Bowen Liu

COMP 531

Dr. Pollack

4/20/2017

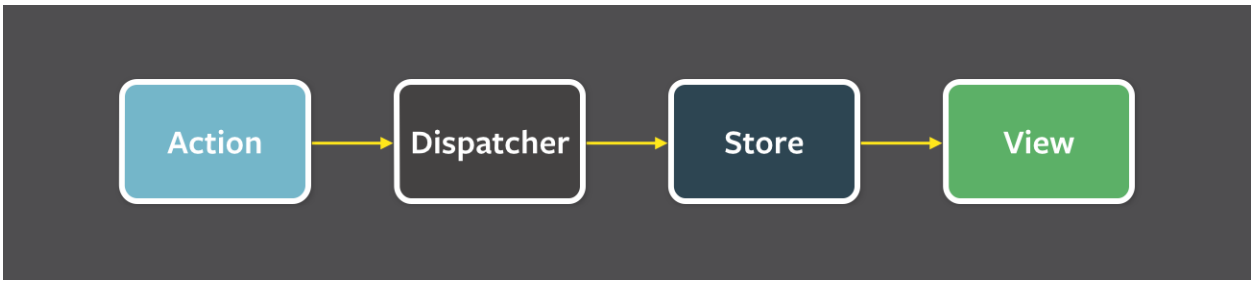
Flux and Redux: Their Relationships and Practical Usages

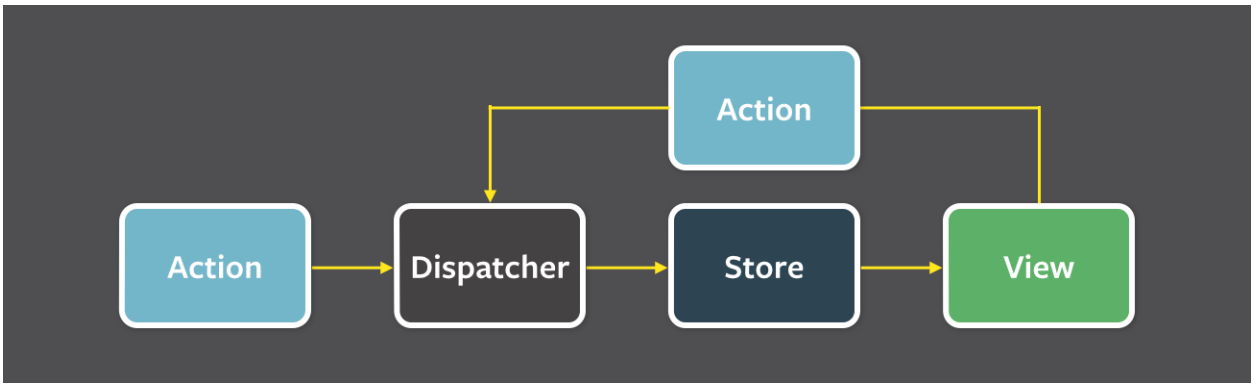
Abstract

In COMP 531, Prof. Pollack chose Redux instead of Facebook Flux or other implementations of Flux for web development in correspondence with React. This paper will dive into the structure and features of Flux, the "framework" of Redux and many others state containers alike, draw comparisons between Flux and Redux, and will conclude on why it is more practical to use Redux in many web development scenarios.

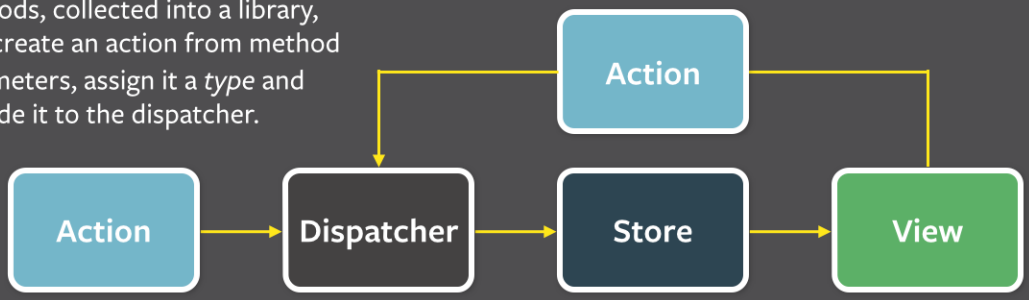
What is Flux?

According to Facebook, who developed Flux, Flux "is the *application architecture* that Facebook uses for building client-side web applications. It complements React's composable view components by utilizing a *unidirectional* data flow. It's more of a *pattern* rather than a formal framework, and you can start using Flux immediately without a lot of new code." In other words, Flux dictates the data flow, or state change, in the web application.





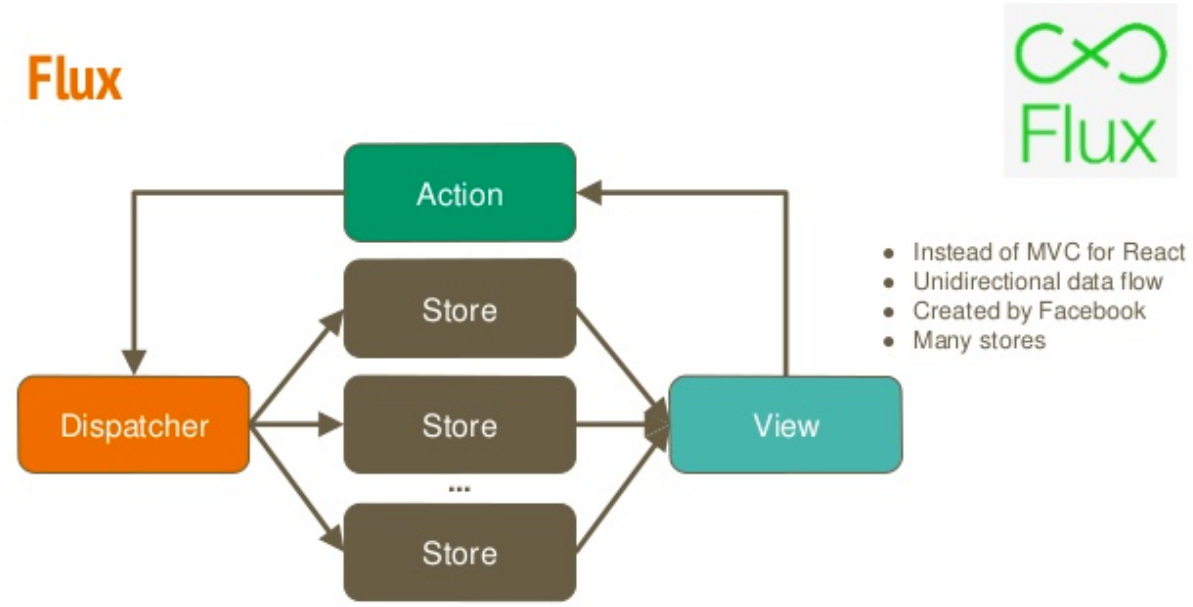
Action creators are helper methods, collected into a library, that create an action from method parameters, assign it a *type* and provide it to the dispatcher.



Every action is sent to all stores via the *callbacks* the stores register with the dispatcher.

After stores update themselves in response to an action, they emit a *change* event.

Special views called *controller-views*, listen for *change* events, retrieve the new data from the stores and provide the new data to the entire tree of their child views.



As can be observed from the illustrations above, this structure allows us to reason easily about our application in a way that is reminiscent of functional reactive programming, or more specifically data-flow programming or flow-based programming, where data flows through the application in a single direction — there are no two-way bindings. Application state is maintained only in the stores, allowing the different parts of the application to remain highly decoupled. Where dependencies do occur between stores, they are kept in a strict hierarchy, with synchronous updates managed by the dispatcher.

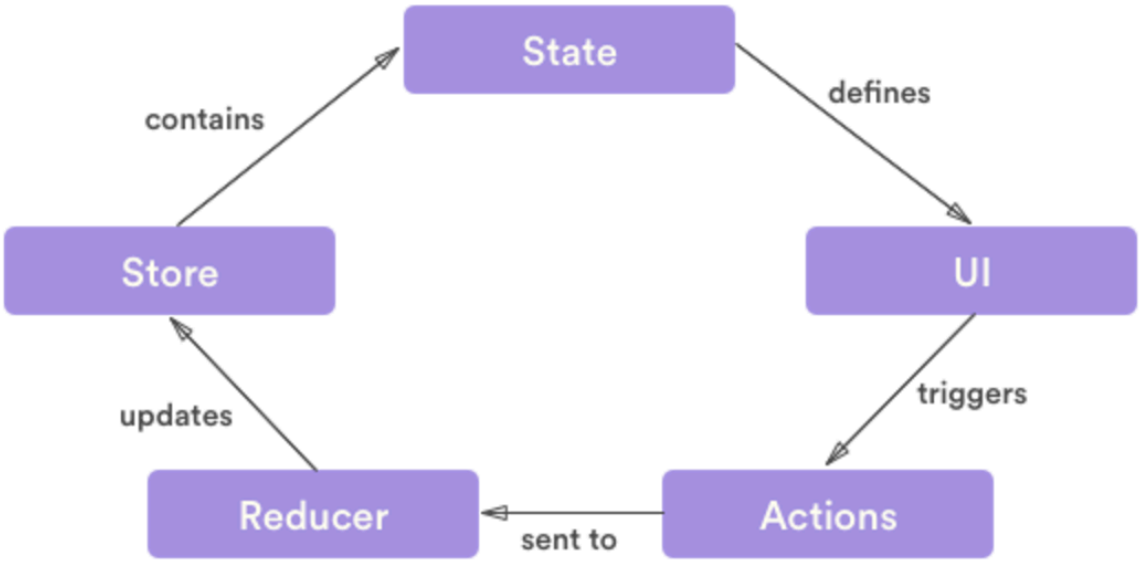
But why does Flux prohibit two-way data binding? The answer is, in short, it may lead to cascading updates. That means changing one object could cause another object to change, which could further trigger more updates. As applications grow, such cascading updates would make it very difficult to predict what exact objects would change as the result of one user interaction.

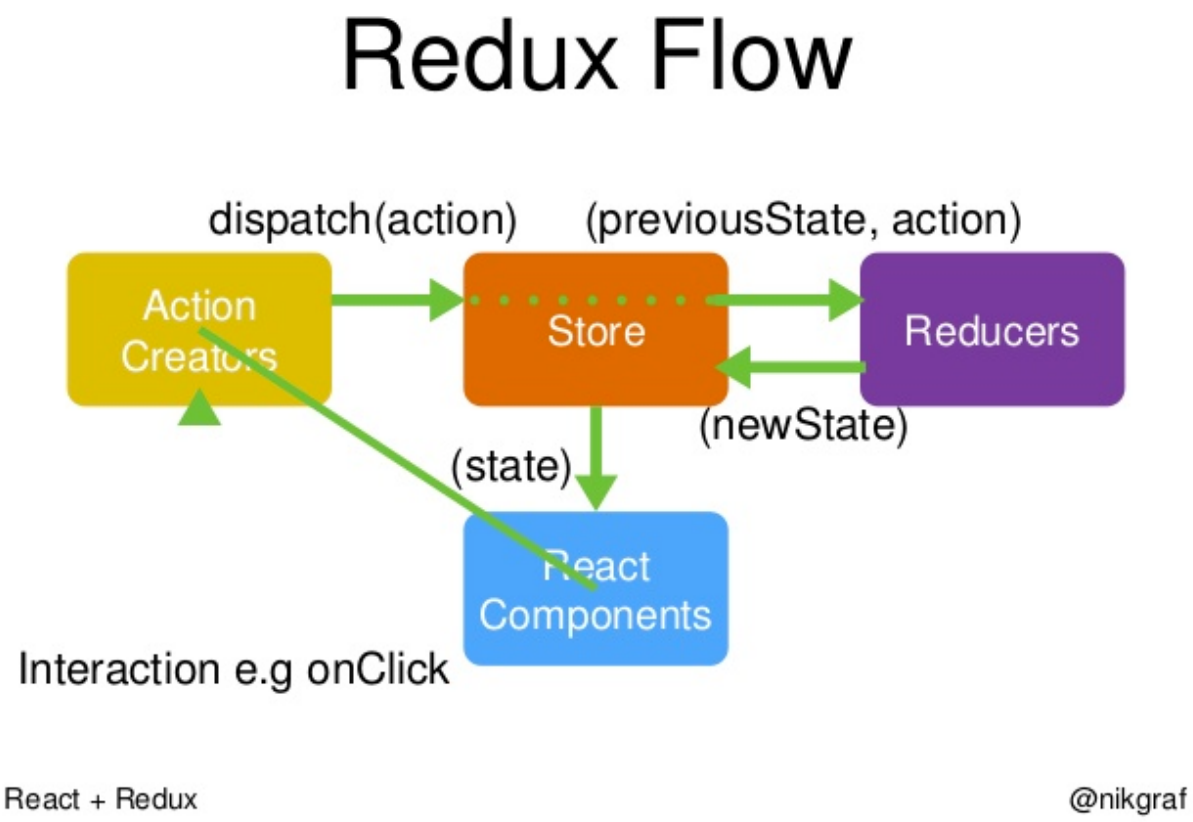
When updates can only change data in a one-way direction, the system as a whole becomes more predictable and thus easier to debug.

What is Redux?

As shown in COMP 531 class, Redux is a tiny yet powerful state container for Javascript Applications. In a typical Redux workflow, the views updates and user options always follow the same route:

1. The state tree defines the views and the action callbacks through *props;*
2. User actions, such as clicks, are sent to an action creator that normalizes them;
3. The resulting redux actions are passed to a reducer that implements the actual app logic;
4. The reducer updates the state tree and dispatches it to a store that, well, stores it;
5. The views update accordingly to the new state tree in the store.





Relations and Comparisons: The Advantages of Redux

First off, Redux is certainly originated in Flux; yet, it is not a pure Flux implementation. It is one particular implementation of a subset of flux concepts. The biggest difference is perhaps that it uses a single store that wraps a state object containing all the state for the app. Then instead of writing a bunch of stores, you as a developer write reducers for it, which are all pure functions. Other main differences between Redux and *full* flux implementations are:

- There are no discrete dispatchers in Redux; your store listens directly for actions, and uses a function called a reducer (more on this later) to return a new app state each time an action is dispatched.
- The state is immutable.

Consider the following as an example: think of your app state as a tree, with a root and child nodes. Were you to change a child node, you won't be able to change it directly as the state is immutable. Thus, you make a copy of the target node, then *apply* your changes. Further, you also have to make a copy of the target node's parent since it wouldn't make sense to have one parent produce two versions of the exact same node. Then you might even append a child node to the new node, too: when all of the above is done, you just need to attach the existing, unchanged nodes to your new ones, and return the whole thing as the new state. More importantly, the original state still exists. The unchanged nodes are not re-rendered in the new state, just the new ones. Clearly, this makes debugging much easier because you can isolate the action that changed the state, and even go back and forth at any time.

Now, comparing the simplicity of Facebook's Flux and Redux: Facebook has expanded the Flux repository to ease the implementation of its stores and containers. However, it is still hard to get started without a significant amount of boilerplate, as the simplicity comes at a cost. Redux is much simpler compared to flux: it does this via making additional assumptions beyond what Flux has made itself. The most profound assumption, for instance, is the assumption that you never mutate your data. Via this, developers no longer need the Flux *Dispatcher()*. In

addition, such assumption also makes it possible to describe changes to the data with pure old functions instead of a bulky switch statement.

Comparing the functionality between Facebook Flux and Redux, moreover, Flux doesn't provide any tools to work with action creators. Additionally, it is not easy for developers to handle asynchronous data. On the other hand, Redux doesn't lose out on any of the functionality, even though it remains much simpler than Flux. For instance, action objects are still created with action creators; there is still a central store object that allows inspection and debugging user input. And by using *redux-thunk*, it is still possible to dispatch multiple actions from one action creator. Thus, a Redux app is a starting state and a sequence of actions, which means things like logging out all the actions, stepping backwards in time, and replaying them in any way can be easily possible.

Conclusion

In this paper, we first reviewed the workflows of Flux and Redux respectively, which are dispatcher—store (multiple) —views—actions for Flux, and store (reducers and state) —views—actions for Redux. Realizing that Redux is not the "orthodox" implementation of Flux, we discussed why such differences are beneficial: the store and reducers are pure functions which reduces side effects; immutable states forcing creations of new states makes transactions easier; lastly, immutable states also make the whole app simpler for things like debugging.

Reference:

1. https://facebook.github.io/flux/
2. http://redux.js.org/
3. http://jamesknelson.com/which-flux-implementation-should-i-use-with-react/
4. https://www.youtube.com/watch?v=wmYvPYaAwso