# React Redux POC for UI Insurance App

This application demonstrates what a React.js based register/login workflow might look like on the Frontend. I used my react-boilerplate as a starting point — the app thus uses Redux, Foundation, react-router, ServiceWorker, AppCache, bcrypt and lots more.

The default username is `kkdrensk` and the default password is `kkdrensk`, but feel free to register new users! The registered users are saved to localStorage, so they'll persist across page reloads.

## Table of Contents

# Purpose of this document

This Document is intended to do a summary about the conclusions extracted from the Proof of Concept (POC) of OcInfra implemented in React/redux.

# Why use REACT and REDUX.

## React

Mean reasons for building a modern web app with **REACT** can be:

- **Makes writing Js easier**. It uses a special syntax called JSX, which allows you mix HTML with Js. This is not mandatory, you can write in plain Js but i suggest that you try it because it's easier for writing components.
- **Components are the future of the web development**.React doesn't use Shadow DOM – instead it gives you the ability to create your own components that you can later reuse, combine, and nest to your heart’s content. I've found this to be the single-biggest productivity boost because it’s so easy to define and manipulate your own components.
- **React is extremely efficient**.React creates its own virtual DOM where your components actually live. This approach gives you enormous flexibility and amazing gains in performance because React calculates what changes need to be made in the DOM beforehand and updates the DOM tree accordingly. This way, React avoids costly DOM operations and makes updates in a very efficient manner.
- **It's awesome for SEO**.One of the biggest issues with JavaScript frameworks is that they are not exactly search engine friendly. Although

there have been some improvements in this area, search engines generally have trouble reading JavaScript-heavy applications.React stands out from the crowd because you can run React on the server, and the virtual DOM will be rendered and returned to the browser as a regular web page. No need for PhantomJS and other tricks!

- **It gives you out-of-the-box developer tools**.When you start your adventure with React, don't forget to install the official React Chrome extension. It makes debugging your app so much easier.After you install the extension, you'll have a direct look into the virtual DOM just as if you were browsing a regular DOM tree in the elements panel. Pretty amazing!
- **Facebook are maintaining this project**. React is now open source, but it was first developed at Facebook for internal purposes. After a while, Facebook engineers realized that they created something truly awesome and decided to share their project with the world.Facebook uses some React, and Instagram’s entire website was built on React after the two companies joined forces. Other successful projects using React include AirBnb, Khan Academy and New York Times.
- **What about mobility? React Native**.Once you get comfortable with building web application with React, you can easily switch to building mobile application using React Native. Though it is not directly related to React, React Native follows same design patterns, making the transition easier! Using only Javascript, you will be able to build the native equivalent to Java, Swift or Objective-C, thus supported both by Android and iOS.

Unfortunately this technology is a library for the presentation of the web layer but in this poc we combine react (View library) with redux (model and controller).

## Redux

is a complimentary library to React that provides a way to easily keep the data(State) and the events(Actions).Summing up is the M and C in a MVC pattern tech. Now , it can compete with other web FWK like angular2

To give some perspective, let’s take the classic model-view-controller (MVC) pattern, since most developers are familiar with it. In MVC architecture, there is a clear separation between data (model), presentation (view) and logic (controller). There is one issue with this, especially in large-scale applications: The flow of data is bidirectional. This means that one change (a user input or API response) can affect the state of an application in many places in the code — for example, two-way data binding. That can be hard to maintain and debug. Flux is very similar to Redux. The main difference is that Flux has multiple stores that change the state of the application, and it broadcasts these changes as events. Components can subscribe to these events to sync with the current state. Redux doesn’t have a dispatcher, which in Flux is used to broadcast payloads to registered callbacks. Another difference in Flux is that many varieties are available, and that creates some confusion and inconsistency.

these are a few benefits of using Redux in this POC.

- **Predictability of outcome**.There is always one source of truth, the store, with no confusion about how to sync the current state with actions and other parts of the application.
- **Maintainability**.Having a predictable outcome and strict structure makes the code easier to maintain.
- **Organization**.Redux is stricter about how code should be organized, which makes code more consistent and easier for a team to work with.
- **Server rendering**.This is very useful, especially for the initial render, making for a better user experience or search engine optimization. Just pass the store created on the server to the client side.
- **Developer tools**.Developers can track everything going on in the app

in real time, from actions to state changes.
- **Community and ecosystem**.This is a huge plus whenever you're learning or using any library or framework. Having a community behind Redux makes it even more appealing to use.
- **Ease of testing**.The first rule of writing testable code is to write small functions that do only one thing and that are independent. Redux's code is mostly functions that are just that: small, pure and isolated.

**Redux** was built on top of **functional programming** concepts. Understanding these concepts is very important to understanding how and why Redux works the way it does. Let's review the fundamental concepts of functional programming:

- It is able to treat functions as first-class objects.
- It is able to pass functions as arguments.
- It is able to control flow using functions, recursions and arrays.
- It is able to use pure, recursive, higher-order, closure and anonymous functions.
- It is able to use helper functions, such as map, filter and reduce.
- It is able to chain functions together. The state doesn't change (i.e. it's immutable). The order of code execution is not important.

Functional programming allows us to write cleaner and more modular code. By writing smaller and simpler functions that are isolated in scope and logic, we can make code much easier to test, maintain and debug. Now these smaller functions become reusable code, and that allows you to write less code, and less code is a good thing. The functions can be copied and pasted anywhere without any modification. Functions that are isolated in scope and that perform only one task will depend less on other modules in an app, and this reduced coupling is another benefit of functional programming.

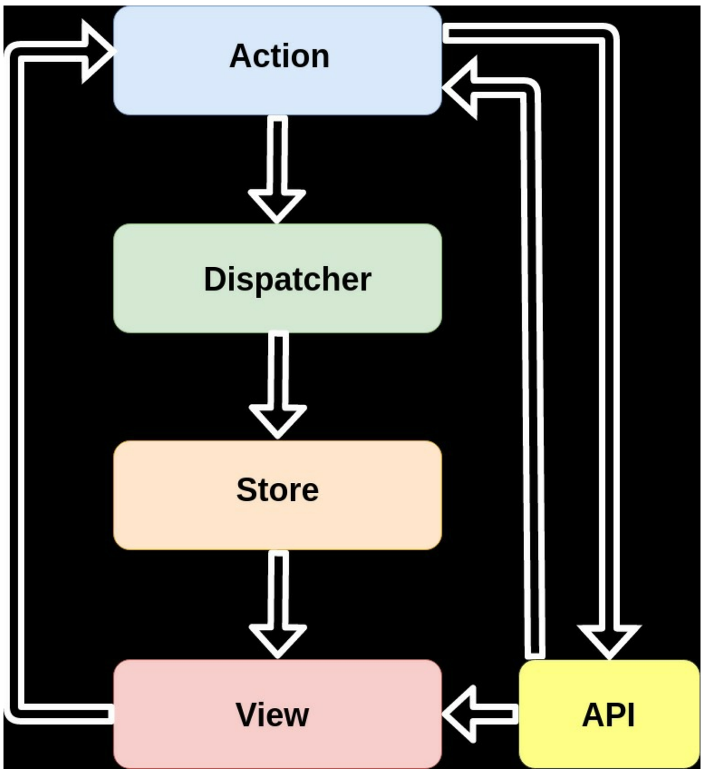
You will see pure functions, anonymous functions, closures, higher-order functions and method chains, among other things, very often when working with functional JavaScript. Redux uses **pure functions** heavily, so it’s important to understand what they are. Pure functions return a new value based on arguments passed to them. They don’t modify existing objects; instead, they return a new one. These functions don’t rely on the state they’re called from, and they return only one and the same result for any provided argument. For this reason, they are very predictable. Because pure functions don’t modify any values, they don’t have any impact on the scope or any observable side effects, and that means a developer can focus only on the values that the pure function returns.

## Overview

Our arquitecture application is going to be structured following react redux principles of a predictable state management with the following entities and it will explained how it works among them.

These entities are:

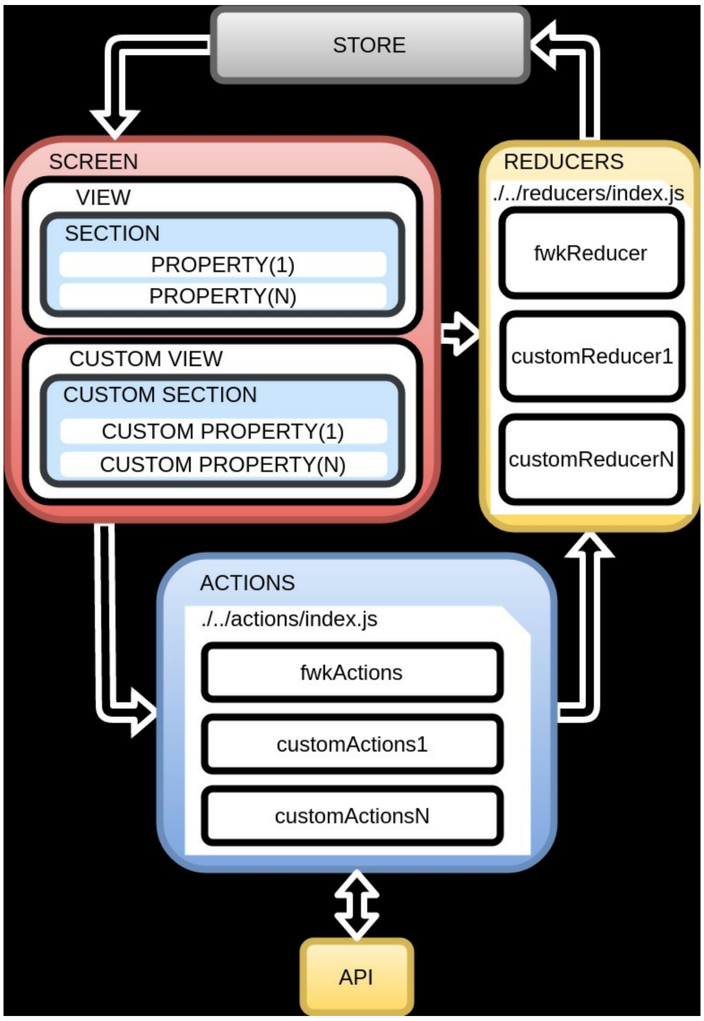
- Action
- Dispatcher
- store
- View
- API



## High Level design

Redux architecture revolves around a strict unidirectional data flow.

All data in an application follows the same lifecycle pattern, making the logic of your app more predictable and easier to understand. It also encourages data normalization, so that you don't end up with multiple, independent copies of the same data that are unaware of one another.



The data lifecycle in any Redux app follows these 4 steps:

- From View, You call store.dispatch(action).
- Store calls the reducer function (Action) you gave it.
- Root reducer may combine the output of multiple reducers into a single state tree
- Store saves the complete state tree returned by the root reducer.

Source

## Components/Layers and responsibilities

### Actions

Actions are payloads of information that send data from your application (View component) to your store (Logic layer). They are the only source of information for the store. You send them to the store using `store.dispatch()`.

For exmaple:

```
/**
 * Sets the metamodel state object given a screenId
 * @param {string} screenId Screen param to get metamodel obj
 */
export function loadMetamodel(screenId) {
  return { type: SET_METAMODEL_OBJECT, screenId};
}
```

Source

## Reducers (Dispatcher)

Actions describe the fact that something happened, but don't specify how the application's state changes in response. This is the job of reducers.

In Redux, all the application state is stored as a single object. It's a good idea to think of its shape before writing any code. What's the minimal representation of your app's state as an object?

In this prrof of concept, our state object is defined like:

```
// The initial application state
const initialState = {
  formState: {
    username: '',
    password: ''
  },
  metadata: null, //
  viewdata: null, // Data
  properties: null,
  currentlySending: false,
  loggedIn: auth.loggedIn(),
  errorMessage: ''
};
```

You'll often find that you need to store some data, as well as some UI state, in the state tree. This is fine, but try to keep the data separate from the UI state.

```
// Piece of case in fwkReducer for handleing METAMODEL OBJECT
export function fwkReducer(state = initialState, action) {
  switch (action.type) {
      case SET_METAMODEL_OBJECT:
        var screenJsonModel = require('./../../metamodel/' +
        return assign({},state, {
          metadata: screenJsonModel
        });
        break;
    default:
      return state;
  }
```

Note that:

- We **do not mutate the state**. We create a copy with Object.assign(). Object.assign(state, { metadata: screenJsonModel }) is also wrong: it will mutate the first argument. You must supply an empty object as the first parameter. You can also enable the object spread operator proposal to write { ...state, ...newState } instead.

- We **return the previous state in the default case**. It is important to return the previous state for any unknown action.

Source

## Store

In the previous sections, we defined the **actions** that represent the facts about “what happened” and the **reducers** that update the state according to those actions.

The Store is the object that **brings them together**. The store has the following responsibilities:

- Holds application state;
- Allows access to state via getState();
- Allows state to be updated via dispatch(action);
- Registers listeners via subscribe(listener);
- Handles unregistering of listeners via the function returned by subscribe(listener).

It's important to note that you'll only have a single store in a Redux application. When you want to split your data handling logic, you'll use reducer composition instead of many stores.

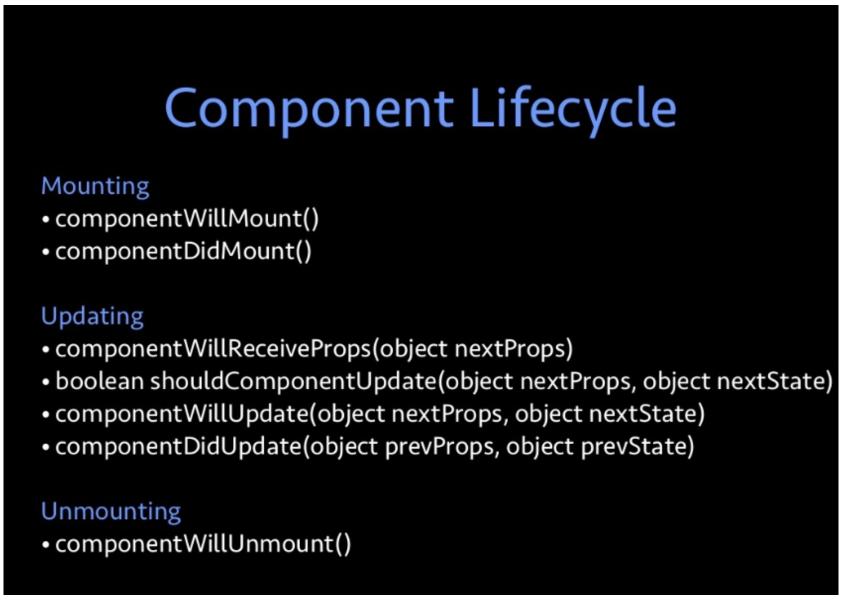
## View

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

```
class View extends Component {
    render() {
        var {data, dispatch} = this.props;

        var renderTitle = () => {
            if (data !== null && data.title !== null && typeo
                return <div>{data.title}</div>;
            } else {
                return <div>Loading</div>;
            }
        }

        var renderSections = () => {

            if (data && data.sections) {
                return data.sections.map((section) => {
                    return <Section key={section.title} title
                });
            } else {
                return <div>No existing sections</div>
            }

        };

        return (
            <div>
                <h1>{renderTitle()}</h1>
                {renderSections()}
            </div>
        )
    }
}
```

This is the API for components



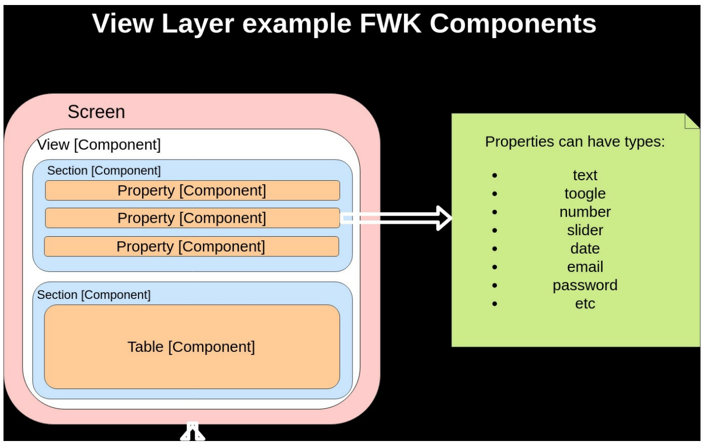
Component Lifecycle

Mounting
- componentWillMount()
- componentDidMount()

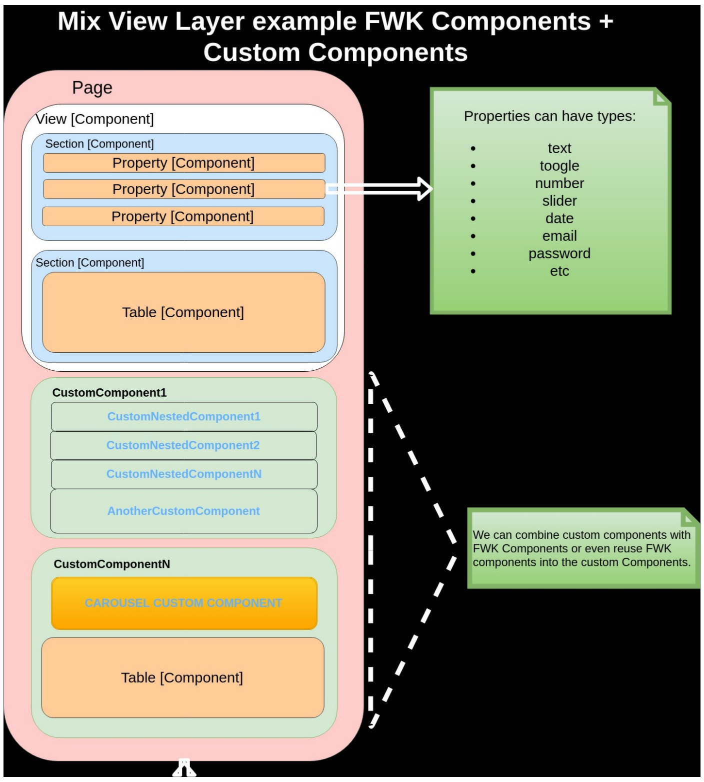
Updating
- componentWillReceiveProps(object nextProps)
- boolean shouldComponentUpdate(object nextProps, object nextState)
- componentWillUpdate(object nextProps, object nextState)
- componentDidUpdate(object prevProps, object prevState)

Unmounting
- componentWillUnmount()

## FWK View layer Diagram



## Custom + FWK View layer



Source

## Utils

In this part of the proof of concept we have stored other components related to empower the functionality of the project but we are going to describe the most important of them:

- **auth.js**. library for managing the login authentication
- **resourceFactory.js**. JS library for manage API Calls like put, get, patch, post, delete, etc
- **transform.js**. When we want to render a page it is neccesary tansform metamodel data and api data depending on the metamodel data. In this point, we merge metamodel data with response API data to generate a viewData for view components. This type of viewData can be, for example, an input type text or a Search table.

## Metamodel

This is the folder for storing all the json metamodel files and the name has to match with the screenId.

## HotLoading

Using react-hot-loader, your changes in the CSS and JS get reflected in the app instantly without refreshing the page. That means that the current application state persists even when you change something in the underlying code! For a very good explanation and demo watch Dan Abramov himself talking about it at react-europe.

## react-router

react-router is used for routing in this application. react-router makes routing really easy to do and takes care of a lot of the work.

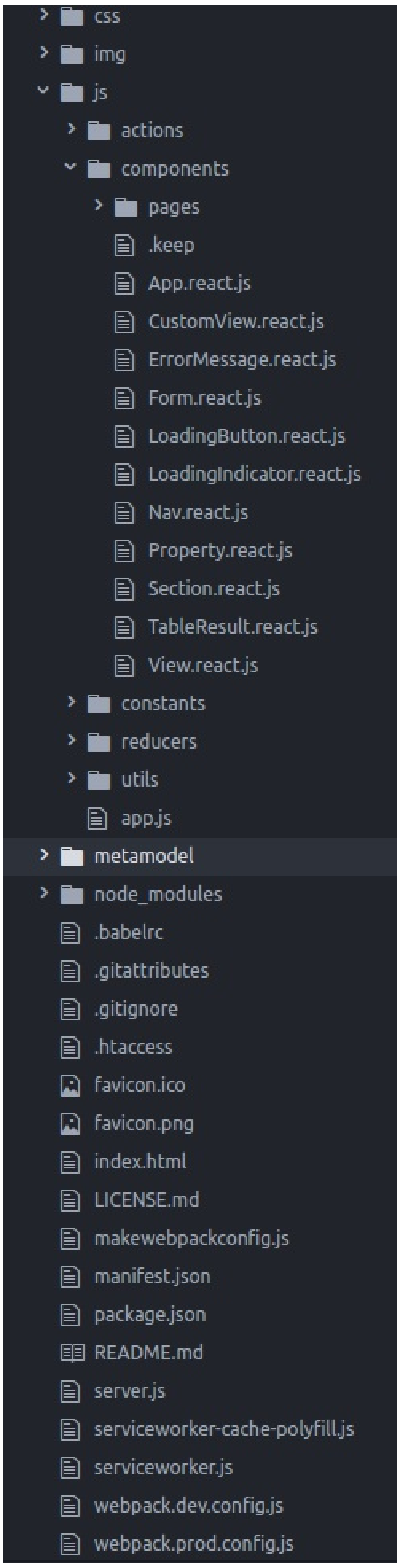
## ServiceWorker and AppCache

ServiceWorker and AppCache make it possible to use the application offline. As soon as the website has been opened once, it is cached and available without a network connection. manifest.json is specifically for Chrome on Android. Users can add the website to the homescreen and use it like a native app!

## Folder Structure

The folder structure of the JS files reflects how Redux works, so if you are not familiar with Redux check out the official documentation.

- actions: Actions get dispatched with this/these utility module(s)
- components: The main JS folder. All the React components are in this folder, with pages (routes) saved in the pages subfolder. E.g. a navigation component Nav.react.js
- constants: Action constants are defined in this/these utility module(s)
- reducers: Reducers manage the state of this app, basically a simplified implementation of Stores in Flux. For an introduction to reducers, watch this talk by @gaearon.
- utils: Utility files.



```
> css
> img
v js
    > actions
    v components
        > pages
          .keep
          App.react.js
          CustomView.react.js
          ErrorMessage.react.js
          Form.react.js
          LoadingButton.react.js
          LoadingIndicator.react.js
          Nav.react.js
          Property.react.js
          Section.react.js
          TableResult.react.js
          View.react.js
    > constants
    > reducers
    > utils
      app.js
> metamodel
> node_modules
  .babelrc
  .gitattributes
  .gitignore
  .htaccess
  favicon.ico
  favicon.png
  index.html
  LICENSE.md
  makewebpackconfig.js
  manifest.json
  package.json
  README.md
  server.js
  serviceworker-cache-polyfill.js
  serviceworker.js
  webpack.dev.config.js
  webpack.prod.config.js
```

## Authentication

Authentication happens in js/utils/auth.js, using fakeRequest.js and fakeServer.js. fakeRequest is a fake XMLHttpRequest wrapper with a similar syntax to request.js which simulates network latency. fakeServer responds to the fake HTTP requests and pretends to be a real server, storing the current users in localStorage with the passwords encrypted using bcrypt. To change it to real authentication, you would only have to import request.js instead of fakeRequest.js and have a server running somewhere.

## Stats, facts and enhancements

===================================

```
Summary
-------

  **Scope Requirements**                               | **Angul
  ---------------------------------------------------- | -------
  **Performance**                                      | Medium
  **Complexity**                                       | High
  **OcInfra Team Learning Curve**                      |
  **App Team Learning Curve**                          |
  **Rendering - Client side**                          | Yes
  **Rendering - Server side**                          | No
  **Mobility**                                         | Needs f
  **ES6 support**                                      | Yes
  **Browser compatibility**                            | Optimiz
  **Metamodel backward compatibility**                 | N/A
  **Bootstrap backward compatibility**                 | N/A
  **Maintainability**                                  | Medium
  **Ease of development**                              | High
  **Suitable for component-based UI framework**        | Weak
  **Data binding**                                     | 2-way
  **DOM**                                              | Regular
  **Packaging**                                        | Weak
  **Abstraction**                                      | Weak
  **Fails-When**                                       | Runtime
  **Templating**                                       | HTML fi
```

```
    **GitHub Stars**                                    | 54 k
    **GitHub Forks**                                    | 27 k
```

## Performance

The work done by Redux generally falls into a few areas: processing actions in middleware and reducers (including object duplication for immutable updates), notifying subscribers after actions are dispatched, and updating UI components based on the state changes. While it is certainly possible for each of these to become a performance concern in sufficiently complex situations, there is nothing inherently slow or inefficient about how Redux is implemented. In fact, React Redux in particular is heavily optimized to cut down on unnecessary re-renders, and React-Redux v5 shows noticeable improvements over earlier versions.

Redux may not be as efficient out of the box when compared to other libraries. For maximum rendering performance in a React application, state should be stored in a normalized shape, many individual components should be connected to the store instead of just a few, and connected list components should pass item IDs to their connected child list items (allowing the list items to look up their own data by ID). This minimizes the overall amount of rendering to be done. Use of memoized selector functions is also an important performance consideration.

As for architecture, anecdotal evidence is that Redux works well for varying project and team sizes. Redux is currently used by hundreds of companies and thousands of developers, with several hundred thousand monthly installations from NPM. One developer reported:

*for scale, we have ~500 action types, ~400 reducer cases, ~150 components, 5 middlewares, ~200 actions, ~2300 tests*

## Complexity

React is easy to learn but itself is not enough to create an SPA. Many people just use React as the "V" in the MVC architecture, which means you have to use other components and patterns with it in order to create a single page app. It also means you can use React with other popular frameworks such as Angular and Backbone and while using React tools for optimizing the performance of your application. In this case, we have choosen REdux that it takes time to learn how the router works and how the data flows through your React app, so Angular has an advantage here by providing everything out of the box. You have to spend some time for creating a FWK structure based on React and Redux but you will build a fast and maintainable solution which will be easy for new people to dive in and work on.

## Mobility

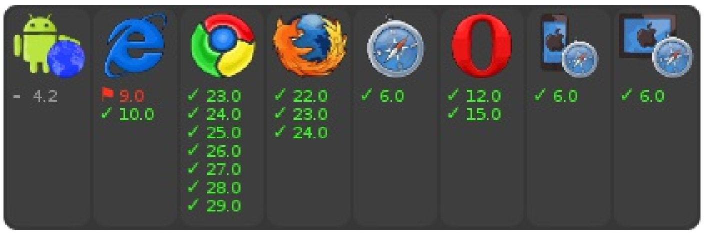
There's an existing technology for React mobility and it's called **React-Native**. React-Native is very similar to ReactJS in a way, but there are differences like:

- Setup and bundling: React-Native is a framework, where ReactJS is a javascript library you can use for your website. React-Native comes with everything you need and you most likely wouldn't need more. When you start a new project with REACT-Native you will notice how easy and fast is to set up.
- DOM and Styling: React-Native doesn't use HTML to render the app, but provides alternative components that work in a similar way.
- Animations and Gestures: you'll have to learn a completely new way to animate the different components of your app with Javascript. The recommended way to animate a component is to use the Animated API

provided by React-Native

- Navigation: Navigator component provided by React-Native is the best option as alternative to react-router used in ReactJS.
- Platform specific code: it is recommendable to make a difference between Android and iOS arquitectures but you can create an unique and universal design for both plartforms.
- Developer Tools: The beauty of working with React-Native is also the ability to use most developer tools you use with ReactJS. Chrome Dev Tools works beautifully to inspect the network requests (although you need to add a little trick to see the requests), display the console logs and stop the code on debuggerstatements. You can even use the great Redux DevTools to inspect the state of your Redux store.
- Publishing:you will need to learn how Xcode and Android Studio work in order to make sure everything is set up properly before the first deployment of your app on the App Store or the Google Play
- Wrapping up: You can build complex UI as quick as you would do with ReactJS and usually works pretty well for both iOS and Android. The learning curve from ReactJS to React-Native is I think quite easy, especially if you like to learn new Javascript frameworks

## Browser Compability



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| - 4.2 | ⚑ 9.0 | ✓ 23.0 | ✓ 22.0 | ✓ 6.0 | ✓ 12.0 | ✓ 6.0 | ✓ 6.0 |
| | ✓ 10.0 | ✓ 24.0 | ✓ 23.0 | | ✓ 15.0 | | |
| | | ✓ 25.0 | ✓ 24.0 | | | | |
| | | ✓ 26.0 | | | | | |
| | | ✓ 27.0 | | | | | |
| | | ✓ 28.0 | | | | | |
| | | ✓ 29.0 | | | | | |

## Virtual DOM

React kind of appears to reload data or the whole app on updates, when really it doesn't. It's the API that handles what changes on every update. It's also part of the way in which React solved traversing the DOM in linear rather than exponential complexity. So if we take a look at facebook's documentation we can see they built their virtual DOM algorithm off of two assumptions to expedite this process, Two elements of different types will produce different trees The developer can hint at which child elements may be stable across different renders with a key prop.

## Packaging and ES6 support

There is some complexity around preparing the component for releasing. I decided to document the process here so I have a solid resource next time. You may be surprised but writing the working jsx file doesn't mean that the component is ready for publishing and is usable for other developers.. The component: react-place ▭ component that renders into a widget. Toolset: new **ES6** and **ES7** spec are landing in our browsers. Babel is a tool that will parse ES6/ES7 code and will produce ES5 version compatible. will resolve the imports and will generate only one file containing the app.

- The base: Let's say that I want an already generated JavaScript file add it to my page with a `<script>` tag. I assume that React is already loaded on the page and I need only the component with its autocomplete widget included. To achieve this I effectively used the file under the lib folder

- The result: The final script is a combination of Babel, Browserify and Uglifyjs for getting an unique file like in the previous FWK in angularJS

## Metamodel backward compatibility

Totally compatible , easy and simple. We use a webpack loader like **json-loader**:

```
module: {
    loaders: [{
        test: /\.js$/, // Transform all .js files required somewh
        loader: 'babel', // ...with the specified loaders...
        exclude: path.join(__dirname, './node_modules/') // ..
      }, {
        test: /\.jpeg$|\.gif$|\.png$/i,
        loader: "url-loader?limit=10000"
      },{
        test: /\.json$/,
        loader: "json-loader"
    }
```

And the way for loding a json file, it would be in a reducer case previously dispatched by an action just like that using **require**:

```
  case SET_METAMODEL_OBJECT:
      var screenJsonModel = require('./../../metamodel/' + acti
      return assign({},state, {
        metadata: screenJsonModel
      });
      break;
```

It is not neccesary load a json file in a Asynchronous way like in the previous AngularJS FWK

## Bootstrap backward compatibility

It is compatible with this POC. Now, this POC is working with foundation. What is Foundation? It is just another CSS Framework but its the most advanced responsive front-end FWK.

To be compatible with Bootstrap we only have to make a configuration with a css-loader and include the bootstrap imports in the index.js.

In webpack.config.js we have to include new loaders:

```
{
      test: /\.scss$/,
      loader: 'style!css!postcss!sass'
    }, {
      test: /\.css$/,
      loader: 'style!css!postcss'
    }
```

and in index.js

```
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap/dist/js/bootstrap.min.js';
```

## Rendering (Client and server side)

When using Redux with server rendering, we must also send the state of our app along in our response, so the client can use it as the initial state. This is important because, if we preload any data before generating the HTML, we want the client to also have access to this data. Otherwise, the markup generated on the client will not match the server markup, and the client would have to load the data again.

To send the data down to the client, we need to:

create a fresh, new Redux store instance on every request; optionally dispatch some actions; pull the state out of store; and then pass the state along to the client. On the client side, a new Redux store will be created and initialized with the state provided from the server. Redux is only job on the server side is to provide the initial state of our app.

Source link

## Abstraction

React works with HOC (Higher Order Components) pattern. Higher Order Components is a great Pattern that has proven to be very valuable for several React libraries. A Higher Order Component is just a React Component that wraps another one.

At a high level HOC enables you to:

- Code reuse, logic and bootstrap abstraction
- Render Highjacking
- State abstraction and manipulation
- Props manipulation

## Fails-When

React fails in runtime but it contains a error code System for production error debugging.

[Source Link] (https://facebook.github.io/react/blog/2016/07/11/introducing-reacts-error-code-system.html)

## Github Stars and Forks

- Github starts: 62,4 k
- GitHub forks: 11,5 k

# POC Experience

- Positive points

    - sync rendering is largely solved via React Resolver.
    - Data-coupling is largely solved via Redux & Alt.
    - You will hit Angular performance issues much sooner than you expected. React will get you much, much further before you need to optimize.
    - Components are simpler & faster than Directives. Oddly enough, having more components clarifies an application’s architecture!
    - Rendering the same application on the server ensures a solid client-side experience by design.

- Negative points

    - JSX is Not Really HTML. in regards to maintenance, building features, andcon-boarding developers not familiar with JSX, it poses a bit of a hinderance.
        - Migrating existing code from other projects requires find/replace of className, for, and various other attributes.
        - Whitespace within {} expressions doesn’t behave how you’d expect.
        - You’re surprised how often you actually need to dangerouslySetInnerHTML.
    - Unidirectional Data Flow Is Complicated. Angular over React is much better in this case, because triggering renders to parent & sibling components, in a beggining, is confusing. Even when a solution is chosen like Redux, in the beggining there was a cognitive leap to go from route to component to action to store and understand who was watching what for changes.
    - Props vs. State. This is a complete change of paradigma for previous developers like i used to be. But once that it is understood, its easy know the diference between property and state.
    - Bring Your Own Architecture (BYOA). This is a problem

comparing with other technologies kind of Angular because React is just only a library that solves a view problem. You have to build your own architecture to achieve a MVC FWK. In this case, it was one of the weak points in this proof of concept. Mix a lot of technologies for building a FWK

Conclusions: What is better?

- AngularJS for prototyping
- React for building a universal app (Isomorphic Javascript -> might be used to describe JavaScript code that “can execute both on the client and the server” )
- Angular is not better or worse than REACT, it is just different but results are similar comparing performance and other things commented above.

## Concerns

### Data binding is a hack around re-rendering

The Holy Grail of simplicity is not in **discussion**. What everyone always wanted was to re-render our entire app when state changes. This way, we could stop having to deal with Root of All Evil problem: state changing over time — we could simply describe what our app looks like given any particular state.

Turns out they did. React implements a virtual DOM which kind of delivers us the Holy Grail.

```
var Hello = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

React.render(
  <Hello name=”World” />,
  document.getElementById(‘container’)
);
```

That’s all of the required API for a React component. You must have a render method. `<ironic mode="on">` Complex, huh? `<ironic/>` OK, but what about that `<div>` ?That’s not JavaScript! It sure isn’t.

## JSX :) Our new friend

This code is actually written in JSX, a super-set of Javascript which includes that brackets syntax for defining components. The code above, when compiled into JavaScript.

## Virtual DOM is FAST

As we already discussed, manipulating the DOM is ridiculously expensive, so it must be done as few times as possible. React’s virtual DOM, however, makes it really fast to compare two trees and find exactly what changed between them. That way, React is able to compute the minimum set of changes necessary to update the DOM. Practically speaking, React can diff two DOM trees and discover the minimum set of operations it needs to perform. This means two things:

- If an input with text is re-rendered and React expects it to have that content, it won’t touch the input. No more state loss!
- Diffing the virtual DOM is not expensive at all, so we can diff it as much as we like. When it’s ready to actually alter the DOM, it will only perform the least possible number of operations. No more slow layout

thrashing!

## React maps state to DOM

Virtual DOM rendering and diffing is the only magical part about React. Its excellent performance, however, is what fundamentally enables us to have a much simpler architecture overall. How simple? React components are idempotent functions. They describe your UI at any point in time, just like a server-rendered app. — Pete Hunt, React: Rethinking best practices