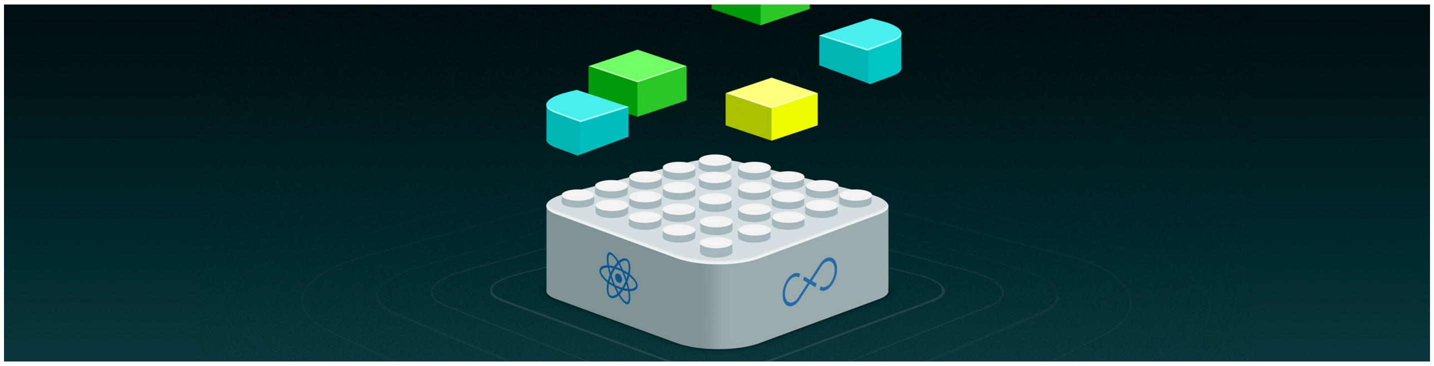
Nylas

Back to Nylas Blogs

The Nylas Engineering Blog



Building Plugins for React Apps

Techniques for modular, robust & flexible JavaScript plugins

By: Evan Morikawa

December 3, 2016

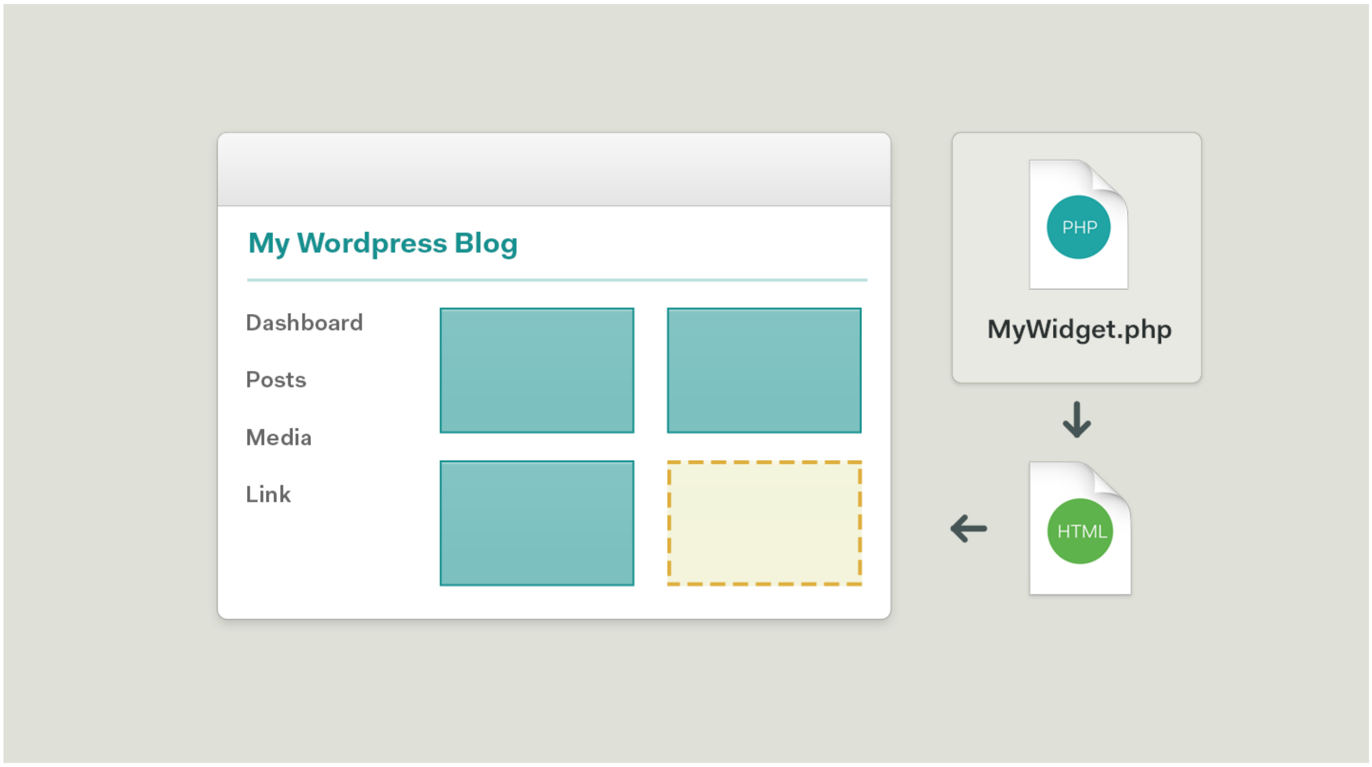
Earlier this year, we published a blog post about how our team is building a new email app using React, with a primary goal of extensibility. That means allowing developers to write plugins that change the app's behavior, just like in Chrome or emacs.

Over the past few months, we've designed a new way to structure large React applications in order to reliably and safely support plugins. This involves substantial changes to the typical Flux-based data layer, a method for exposing powerful integration points, and safe runtime isolation for plugin business logic.

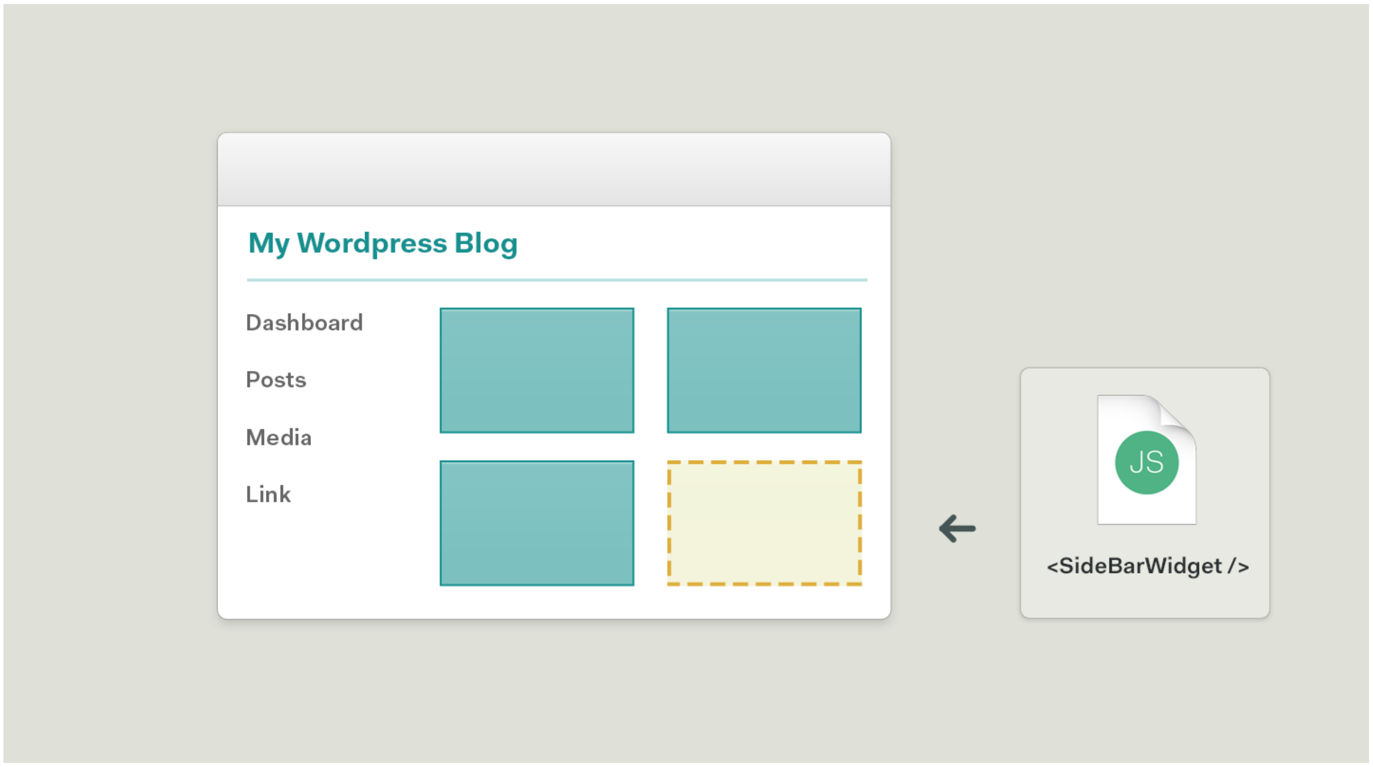
This blog post has the full story, including example code. Even if you're not building an app that requires plugin support, you can still apply these concepts to create more robust large React apps.

Why Plugins are Needed

Wordpress has defined the word 'plugin' for millions of developers looking to extend their blogs. There are plugins to customize the authoring interface, add brand new sidebars, register their own settings panels, and more. It's extremely powerful. Even the Wordpress dashboard supports plugins.



Because Wordpress is written in PHP, plugins are entirely rendered on the server. That was amazing back in 2004, but more than a decade later, things have changed. If rebuilt today, the Wordpress dashboard would probably be a client-side JavaScript application written in React, and it would support plugins as first-class objects.



Supporting Plugins With React

React allows you to define self-contained components with isolated behavior and rendering, so it's a great foundation for plugins. But once we started building, we found three key issues:

1. **Plugin components need to access state, and want to implement features that change state in unpredictable ways.** How can we make this safe and keep our views in sync? How do we organize our state?

2. **Plugin components need to appear somewhere in the application, but other components don't know how to render them.** Adding them directly to the DOM will produce invariant errors. How do we render these unknown components?

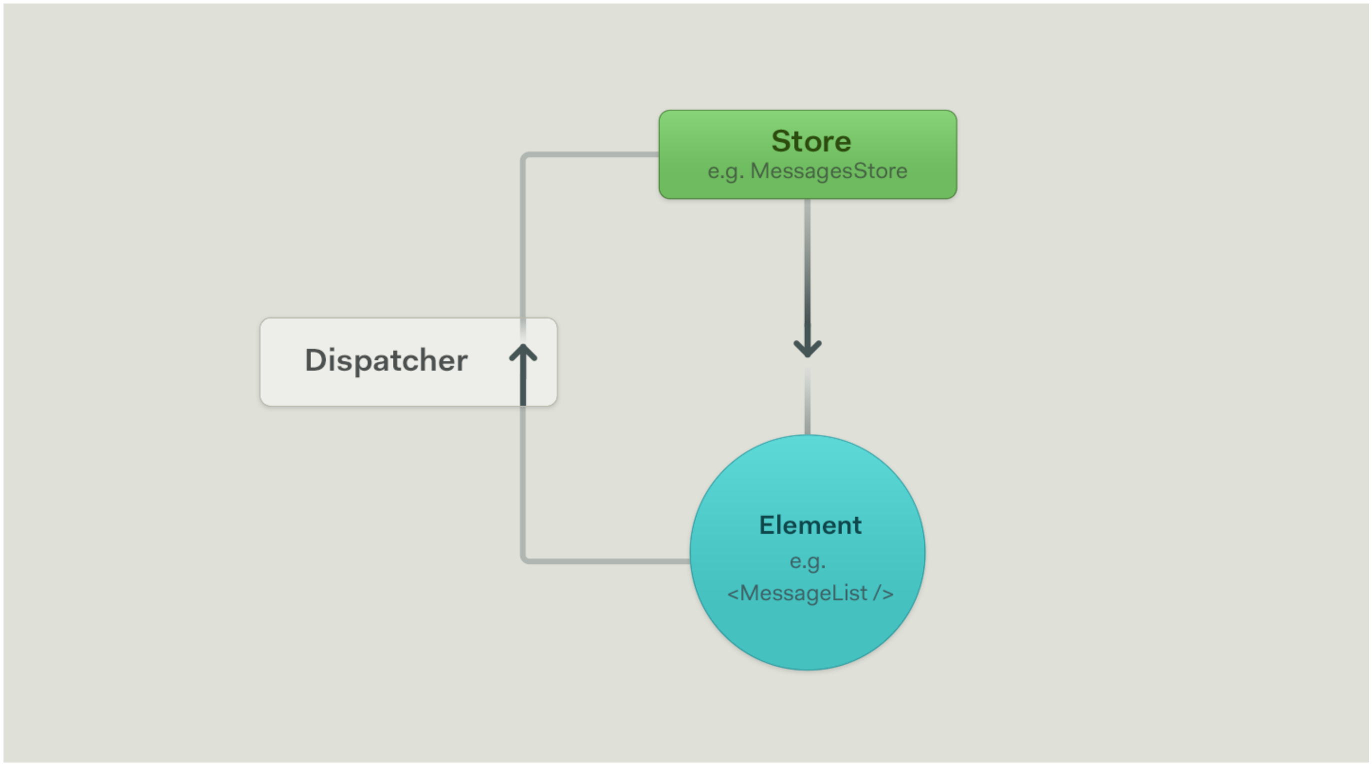
3. **Plugin components may throw exceptions, and they could crash the entire app.** How do we isolate fragile components to keep our app robust to plugin failures?

Expanding the Flux Pattern for Shared State in Large Applications

Allowing unknown 3rd party components to mutate shared state becomes hugely problematic in large apps. We've solved this by centralizing our Flux stores and exposing flexible interfaces for components.

What is Flux?

Let's use the example of building a simple chat app, which has a list of messages in its main view. This is how state flow works using the Flux pattern.



The green **Store** is a singleton object that holds data. In our example, it holds list of messages so we call it MessageStore.

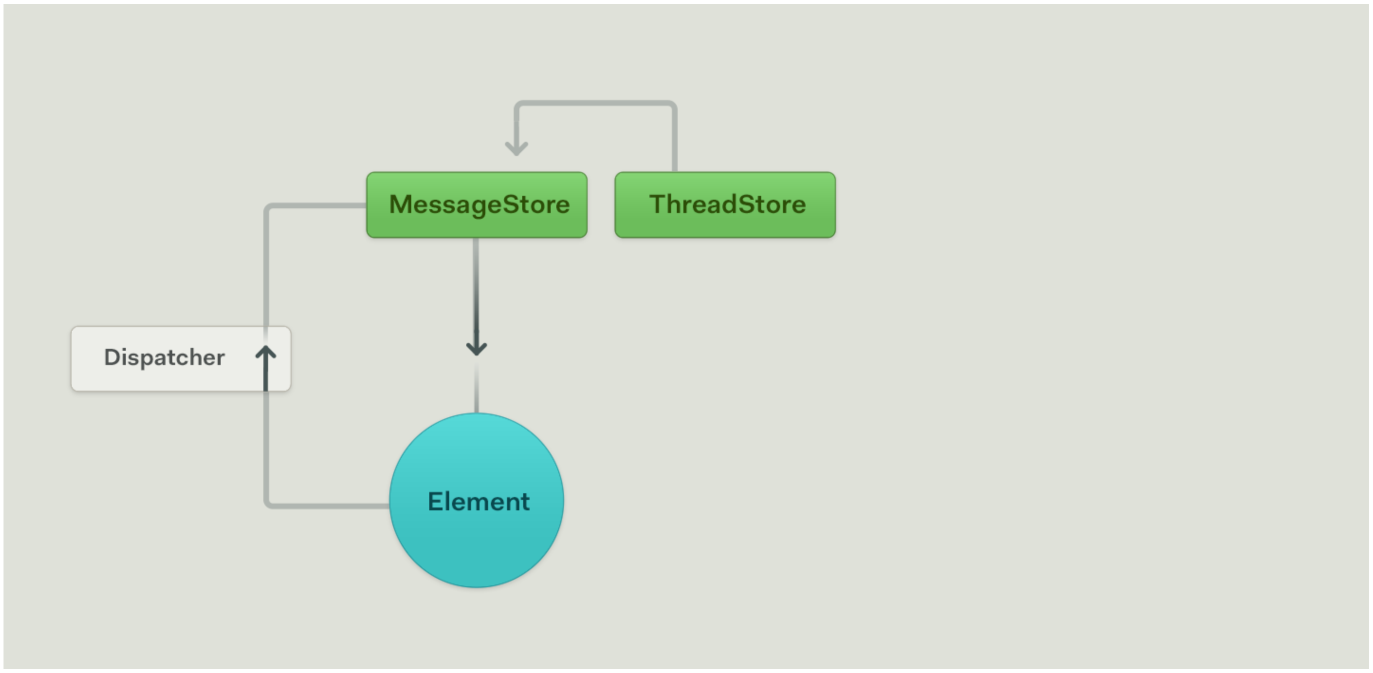
The blue **Component** is responsible for rendering data. In our example, this renders the list of messages. When data in the MessageStore changes, the MessageList component is updated with those changes and re-renders the view.

The global **Dispatcher** handles routing actions and events, usually from a component back to a store. If you clicked a "star" button on a message, that could broadcast an action throughout the system, which would update the store, and cause components to re-render.

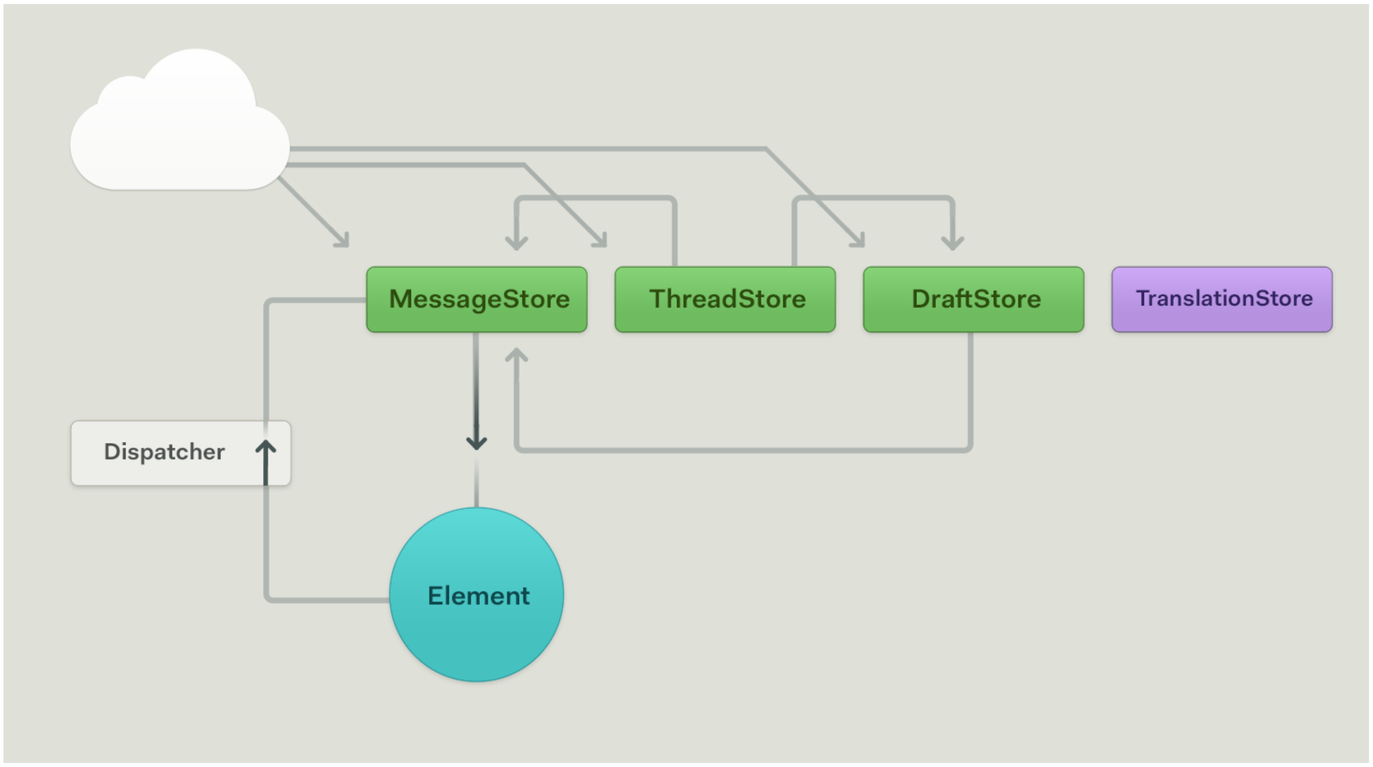
This is a simple version of the one-way data flow that React developers refer to as “Flux.” Because views aren’t bound directly to data models, Flux makes it very hard for views to get out of sync with the underlying state. This is a huge benefit over the traditional view controller-style interfaces, and is a critical enabler of extensibility.

From One Store to Many

Say you need to expand this chat app to manage multiple ongoing threads with different people. This is pretty simple: we just create a ThreadStore and ThreadList. Whenever a new thread is selected, the MessageStore needs to change the list of active messages in order for the MessageList to re-render. We do this by having the MessageStore listen to events from the ThreadStore.



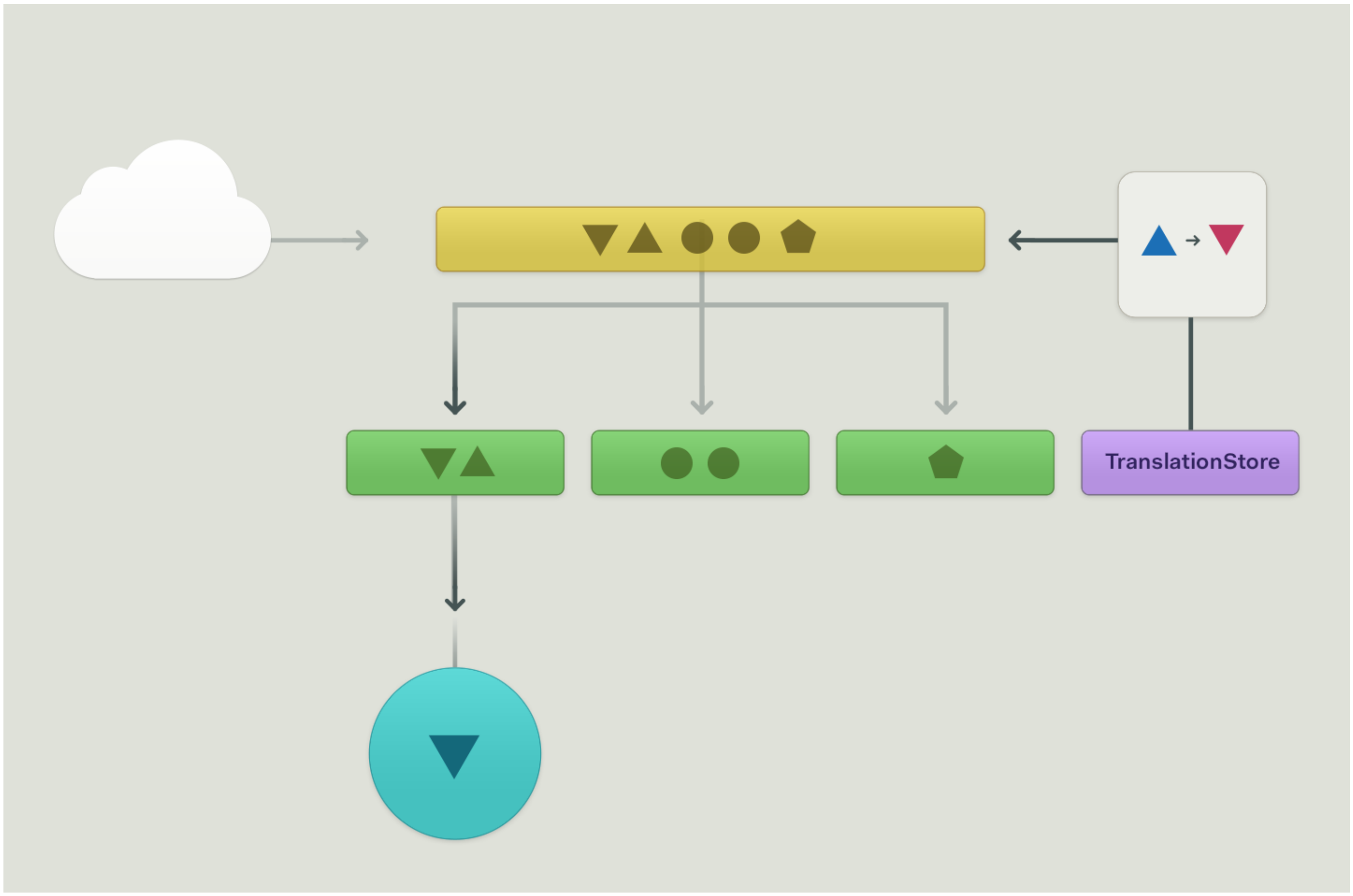
Linking stores like this is also a common design pattern in Flux, and gives way to interdependent data structures. Adding more connected stores can make this quickly grow out of control. It’s hard to reason about where data is being transformed, and easy to introduce obscure circular dependencies. This doesn’t scale, especially with plugins.



For example, say I want to build a translation plugin. This plugin primarily needs to transform the data, so I would write a TranslationStore, shown in purple. It would translate outgoing messages from the DraftStore, and publish translated messages to the MessageStore. Notice the growing spaghetti code?

The OmniStore

Our solution is to centralize the app's data in an OmniStore: a single top-level store that acts as a source of truth and eliminates the dependency chain. This store is just like a regular Flux store, except that it now holds all the primary application state. Using this design, other stores only cache and aggregate subsets of the central data, and vend it to to individual components.



This pattern is much easier to reason about when building and debugging. By decoupling previously dependent stores, it also allows us to safely introduce 3rd party plugin stores. Now the TranslationStore can directly update data in the OmniStore (yellow), which will broadcast notifications and cause the relevant components to re-render.

OmniStores can be implemented like any Flux store. If your app needs to work offline, we've found that the OmniStore is a natural place to implement a persistence layer (such as SQLite or HTML5 LocalStorage). Potentially this can also be extended to full ORM-like functionality, such as in Facebook's announced-but-not-yet-shipped Relay framework.

Modifying Flux to use an OmniStore enables us to share mutable state across our application, even with unknown 3rd party plugins. But how do those same plugins render their views within the app?

Dynamically Injecting React Components

In the previous example, we introduced a 3rd party plugin for translating messages. But this plugin probably also needs a “translate” button at the bottom of the message composer next to the send button. Here’s what the JSX might look like:

static-components.jsx

```
<div id=“actions”>
  <UndoButton draft={this.props.draft} />
  <AttachFileButton draft={this.props.draft} />
  <SendButton draft={this.props.draft} />
  // new component here!
</div>
```

The current draft is passed as a prop to each of the three buttons. But how can we enable our React app to dynamically inject a new TranslateButton here?

First, we move the list of buttons we are rendering from JSX to an array. This is a cool little React trick, and will render the exact same content. Since each button’s styling and behavior is self-contained, our app can simply pass the same props to all of them.

component-variables.jsx

```
render: function() {
   var componentClasses = [
      UndoButton,
      RedoButton,
      AttachFileButton
   ];

   return <div id=“actions”>{
       componentClasses.map(function(componentClass){
         return <componentClass draft={this.props.draft}/>;
       });
   }</div>;
}
```

The next step is to upgrade the components array into a Flux store, This new ComponentStore has an `componentClasses()` method to return an array of component constructors. The new toolbar component simply needs to observe the ComponentStore and render the passed component items.

injected-components.jsx

```
<div id="actions">
  <InjectedComponentSet
     location={'composer-actions'}
     passedProps={ {draft: this.props.draft} } />
</div>
```

The key here is that **views are just data**. Our goal is to write functions that operate on data and return data. JSX is just syntactic sugar for the common data shape we use all the time in React (i.e. type, props, children). And if we treat views as just data, we can take advantage of the same Flux patterns used throughout the app.

Maintaining an index of components in a Flux Store has a few key advantages. Components can leverage existing mixins and patterns, and listen to the store for changes in available components. This enables the app stays in sync as plugins are loaded and unloaded. We can also create Flux Actions for registering and unregistering components, which makes these processes globally visible within the application.

The props passed to an injected component are essentially the component's API. They can be clearly documented for developers, and reflect the true application state (unrelated to the DOM state). A component's lifecycle methods (i.e. register/unregister) define that component's entire existence, so there is no need to subscribe to other application events.

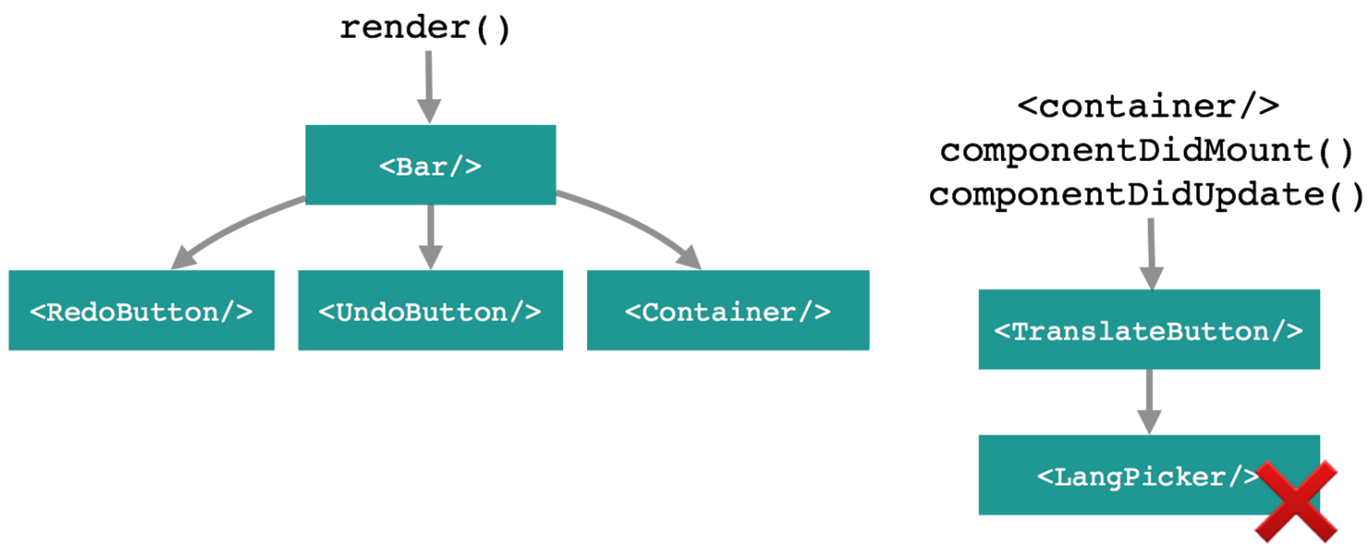
But there will always be bugs. Introducing these plugins means running arbitrary code, which can slow things down, or worse… crash your app.

Isolating Unknown Components

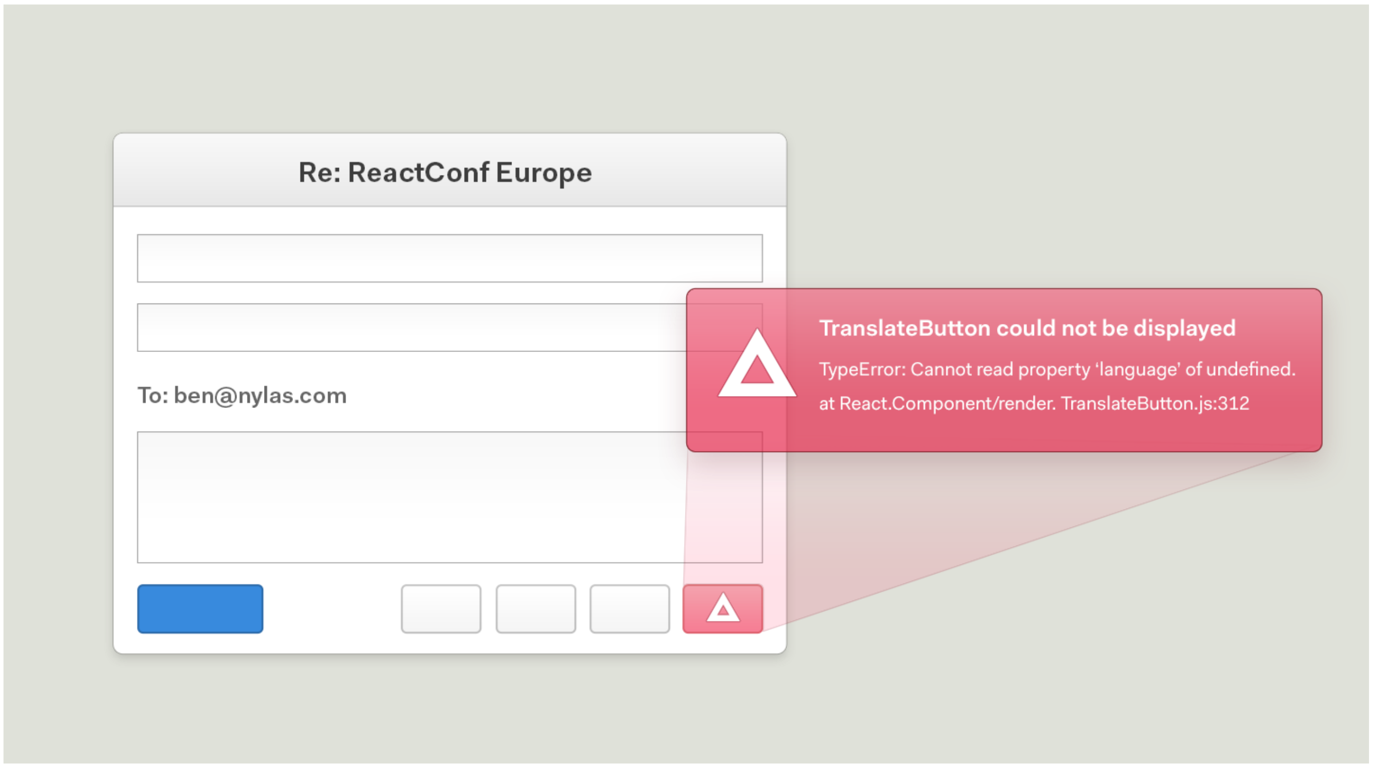
One of the downsides for a globally reactive application is that exceptions in lifecycle or render methods can sink the entire app. One bad egg can become the destroyer of worlds, leaving a trail of invariant errors with a broken UI.

We solve this by rendering components in a new React root. This means we initially render a placeholder div, and then manually call React.render within a try/catch block.

To manage the lifecycle, we trigger this on all component update events (componentDidUpdate / componentDidMount), and also call React.unmountComponentAtNode when components unmount. React.render is built to be flexible, and handles both rendering fresh or update existing views.



This lets us gracefully handle any errors in the render methods, and safely isolate, kill, and report bad components without impacting the rest of our app. We can even display helpful error messages to users and developers when things go wrong.



TranslateButton could not be displayed

TypeError: Cannot read property 'language' of undefined.
at React.Component/render. TranslateButton.js:312

Sample code

Want to try building plugins yourself? We’ve put together sample code for all of the techniques described above: OmniStores, component injection, and plugin isolation. You can check it out on GitHub.

Wrapping Up

In the last decade, the web community has undergone a dramatic shift toward client-side JavaScript applications. We’ve found that React and Flux are a great fit for enabling plugins and extendable applications. If you have comments, suggestions, or ideas for future posts, please let us know on Twitter.

Thanks to Pete Hunt, Lee Byron, and Katy Moe for reading drafts of this post.

This content was originally given as a talk at ReactEurope 2015.