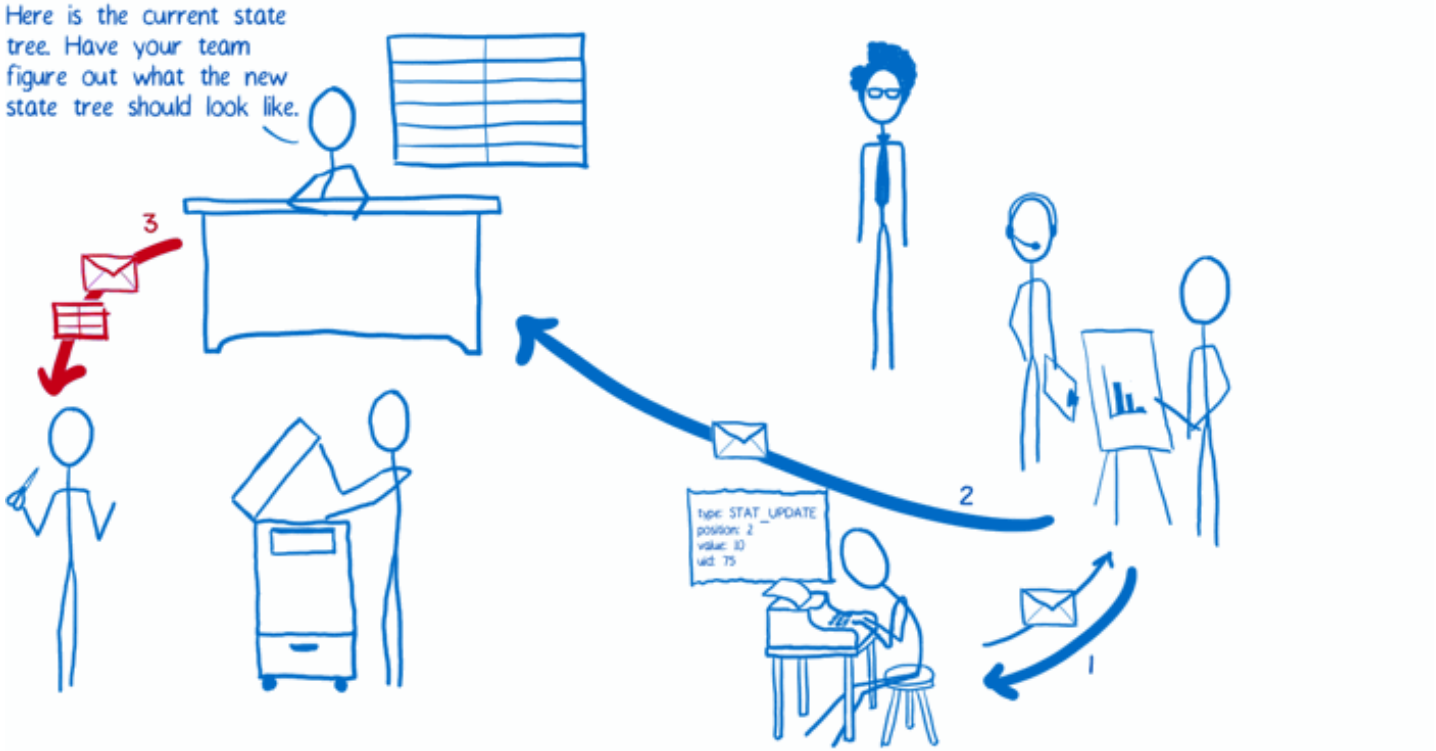
Redux: The data flow

2. View dispatches the action.



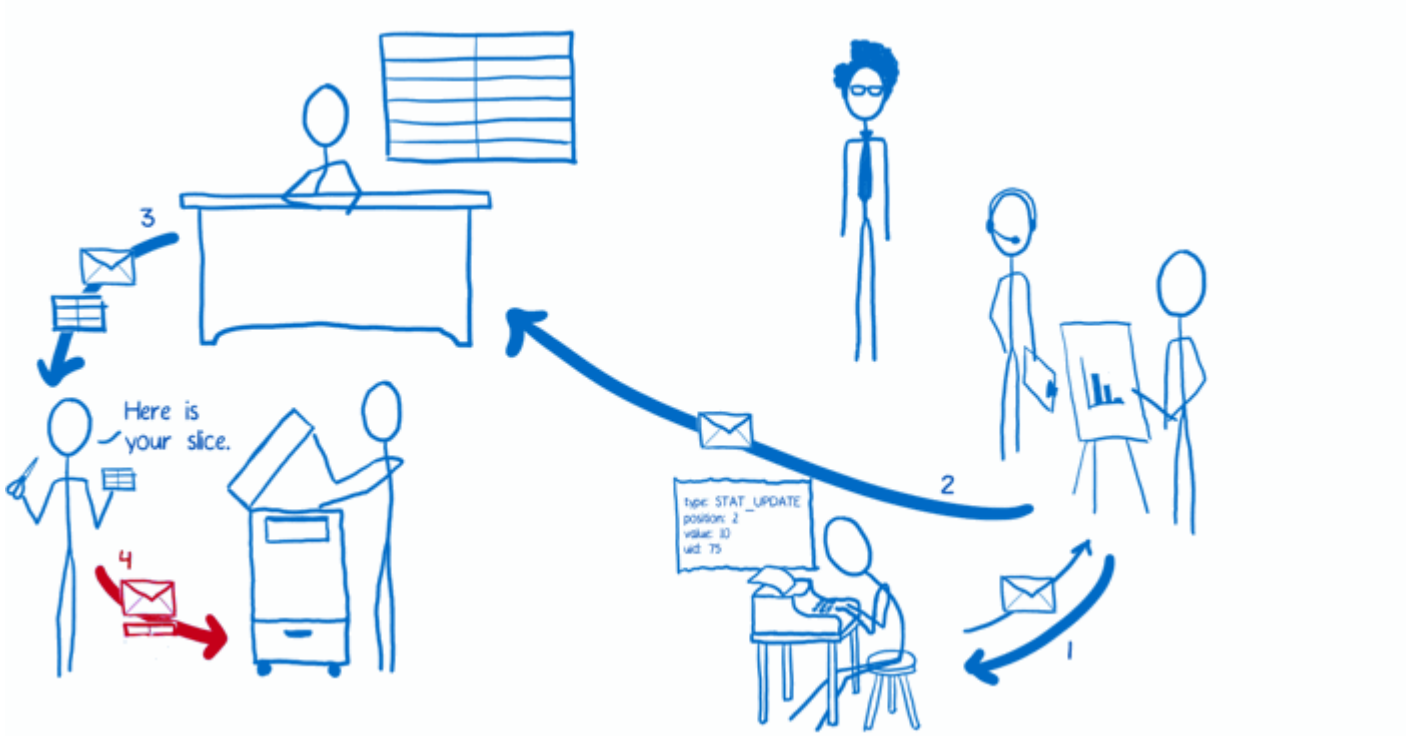
Redux: The data flow

3. The store receives the action. It sends the current state tree and the action to the root reducer.



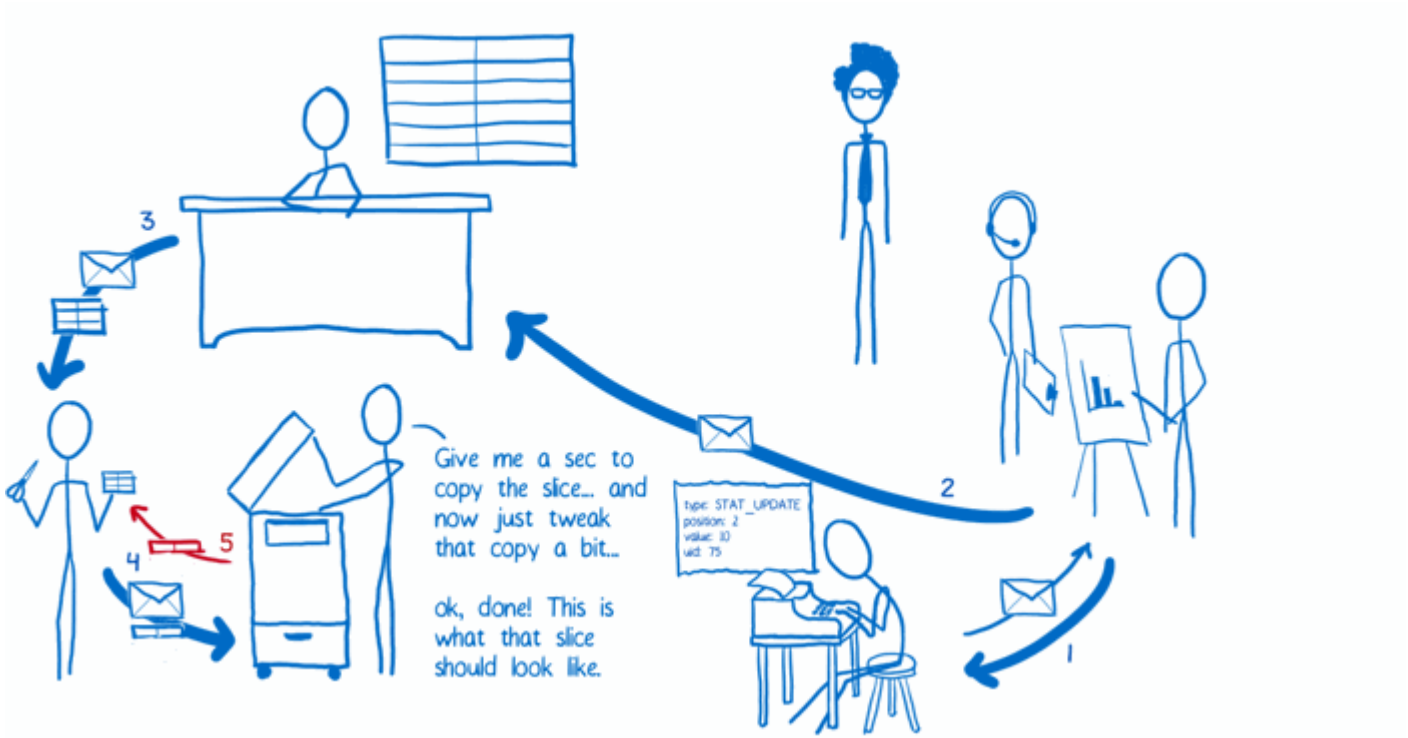
Redux: The data flow

4. The root reducer cuts apart the state tree into slices. Then it passes each slice to the sub-reducer that knows how to deal with it.



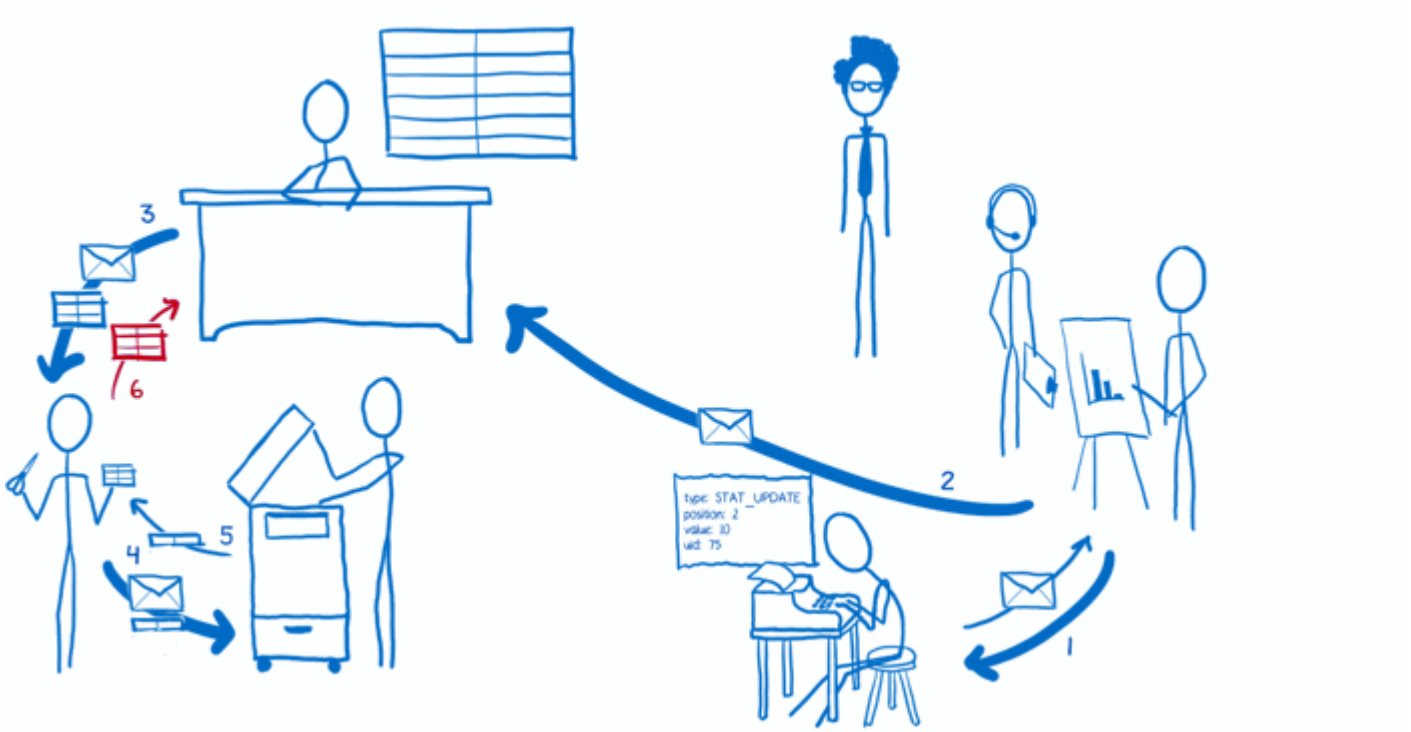
Redux: The data flow

5. The sub-reducer copies the slice and makes changes to the copy. It returns the copy of the slice to the root reducer.



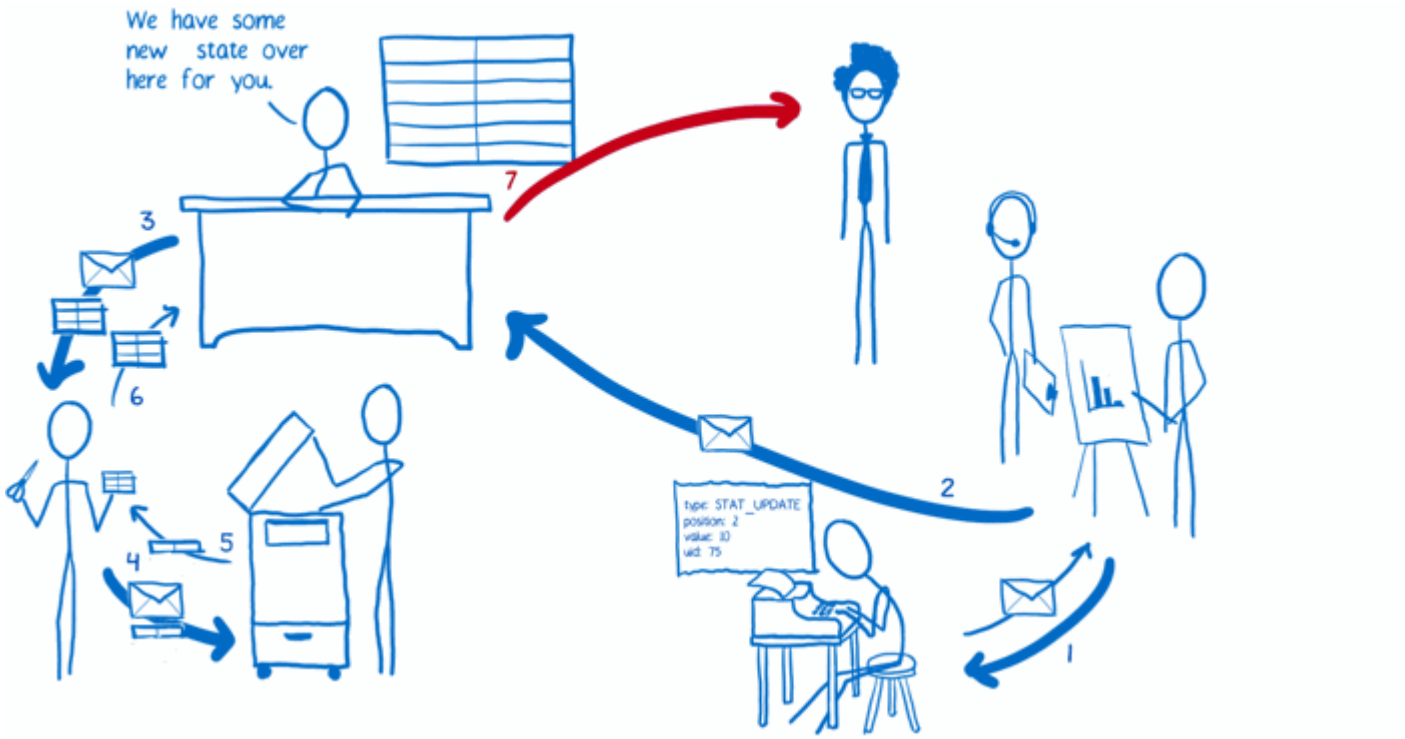
Redux: The data flow

6. Once all of the sub-reducers have returned their slice copies, the root reducer pastes all of them together to form the whole updated state tree, which it returns to the store. The store replaces the old state tree with the new one.



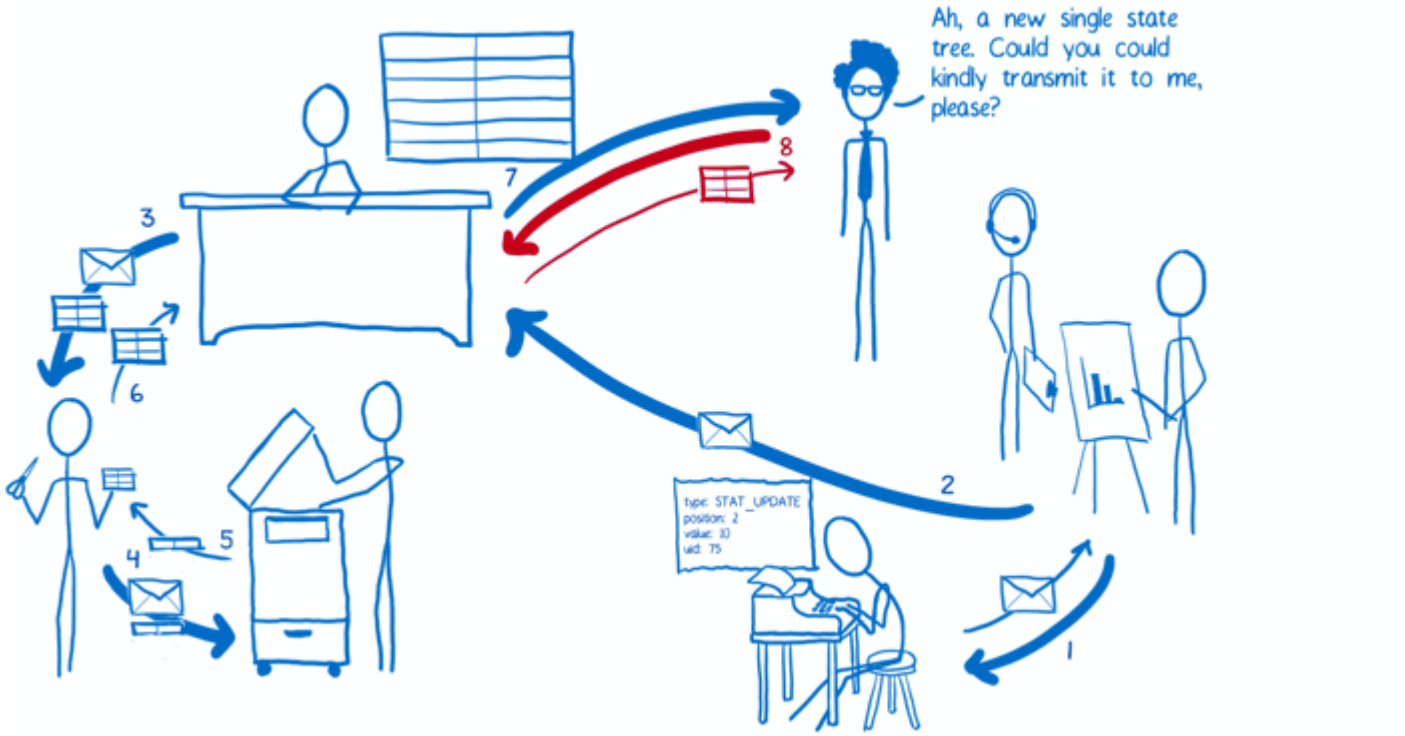
Redux: The data flow

7. The store tells the view layer binding that there’s new state.



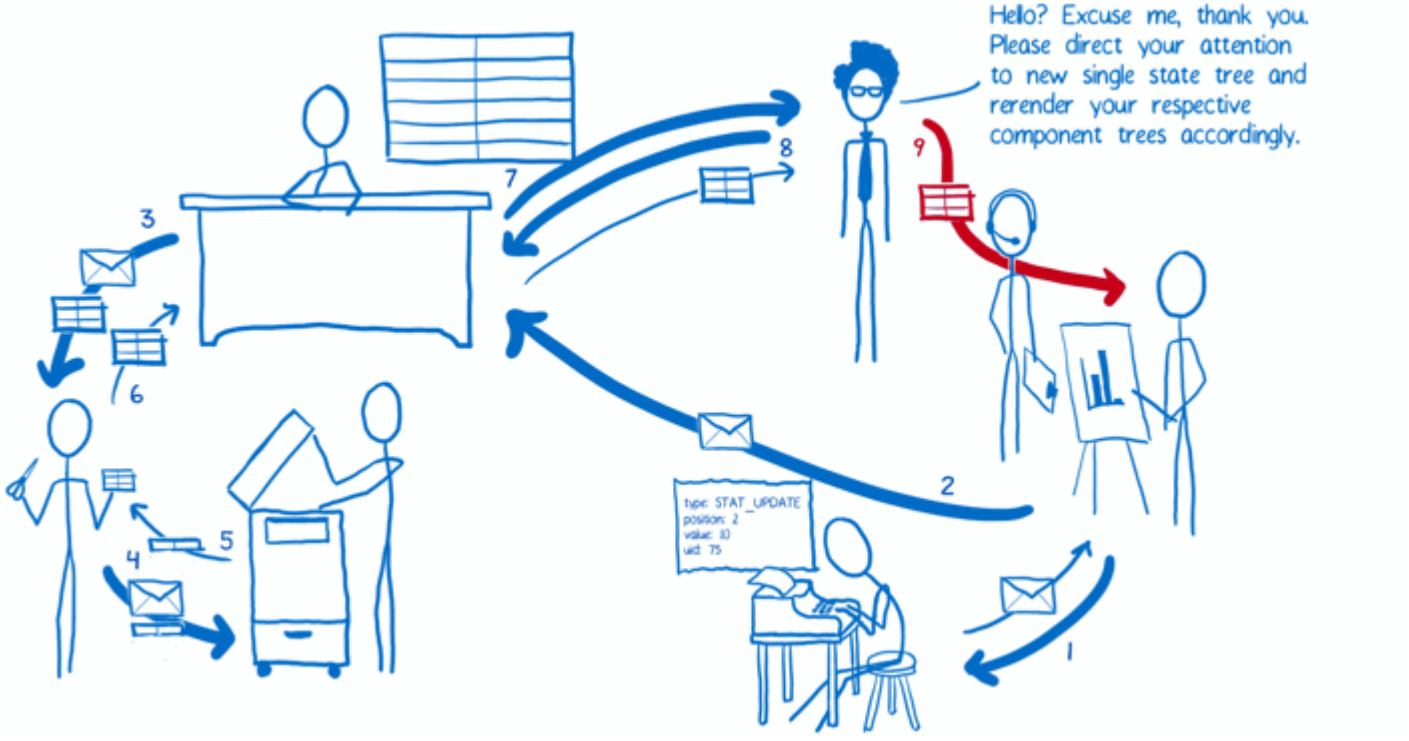
Redux: The data flow

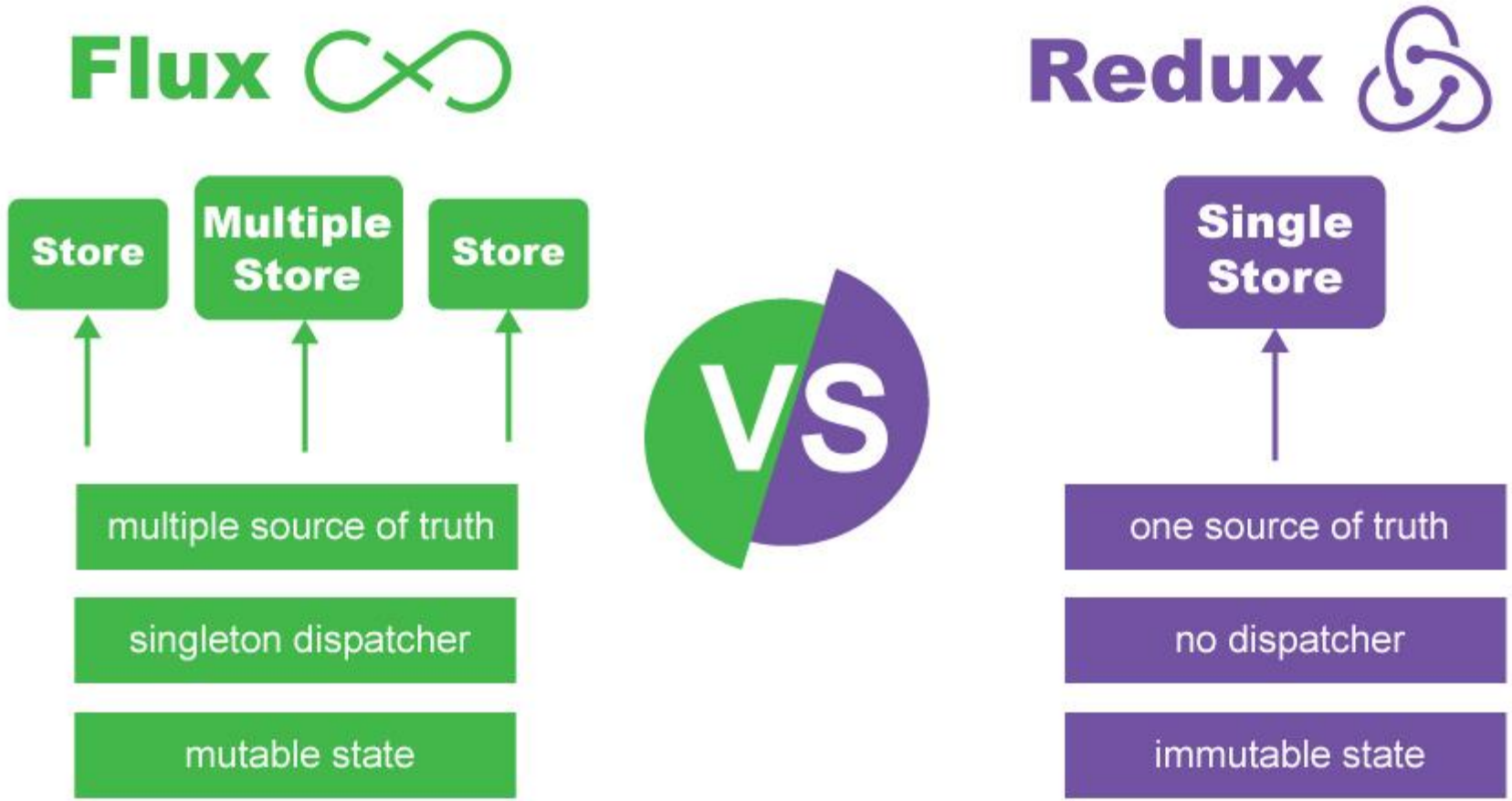
8. The view layer binding asks the store to send over the new state.



Redux: The data flow

9. The view layer binding triggers a re-render.




Flux
Store
Multiple Store
Store
multiple source of truth
singleton dispatcher
mutable state
VS
Redux
Single Store
one source of truth
no dispatcher
immutable state

Testing React / Redux Apps with Jest & Enzyme

- Dev-Dependencies package.json file:
- Enzyme is a JavaScript Testing utility for React that makes it easier to test your React Components' output.

```
"devDependencies": {
  "enzyme": "^3.11.0",
  "enzyme-adapter-react-16": "^1.15.2",
  "enzyme-to-json": "^3.4.4",
  "redux-mock-store": "^1.5.4"
}
```

Enzyme

- **Mount, Shallow, Render**
  - import { mount, shallow, render } from ‘enzyme’;
  - Mounting
    - Full DOM rendering including child components
    - Ideal for use cases where you have components that may interact with DOM API, or use React lifecycle methods in order to fully test the component
    - As it actually mounts the component in the DOM .unmount() should be called after each tests to stop tests affecting each other
    - Allows access to both props directly passed into the root component (including default props) and props passed into child components

Enzyme

- Shallow
  - Renders only the single component, not including its children. This is useful to isolate the component for pure unit testing.

```
const ButtonWithIcon = ({icon, children}) => (
    <button><Icon icon={icon} />{children}</button>
);
```

Will be rendered by React like this:

```
<button>
   <i class="icon icon_coffee"></i>
   Hello Jest!
</button>
```

shallow rendering:

```
<button>
   <Icon icon="coffee" />
   Hello Jest!
</button>
```

- Render
  - Renders to static HTML, including children
  - Does not have access to React lifecycle methods
  - Less costly than mount but provides less functionality

Enzyme Shallow example

```
function List(props) {
  const { items } = props;
  if (!items.length) {
    return <span className="empty-message">No items in list</span>;
  }
  return (
    <ul className="list-items">
      {items.map(item => <li key={item} className="item">{item}</li>)}
    </ul>
  );
}
```

```
it('renders list-items', () => {
    const items = ['one', 'two', 'three'];
    const wrapper = shallow(<List items={items} />);

    // Expect the wrapper object to be defined
    expect(wrapper.find('.list-items')).toBeDefined();
    expect(wrapper.find('.item')).toHaveLength(items.length);
    expect(wrapper.contains(<li key='two' className="item">two</li >)).toBeTruthy();
    expect(wrapper.find('.item').get(0).props.children).toEqual('one');
});
```

Enzyme mount and render example

```
function ListItem(props) {
    const { item } = props;
    return <li className="item">{item}</li>;
}

function List(props) {
  const { items } = props;
  return (
    <ul className="list-items">
      {items.map(item => <ListItem key={item} item={item} />)}
    </ul>
  );
}
```

```
it('renders list-items', () => {
    const items = ['one', 'two', 'three'];

    // Replace shallow with mount
    const wrapper = mount(<List items={items} />);
    // Expect the wrapper object to be defined
    expect(wrapper.find('.list-items')).toBeDefined();
    expect(wrapper.find('.item')).toHaveLength(items.length);
 });
```

```
it('renders list-items', () => {
  const items = ['one', 'two', 'three'];
  const wrapper = render(<List items={items} />);
    // Expect the wrapper object to be defined
  expect(wrapper.find('.list-items')).toBeDefined();
  expect(wrapper.find('.item')).toHaveLength(items.length);
});
```

Snapshot testing

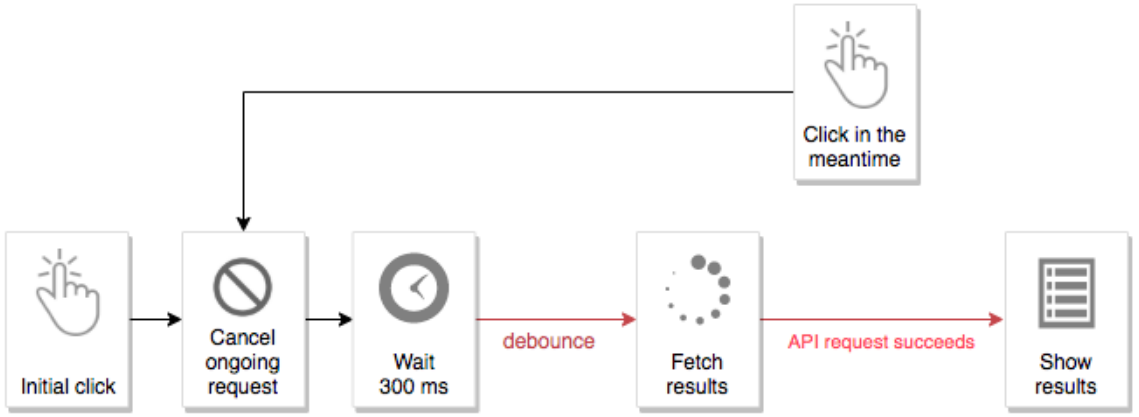
- Jest snapshots are like those old text UIs with windows and buttons made of text characters: it’s a rendered output of your component stored in a text file.

```
it('will fail every time', () => {
  const user = {
    createdAt: new Date(),
    id: Math.floor(Math.random() * 20),
    name: 'LeBron James',
  };

  expect(user).toMatchSnapshot();
});

// Snapshot
exports[`will fail every time 1`] = `
Object {
  "createdAt": 2018-05-19T23:36:09.816Z,
  "id": 3,
  "name": "LeBron James",
}
`;
```

```
it('will check the matchers and pass', () => {
  const user = {
    createdAt: new Date(),
    id: Math.floor(Math.random() * 20),
    name: 'LeBron James',
  };

  expect(user).toMatchSnapshot({
    createdAt: expect.any(Date),
    id: expect.any(Number),
  });
});

// Snapshot
exports[`will check the matchers and pass 1`] = `
Object {
  "createdAt": Any<Date>,
  "id": Any<Number>,
  "name": "LeBron James",
}
`;
```

Handling Asynchronous Actions in Redux

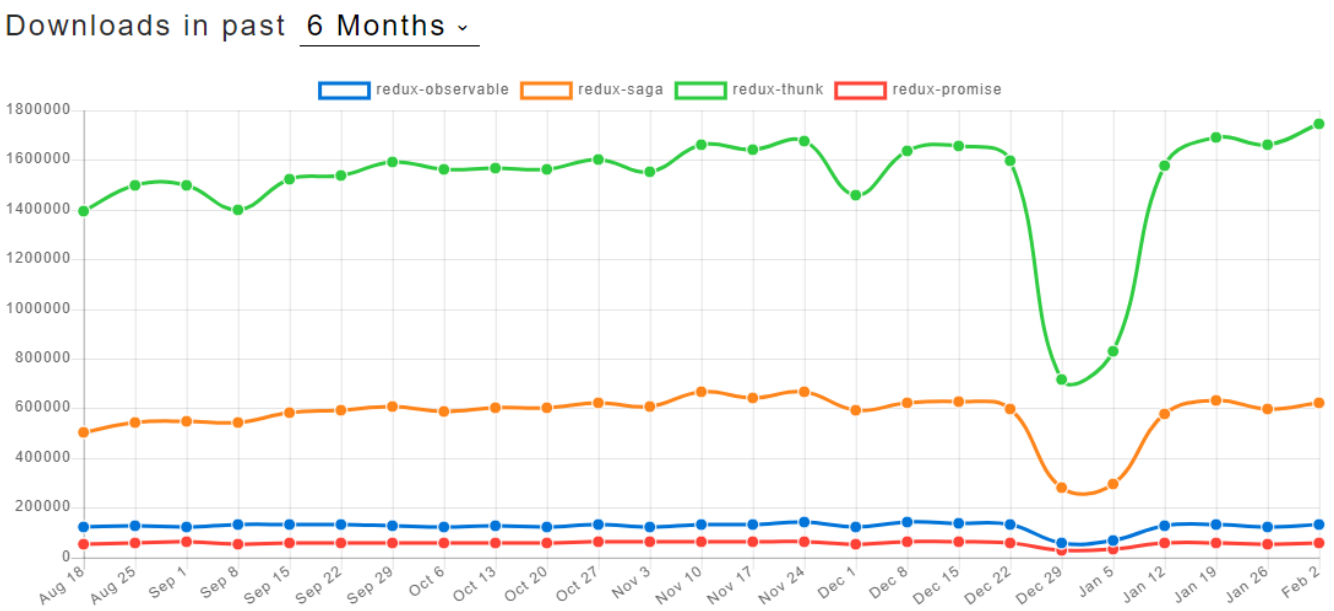
- Synchronous Redux reducer ends up not being enough, because many effects of UI actions are asynchronous, i.e.:
  - Actions such as “filtering search results” and “retrieving API response” take time and can’t be dispatched immediately,
  - Actions such as “cancel the ongoing request” need to rely on other actions (not the reducer’s state).



Actions marked as red are asynchronous

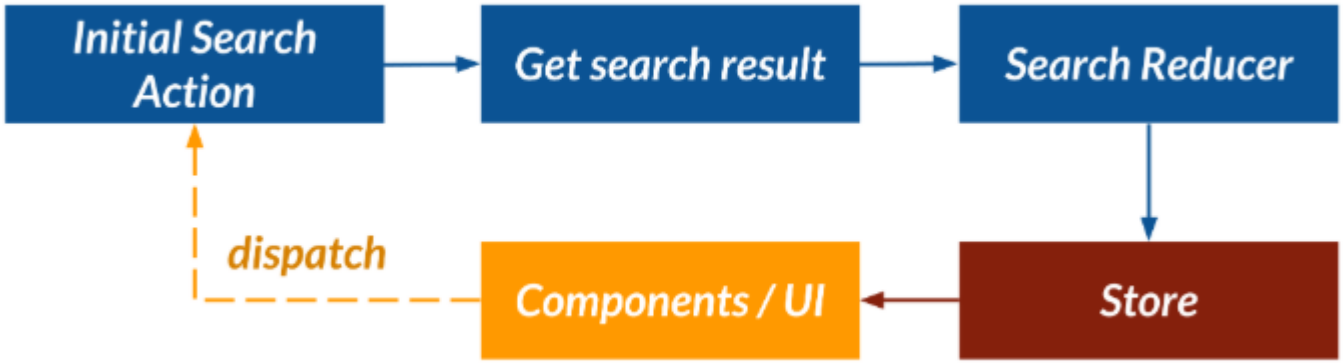
Current approaches to the problem

- An action creator that returns a function which will dispatch multiple actions over time
  - redux-thunk
  - redux-promise
- A “4th” element in the Redux store, that listens to the actions, acts on them, and dispatches another actions in the background as a result
  - using JS generators syntax and custom APIs
    - redux-saga
    - redux-ship
  - using Observables
    - redux-effects
    - redux-cycles



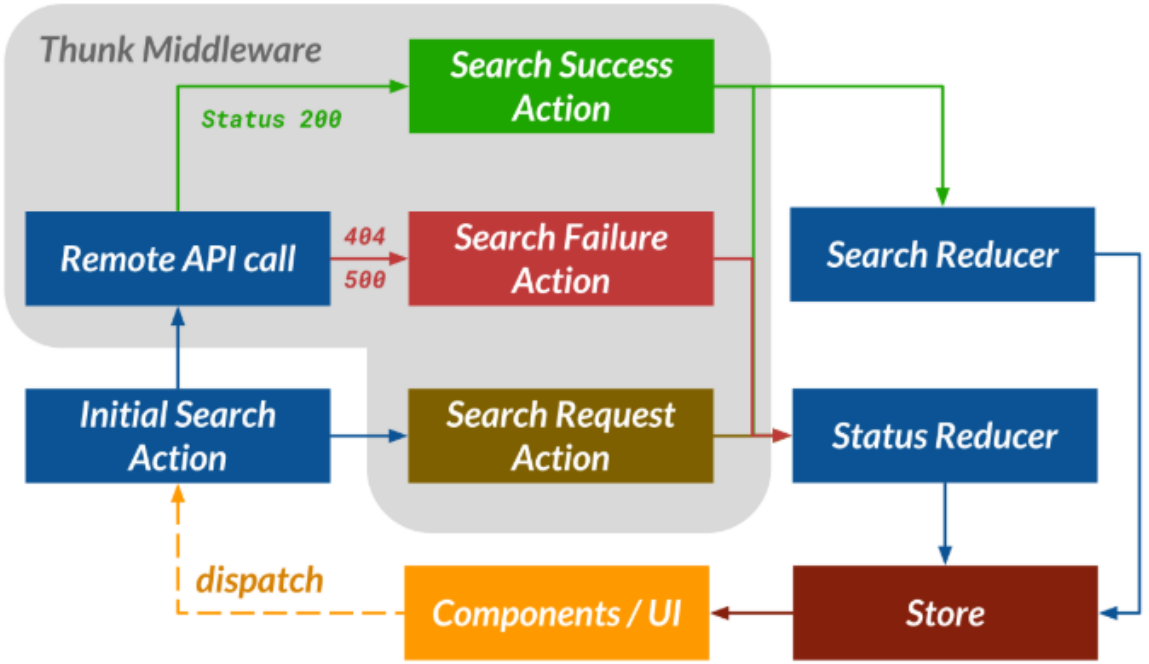
Redux way

- When a user performs a search action, we want to update the store with a new search result via **searchReducer**.
- We can then render our components accordingly. The general flow of events looks like this:



Redux Thunk way

- Make a remote API call to our backend API server.
  - When the request receives a successful JSON response, we'll dispatch the **searchSuccess** action along with the payload to **searchReducer**.
- The **statusReducer** is responsible for tracking state changes whenever a user performs an action.
  - Pending (when a user first initiates the action)
  - Success (when a request returns a successful response)
  - Error (when a request returns an error response)



Redux Thunk way: SearchAction

```
export const searchRequest = () => ({
  type: 'SEARCH_REQUEST'
});

export const searchSuccess = (result) => ({
  type: 'SEARCH_SUCCESS', result
});

export const searchFailure = (error) => ({
  type: 'SEARCH_FAILURE', error
});
```

```
export function search(title) {
  return (dispatch) => {
    dispatch(searchRequest());

    apiClient.query(title)
      .then(response => {
        dispatch(searchSuccess(response.data))
      })
      .catch(error => {
        dispatch(searchFailure(error.response.data))
      });
  }
}
```

Redux Thunk way: StatusReducer.js

```
const initialState = {
    search: '',
    searchError: ''
};
```

```
export default (state = initialState, action) => {
  const actionHandlers = {
    'SEARCH_REQUEST': {
      search: 'PENDING',
      searchError: '',
    },
    'SEARCH_SUCCESS': {
      search: 'SUCCESS',
      searchError: '',
    },
    'SEARCH_FAILURE': {
      search: 'ERROR',
      searchError: action.error,
    }
  };
  const statesToUpdate = actionHandlers[action.type];
  state = Object.assign({}, state, statesToUpdate);
  return state;
}
```

Redux Thunk way: SearchReducer.js

```
const initialState = {
    "title": "",
    "year": "",
    "plot": "",
    "poster": "",
    "imdbID": "",
  }

  export default (state = initialState, action) => {
    if (action.type === 'SEARCH_SUCCESS') {
      state = action.result;
    }
    return state;
  }
```

Redux Thunk way: store.js

```
import { createStore, combineReducers, applyMiddleware } from 'redux';
import thunk from "redux-thunk";

import search from './reducers/searchReducer';
import status from './reducers/statusReducer';

export default createStore(
  combineReducers({
    search,
    status
  }),
  {},
  applyMiddleware(thunk)
)
```

Redux Thunk way: Container component

```
const SearchContainer = (props) => (
    <main id='search-container'>
      <SearchInputForm placeholder='Search movie title...'
        onSubmit={ (title) => props.search(title) }/ >
      {
        (props.searchStatus === 'SUCCESS')
        ? <MovieItem movie={ props.result } />
        : null
      }
      {
        (props.searchStatus === 'PENDING')
        ? <section className='loading'>
            <img src='../../images/loading.gif' />
          </section>
        : null
      }
      {
        (props.searchStatus === 'ERROR')
        ? <section className='error'>
           <i className="red exclamation triangle icon"></i>
             { props.searchError }
           </section>
        : null
      }
    </main>
  );
```

```
const mapStateToProps = (state) => (
  {
    searchStatus: state.status.search,
    searchError: state.status.searchError,
    result: state.search,
  }
);

const mapDispatchToProps = (dispatch) => (
  {
    search: (title) => {
      dispatch(search(title))
    }
  }
);

export default connect(mapStateToProps, mapDispatchToProps)(SearchContainer);
```

Unit Tests for Asynchronous Redux Thunks

- Basic asynchronous login thunk example

```
import { loginRequest, loginFailure, loginSuccess } from 'actions/sessionActions';
import { login } from 'api/api';

export const loginThunk = (username, password) => async (dispatch) => {
    dispatch(loginRequest());
    try {
        const { userId, sessionId } = await login(username, password);
        dispatch(loginSuccess({ userId, sessionId }));
    }
    catch (err) {
        dispatch(loginFailure(err));
    }
};
```

Unit Tests for Asynchronous Redux Thunks

- **Mock dependencies**
  - Here we will mock the login API method:

```
// import { login } from 'api/api'; // Don't do this
const login: jest.Mock = require('api/api').login; // Do this
jest.mock('api/api', () => ({
    login: jest.fn(),
}));
```

Unit Tests for Asynchronous Redux Thunks

- Test One: Dispatching a login request

```
it('dispatches a login request', async () => {
    const dispatch = jest.fn();
    await loginThunk('username', 'password')(dispatch);
    expect(dispatch).toHaveBeenCalledWith(loginRequest());
});
```

- First we use Jest to create a spy for the dispatch function.
- loginThunk('username', 'password') returns the thunk action, which has the signature (dispatch: Dispatch) => Promise<void>. Then the test passes in the dispatch spy and waits for the promise to resolve or reject.
- Once the promise has resolved (i.e. once the thunk has completed) the test then validates that the dispatch method received a call with a loginRequest action.

Unit Tests for Asynchronous Redux Thunks

- **Test Two: When login succeeds**

```
describe('when login succeeds', () => {
    beforeEach(() => {
        login.mockResolvedValue({ userId: 'foo', sessionId: 'bar' });
    });
    it('dispatches success', async () => {
        const dispatch = jest.fn();
        await loginThunk('username', 'password')(dispatch);
        expect(dispatch).toHaveBeenLastCalledWith(loginSuccess({ userId: 'foo', sessionId: 'bar' }));
    });
});
```

Unit Tests for Asynchronous Redux Thunks

- **Test three: When login fails**

```
describe('when login fails', () => {
    const error = new Error('FAIL!');
    beforeEach(() => {
        login.mockRejectedValue(error);
    });
    it('dispatches failure', async () => {
        const dispatch = jest.fn();
        await loginThunk('username', 'password')(dispatch);
        expect(dispatch).toHaveBeenLastCalledWith(loginFailure(error));
    });
});
```

Redux Saga

- Redux-Saga is a library that aims to make application side effects (e.g., asynchronous actions such as fetching data) easier to handle and more efficient to execute.
- Saga is similar to a separate thread in your application that's solely responsible for side effects
- Unlike Redux-Thunk, which utilizes callback functions, a Redux-Saga thread can be started, paused and cancelled from the main application with normal Redux actions.
- Like Redux-Thunk, Redux-Saga has access to the full Redux application state and it can dispatch Redux actions as well.

Redux Saga

- Redux-Saga utilizes a new ES6 feature called generators.
- Generators are functions which can be exited and later re-entered.
- Calling a generator function, marked by the asterisk to the right of the function keyword, will return an iterator.
- When the iterator's next method is invoked, the generator function's body is executed up until the first yield (e.g., line 2 above).
- The iterator's next method returns an object with a value property containing the yielded value and a done boolean property, which indicates whether the generator has yielded its last value.

```
1  function* generator(i) {
2    yield i;
3    yield i + 10;
4  }
5
6  var gen = generator(10);
7
8  console.log(gen.next().value);
9  // expected output: 10
10
11 console.log(gen.next().value);
12 // expected output: 20
13
```

Redux Saga

- **When to use Redux Saga?**
  - In an application using Redux, when you fire an action something changes in the state of the app.
  - As this happens, you might need to do something that derives from this state change.
    - For example you might want to:
      - make a HTTP call to a server
      - send a WebSocket event
      - fetch some data from a GraphQL server
      - save something to the cache or browser local storage
  - Those are all things that don't really relate to the app state, or are async, and you need to move them into a place different than your actions or reducers (while you technically *could*, it's not a good way to have a clean codebase).

Redux Saga

- How it works
  - A **saga** is some “story” that reacts to an **effect** that your code is causing.
  - Create a **middleware** with a list of **sagas** to run, which can be one or more, and we connect this middleware to the Redux store.
  - The first step of saga functioning begins with the operation and execution of a promise.
  - After this cycle, the middleware suspends the saga until the time a promise is resolved.
  - After the latter step, the middleware restores the saga until the determination of the yield statement.
  - Thereafter, the saga is suspended again until the resolution of the promise. This circle continues.

Redux Saga helper functions

- Inside the saga code, you will generate effects using a few special helper functions provided by the redux-saga package. :
  - takeEvery()
  - takeLatest()
  - take()
  - call()
  - put()
- When an effect is executed, the saga is paused until the effect is fulfilled.

Redux Saga helper functions

- takeEvery(): Spawns a saga on each action dispatched to the Store that matches pattern
- The watchMessages generator pauses until an ADD_MESSAGE action fires.
- Every time it fires, it’s going to call the postMessageToServer function, infinitely, and concurrently.
- There is no need for postMessageToServer to terminate its execution before a new once can run

```
import { takeEvery } from 'redux-saga/effects'

function* watchMessages() {
  yield takeEvery('ADD_MESSAGE', postMessageToServer)
}
```

Redux Saga helper functions

- takeLatest()
  - similar to takeEvery() but only allows one function handler to run at a time, avoiding concurrency. If another action is fired when the handler is still running, it will cancel it, and run again with the latest data available.
- put()
  - Dispatches an action to the Redux store. Instead of passing in the Redux store or the dispatch action to the saga, you can just use put()

```
yield put({ type: 'INCREMENT' })
yield put({ type: "USER_FETCH_SUCCEEDED", data: data })
```

Redux Saga helper functions

- call()
  - When you want to call some function in a saga, you can do so by using a yielded plain function call that returns a promise:

  - call(delay, 1000)
    - Returns { CALL: {fn: delay, args: [1000]}}
  - Call(fetch, path)
    - Returns {CALL : {fn: fetch, args: [path]}}

Redux Saga helper functions

- **Running effects in parallel**
  - all()

```
import { call } from 'redux-saga/effects'

const todos = yield call(fetch, '/api/todos')
const user = yield call(fetch, '/api/user')
// the second fetch() call won't be executed until the first one succeeds.

// To execute them in parallel, wrap them into all():

import { all, call } from 'redux-saga/effects'

const [todos, user]  = yield all([
  call(fetch, '/api/todos'),
  call(fetch, '/api/user')
])
```

  - all() won't be resolved until both call() return

Redux Saga helper functions

- **Running effects in parallel**
  - **race( )**
    - It’s a race to see which one finishes first, and then we forget about the other participants.
  - It’s typically used to cancel a background task that runs forever until somethin g occurs:

```
function* someBackgroundTask() {
  while(1) {
    //...
  }
}

yield race([
  bgTask: call(someBackgroundTask),
  cancel: take('CANCEL_TASK')
])
```

Redux-Thunk Vs Redux-Saga

```
export function receiveBooks(data) {
  return {
    type: types.RECEIVE_BOOKS,
    books: data.books,
    categories: data.categories,
    genres: data.genres
  };
}

export function fetchBooks() {
  return dispatch => {
    fetch("/books.json").then(response => {
        const data = response.json();
        dispatch(receiveBooks(data));
    })
    .catch(error =>
        dispatch({ type: types.FETCH_FAILED, error })
    );
  };
}
```

```
import { takeLatest } from "redux-saga"
import { call, put } from "redux-saga/effects"

function* fetchBooks(path) {
    try {
        const data = yield call(fetch, path);
        yield put({type: "RECEIVE_BOOKS", data });
    } catch (e) {
        yield put({type: "FETCH_FAILED", message: e.message});
    }
}

function* fetchSaga() {
  yield* takeLatest("FETCH_BOOKS", fetchBooks);
}

export default fetchSaga;
```

Higher-Order Component

- **Higher-order component** is a function that **takes (wraps) a component and returns a new component**
- They are a pattern that stems from React's nature that privileges composition over inheritance
  - **Types of Higher-Order Components**
    - **Props Proxy (ppHOC)**
    - **Inheritance Inversion (iiHOC)**

HOC: **Props Proxy (ppHOC)**

- Props Poxy takes a Component as an argument and returns a new component.
  - Props Proxy HOCs are useful to the following situations:
    - Manipulating props
    - Abstracting State
    - Wrapping/Composing the WrappedComponent with other elements

```
const propsProxyHOC = (WrappedComponent) => {
    return class extends React.Component {
        render() {
            return <WrappedComponent {...this.props} />
        }
    }
}
```

HOC: **Inheritance Inversion (iiHOC)**

- Inheritance Inversion gives the HOC access to the WrappedComponent instance, which means you can use the state, props, component lifecycle and **even the render method**.
- Inversion Inheritance HOCs are useful for the following situations:
  - Render High-jacking
  - Manipulating state

```
const iiHOC = (WrappedComponent) => {
    return class extends WrappedComponent {
        render() {
            return super.render();
        }
    }
}
```

HOC example

```
class Welcome extends React.Component {
    render() {
        return(
            <div> Welcome {this.props.user} </div>
        )
    }
}

const withUser = (WrappedComponent) =>{
    return class extends React.Component {
        render() {
            if(this.props.user) {
                return (
                    <WrappedComponent {...this.props} />
                )
            } else {
                <div>Welcome Guest!</div>
            }
        }
    }
}
```

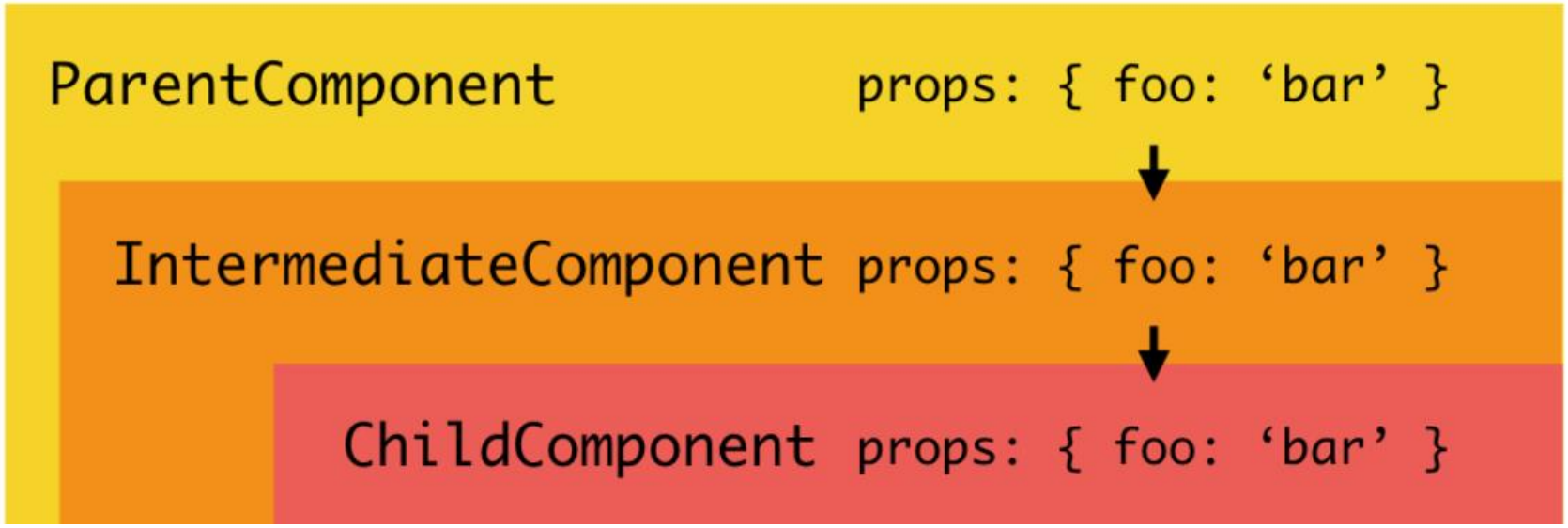
```
const withLoader = (WrappedComponent) =>{
    return class extends WrappedComponent {
        render() {
            const {isLoaded} = this.props;
            if(!isLoaded) {
                return <div>Loading...</div>
            }
            return super.render();
        }
    }
}

export default withLoader(withUser(Welcome));

usage: <Welcome isLoaded={false} user="Banuprakash"/>
```

HOC: use case

- **You may have a good use-case for HOCs if:**
  - For cross-cutting concerns
  - The behavior is **not specific to any single component,** but rather **applies to many or all components in the app**
  - **Components can be used stand-alone** without the behavior from the HOC.
  - **No custom logic needs to be added to a component being wrapped by the HOC**.

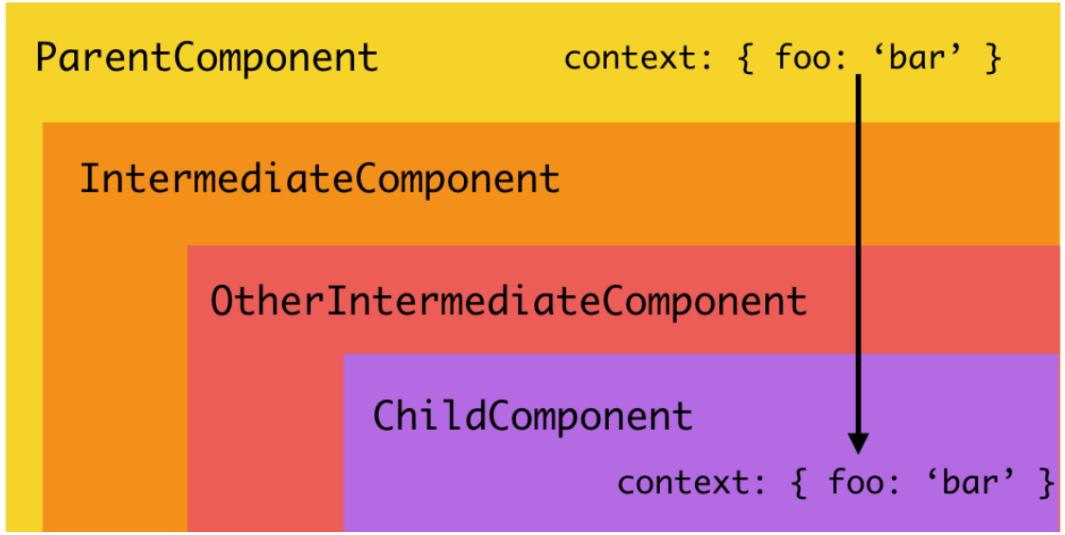
Props

- In React the primary mechanism for communication between your components is through properties, or props, for short



Context

- Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language
- Context acts like a portal in your application in which components can make data available to other components further down the tree without being passed through explicitly as props



Context: Create provider and consumer components

```
import React, { createContext } from 'react';

const UserContext = createContext({
  username: '',
  updateUsername: () => {},
});

export class UserProvider extends React.Component {
  updateUsername = newUsername => {
    this.setState({ username: newUsername });
  };

  state = {
    username: 'user',
    updateUsername: this.updateUsername,
  };
```

```
  render() {
    return (
      <UserContext.Provider value={this.state}>
        {this.props.children}
      </UserContext.Provider>
    );
  }
}

export const UserConsumer = UserContext.Consumer;
```

Context: using the Provider

- The <UserProvider> component needs to wrap around all components that share state.

```
function App() {
  return (
    <UserProvider>
      <UserMessage />
      <SettingsForm />
    </UserProvider>
  );
}
```

Context: Using the consumer to read state

```
export default function UserMessage() {
  return (
    <UserConsumer>
      {({ username }) => <h1>Welcome {username}!</h1>}
    </UserConsumer>
  );
}
```

Context: Using the consumer to update the state

```
export default function UserSettings() {
  return (
    <UserConsumer>
      {({ updateUsername }) => (
        <div>
          <h2>Settings</h2>
          <label htmlFor="username">Username: </label>
          <input
            id="username"
            type="text"
            onChange={event => {
              updateUsername(event.target.value);
            }}
          />
        </div>
      )}
    </UserConsumer>
  );
}
```

React Hooks

- *Hooks* are a new addition in React 16.8. They let you use state and other React features without writing a class.
- **React's State Hook**
  - **Here, useState is a Hook. We call it inside a function component to add some local state to it. React will preserve this state between re-renders. useState returns a pair: the current state value and a function that lets you update it.**

```
class Counter extends Component {
    state = {
      count: 0
    };
    setCount = () => {
      this.setState({ count: this.state.count + 1 });
    };

    render() {
      return (
        <div>
          <h1>{this.state.count}</h1>
            <button onClick={this.setCount}>Count Up</button>
        </div>
      );
    }
}
```

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>Count Up</button>
    </div>
  );
}
```

React Hooks

- **Using Multiple State Hooks**
  - We can even use useState() multiple times in the same function.

```
import React, { useState } from 'react';

function AllTheThings() {
  const [count, setCount] = useState(0);
  const [products, setProducts] = useState([{ name: 'Surfboard', price: 100 }]);
  const [coupon, setCoupon] = useState(null);

  return <div>{/_ use all those things here _/}</div>;
}
```

React’s Effect Hook

- Effects are similar to componentDidMount, componentDidUpdate, and componentWillUnmount.
  - This is what the Effect Hook is for. Side-effects are things you want your application to make like:
    - Fetching data
    - Manually changing the DOM (document title)
    - Setting up a subscription
  - Effects will run after every render, including the first render.

React’s Effect Hook

- useEffect similar to componentDidMount and componentDidUpdate.

```
class DoSomething extends Component {
    componentDidMount() {
      console.log('i have arrived at the party!');
      document.title = 'present';
    }

    render() {
      return <div>stuff goes here</div>;
    }
  }
```

```
function DoSomething() {
    useEffect(() => {
      console.log('i have arrived at the party!');
      document.title = 'present';
    });

    return <div>stuff goes here</div>;
  }
```

React’s Effect Hook

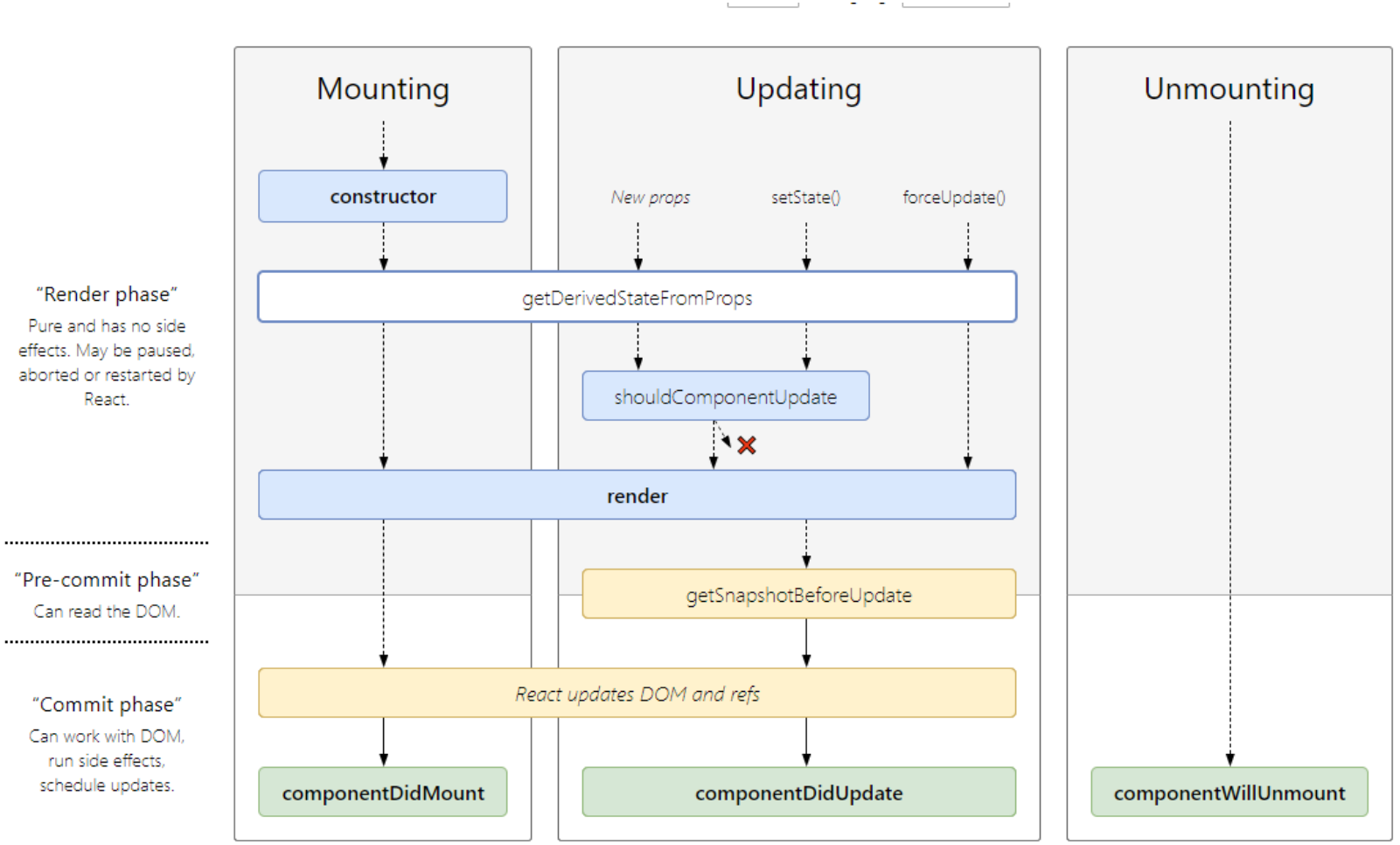
componentDidMount: Runs once

```
// only run on mount. pass an empty array
useEffect(() => {
  // only runs once
}, []);
```

componentDidUpdate: Runs on changes

```
// only run if count changes
useEffect(
  () => {
    // run here if count changes
  },
  [count]
);
```

React Performance



React Performance

- *shouldComponentUpdate*
  - The *shouldComponentUpdate* method is called every time a component receives new props or changes its state and is one of the primary ways to optimize your React app's performance.
  - *shouldComponentUpdate* receives the next state and props of the Component, which we can use to compare to the current state of the Component and then return a *boolean* whether we want to re-render or not.

```
class ProfileSection extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return this.props.id !== nextProps.id && nextState.visible;
  }  render() {
    // ...
  }
}
```

React Performance

- **PureComponent**
  - The React team determined that in most cases people just simply compared if the props or state have changed in order to trigger the re-render.
  - This is why they created React's *PureComponent.*

```
import React, {PureComponent} from 'react';
class ProfileSection extends PureComponent {
  render() {
    // will only happen if props and state are different
    //...
  }
}
```

React Performance

- **React memo**
  - React team released *React.memo*, which means that you no longer need to refactor your functional Component to a class Component in order to take advantage of this functionality

```
import React, {memo} from 'react';

const ProfileSection = props => (
  // ...
);
export default memo(ProfileSection);
```

React Performance

- **Binding in Constructor vs Arrow Functions**
  - Regular functions are added to the Component's *prototype* while Arrow functions become an instance property.

```
class Button extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log(this.props);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click Me
      </button>
    );
  }
}
```

```
class Button extends React.Component {
  handleClick = () => {
    console.log(this.props);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click Me
      </button>
    );
  }
}
```

React Performance

- Avoid binding/inlining when rendering
  - if you perform some kind of inlining or binding in render you are always triggering a re-render.

```
class Button extends React.Component {
  handleClick() {
    console.log(this.props);
  }

  render() {
    return (
      <button onClick={this.handleClick.bind(this)}>
        Click Me
      </button>
    );
  }
}
```

```
function Button() {
  return (
    <button onClick={() => console.log(this.props)}>
      Click Me
    </button>
  );
}
```

React Performance

- **Not using (proper) keys**
  - React needs a key prop when rendering lists so that it knows which elements have been added, removed or changed.
  - This improves the rendering performance of the list because if something changes in the absence of keys React will re-render the entire list and you can imagine the cost it bears when rendering a big list.
  - Using indexes as keys is also a bad practice because then your list becomes dependent on the order of the elements, which leads to bugs if you attempt to sort the list or add new elements to it.

```
things.map((thing, index) => (
  <Hello key={index} />
));
```

```
things.map((thing) => (
  <Hello key={thing.id} />
));
```

React Performance

- Use React.Fragments to Avoid Additional HTML Element Wrappers

```
class Comments extends React.PureComponent{
  render() {
    return (
      <React.Fragment>
        <h1>Comment Title</h1>
        <p>comments</p>
        <p>comment time</p>
      </React.Fragment>
    );
  }
}
```

OR

```
class Comments extends React.PureComponent{
  render() {
    return (
      <>
        <h1>Comment Title</h1>
        <p>comments</p>
        <p>comment time</p>
      </>
    );
  }
}
```

React Performance

- **Debouncing Event Action in JavaScript**
  - debouncing is a technique to prevent the event trigger from being fired too often.

```
import debouce from 'lodash.debounce';

class SearchComments extends React.Component {

 setSearchQuery = debounce(e => {
    // Fire API call or Comments manipulation on client end side
 }, 1000);

 render() {
   return (
     <div>
       <h1>Search Comments</h1>
       <input type="text" onChange={this.setSearchQuery} />
     </div>
   );
 }
}
```

React Performance

- Avoid Async Initialization in componentWillMount()
  - componentWillMount() is only called once and before the initial render.
  - Since this method is called before render(), our component will not have access to the refs and DOM element. [Use componentDidMount instead]
  - The componentWillMount() is good for handling component configurations and performing synchronous calculation based on props since props and state are defined during this lifecycle method.

React Performance

- **Using Web Workers for CPU Extensive Tasks**
  - What happens when we have 20,000 posts? It will slow down the rendering since the sort method will run in the same thread.

```
// Sort Service for sort post by the number of comments
function sort(posts) {
  for (let index = 0, len = posts.length - 1; index < len; index++) {
      for (let count = index+1; count < posts.length; count++) {
          if (posts[index].commentCount > posts[count].commentCount) {
              const temp = posts[index];
              posts[index] = users[count];
              posts[count] = temp;
          }
      }
  }
  return posts;
};
```

```
export default class Posts extends React.Component{
  state = {
      posts: this.props.posts
  }

  doSortingByComment = () => {
      if(this.state.posts && this.state.posts.length){
          const sortedPosts = sort(this.state.posts);
          this.setState({
              posts: sortedPosts
          });
      }
  }

  render(){
      const posts = this.state.posts;
      return (
          <React.Fragment>
              <Button onClick={this.doSortingByComment}>
                  Sort By Comments
              </Button>
              <PostList posts={posts}></PostList>
          </React.Fragment>
      )
  }
}
```

React Performance using Web Worker

```
// sort.worker.js

// In-Place Sort function for sort post by number of comments
export default  function sort() {

  self.addEventListener('message', e =>{
      if (!e) return;
      let posts = e.data;

      for (let index = 0, len = posts.length - 1; index < len; index++) {
          for (let count = index+1; count < posts.length; count++) {
              if (posts[index].commentCount > posts[count].commentCount) {
                  const temp = posts[index];
                  posts[index] = users[count];
                  posts[count] = temp;
              }
          }
      }
      postMessage(posts);
  });
}
```

React Performance using Web Worker

- Use Web Workers for tasks like image processing, sorting, filtering, and other CPU extensive tasks.

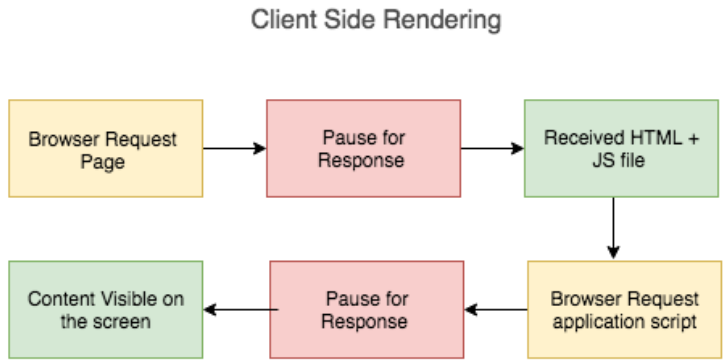
```
export default class Posts extends React.Component{
  state = {
      posts: this.props.posts
  }
  componentDidMount() {
      this.worker = new Worker('sort.worker.js');

      this.worker.addEventListener('message', event => {
          const sortedPosts = event.data;
          this.setState({
              posts: sortedPosts
          })
      });
  }
  doSortingByComment = () => {
      if(this.state.posts && this.state.posts.length){
          this.worker.postMessage(this.state.posts);
      }
  }

  render(){
      const posts = this.state.posts;
      return (
          <React.Fragment>
              <Button onClick={this.doSortingByComment}>
                  Sort By Comments
              </Button>
              <PostList posts={posts}></PostList>
          </React.Fragment>
      )
  }
}
```

React Performance

- **Consider Server-side Rendering**
  - Server-side rendering provides performance benefit and consistent SEO performance
  - React app page source without server-side rendering

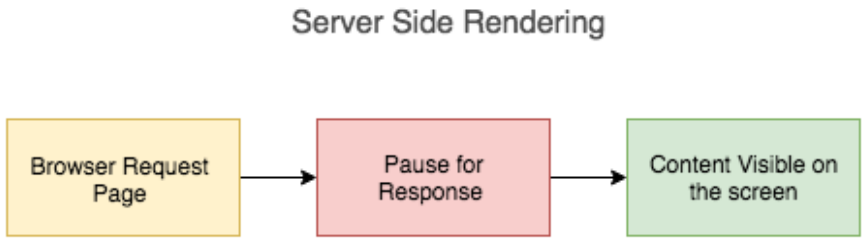
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link rel="shortcut icon" href="/favicon.ico">
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="/app.js"></script>
  </body>
</html>
```

The browser will also fetch the app.js bundle, which contains the application code and render the full page after a second or two.



React Performance

- **Consider Server-side Rendering**
  - The same app with server-side rendering enabled:
    - We see that only one trip to the server happens before the users get their content.
    - When the browser requests a page, the server loads React in the memory and fetches the data required to render the app.
    - After that, the server sends generated HTML to the browser, which is immediately shown to the user. [Time to First Meaningful Paint]
  - Some popular solutions that provides SSR for React apps:
    - Next.js
    - Gatsby



React Performance

- Using next.js [npm i --save next]

```
import Link from 'next/link';

const linkStyle = {
  marginRight: 15
};

const Header = () => (
  <div>
    <Link href="/">
      <a style={linkStyle}>Home</a>
    </Link>
    <Link href="/about">
      <a style={linkStyle}>About</a>
    </Link>
  </div>
);

export default Header;
```

```
"scripts": {
  "dev": "next",
  "build": "next build",
  "start": "next start"
}
```

```
import Header from '../components/Header';

export default function Index() {
  return (
    <div>
      <Header />
      <p>Hello Next.js</p>
    </div>
  );
}
```