Change And Its Detection In JavaScript Frameworks

Posted on Monday Mar 2, 2015 by Tero Parviainen (@teropa)

In 2015 there is no shortage of options when it comes to JavaScript frameworks. Between Angular, Ember, React, Backbone, and their numerous competitors, there's plenty to choose from.

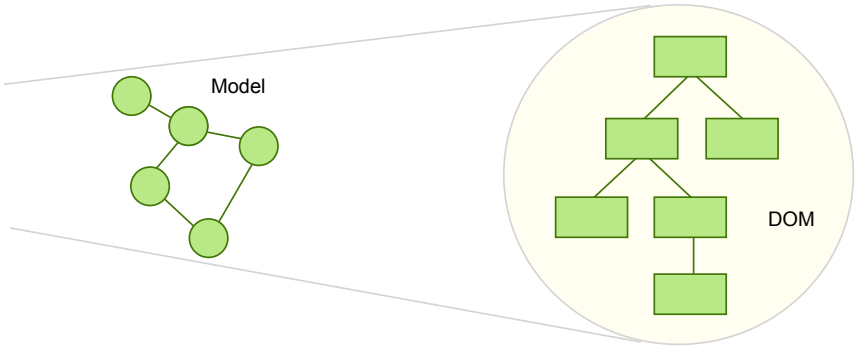
One can compare these frameworks in various ways, but I think one of the most interesting differences between them is the way they manage *state*. In particular, it is useful to think about what these frameworks do when state changes over time. What tools do they give you to reflect that change in your user interface?

Managing the synchronization of app state and the user interface has long been a major source of complexity in UI development, and by now we have several different approaches to dealing with it. This article explores a few of them: Ember's data binding, Angular's dirty checking, React's virtual DOM, and its relationship to immutable data structures.

Projecting Data

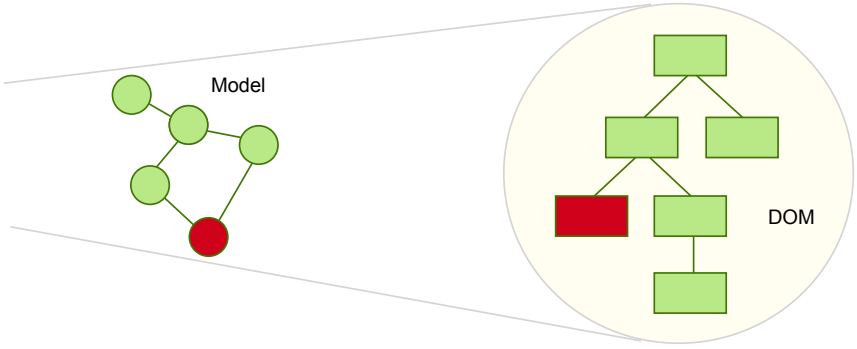
The basic task we're talking about is taking the internal state of the program and mapping it to something visible on the screen. You take an assortment of objects, arrays, strings, and numbers and end up with a tree of text paragraphs, forms, links, buttons, and images. On the web, the former is usually represented as JavaScript data structures, and the latter as the Document Object Model

We often call this process rendering, and you can think of it as a *projection* of your data model to a visible user interface. When you render a template with your data, you get a DOM (or HTML) representation of that data.



This by itself is simple enough: While the mapping from data model to UI may not always be trivial, it's basically still just a function with straightforward inputs and outputs.

Where things start to get more challenging is when we start talking about data *changing over time*. This can happen when the user interacts with the UI, or when something else happens in the world that updates the data. The UI needs to reflect this change. Furthermore, because rebuilding DOM trees is expensive, we'd like to do as little work as possible to get that updated data on the screen.

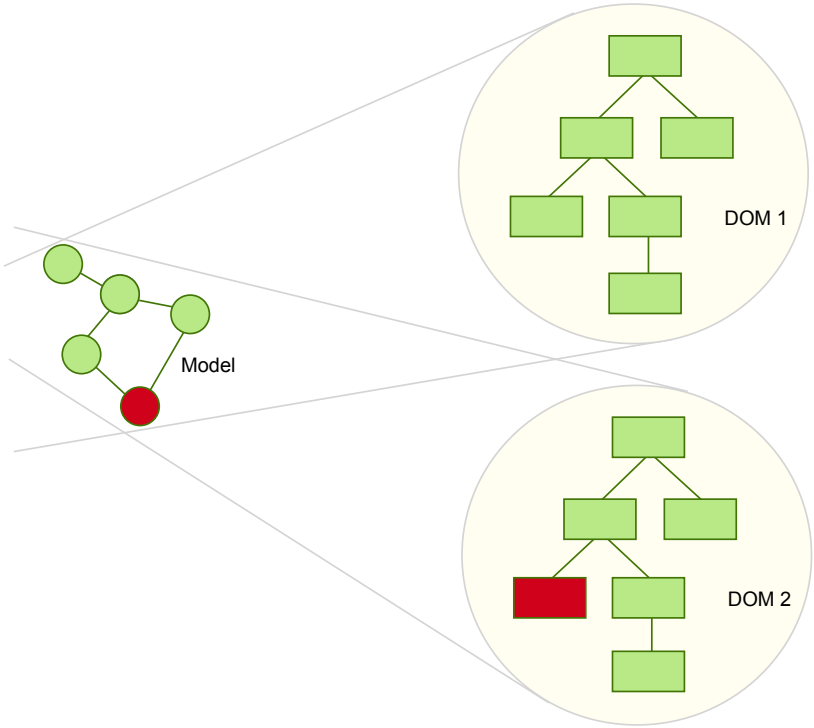


This is a much more difficult problem than merely rendering a UI once, since it involves *state changes*. This is also where the solutions out there begin to show their differences.

Server-Side Rendering: Reset The Universe

> "There is no change. The universe is immutable."

Before the era of Big JavaScript, every interaction you had with a web application used to trigger a server roundtrip. Each click and each form submission meant that the webpage unloaded, a request was sent to the server, the server responded with a *new* page that the browser then rendered.



With this approach, there wasn't really any state to manage in the front-end. Each time something happened, that was the end of the universe, as far as the browser was concerned. Whatever state there was, it was a backend concern. The frontend was just some generated HTML and CSS, with perhaps a bit of JavaScript sprinkled on top.

While this was a very simple approach from a front-end perspective, it was also very slow. Not only did each interaction mean a full re-rendering of the UI, it was also a *remote* re-rendering, with a full roundtrip to a faraway data center.

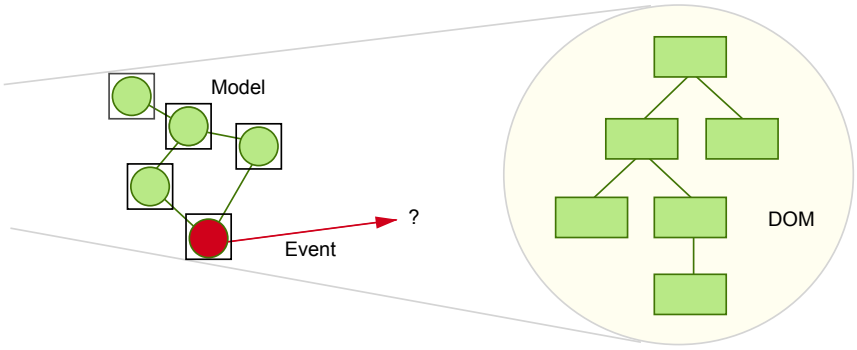
Most of us don't do this in our apps anymore. We may render the *initial* state of our app server-side but then switch to managing things in the front-end (which is what isomorphic JavaScript is largely about). There are people that are still succeeding with more sophisticated versions of this pattern though.

First-gen JS: Manual Re-rendering

> "I have no idea what I should re-render. You figure it out."

The first generations of JavaScript frameworks, like Backbone.js, Ext JS, and Dojo, introduced for the first time an actual *data model* in the browser, instead of just having some light-weight scripting on top of the DOM. That also meant that for the first time you had *changing state* in the browser. The contents of the data model could change, and then you had to get those changes reflected in the user interface.

While these frameworks gave you the architecture for separating your UI code from your models, they still left the synchronization between the two up to you. You can get some sort of events when changes occur, but it is your responsibility to figure out what to re-render and how to go about it.



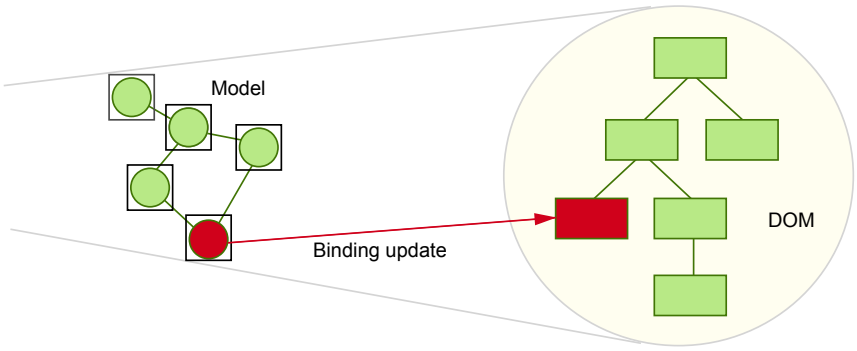
The performance considerations of this model are also left largely to you as an application developer. Since you control what gets updated and when, you can tweak it pretty much as you'd like. It often becomes a balancing act between the simplicity of re-rendering large sections of the page, and the performance of re-rendering just the bits that need updating.

Ember.js: Data Binding

> "I know exactly what changed and what should be re-rendered because I control your models and views."

Having to manually figure out re-rendering when app state changes is one of the major sources of incidental complexity in first-gen JavaScript apps. A number of frameworks aim to eliminate that particular problem. Ember.js is one of them.

Ember, just like Backbone, sends out events from the data model when changes occur. The difference is that with Ember, there's also something the framework provides for the receiving end of the event. You can bind the UI to the data model, which means that there is a listener for update events attached to the UI. That listener knows what updates to make when it receives an event.



This makes for a pretty efficient update mechanism: Though setting all the bindings up takes a bit of work initially, after that the effort needed to keep things in sync is minimal. When something changes, only the parts of the app that actually need to do something will activate.

The big tradeoff of this approach is that Ember must always be aware of changes that occur in the data model. That means you need to have your data in special objects that inherit from Ember's APIs, and that you need to change your data using special setter methods. You can't say `foo.x = 42`. You have to say `foo.set('x', 42)`, and so on.

In the future this may be helped somewhat by the arrival of Proxies in ECMAScript 6. It lets Ember decorate regular objects with its binding code, so that all the code that interacts with those objects doesn't necessarily need to adhere to the setter conventions.

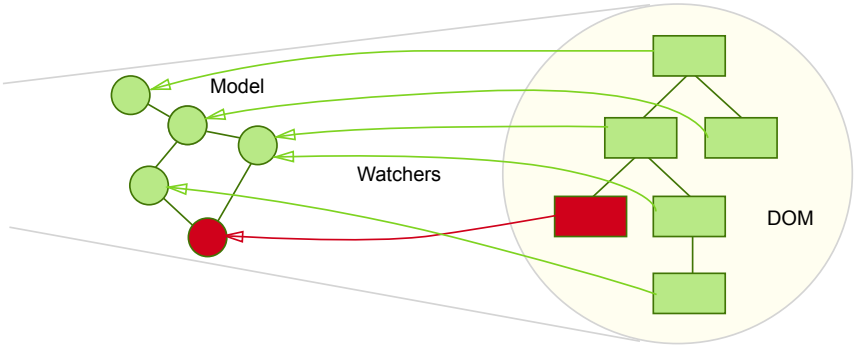
AngularJS: Dirty Checking

> "I have no idea what changed, so I'll just check everything that may need updating."

Like Ember, Angular also aims to solve the problem of having to manually re-render when things have changed. However, it approaches the problem from a different angle.

When you refer to your data in an Angular template, for example in an expression like `{{foo.x}}`, Angular not only renders that data but also creates a *watcher* for that particular value. After that, whenever anything

happens in your app, Angular checks if the value in that watcher has changed from the last time. If it has, it re-renders the value in the UI. The process of running these watchers is called *dirty checking*.



The great benefit of this style of change detection is that now you can use anything in your data models. Angular puts no restrictions on that - it doesn't care. There are no base objects to extend and no APIs to implement.

The downside is that as the data model now doesn't have any built-in probes that would tell the framework about changes, the framework doesn't have any insight into if and where it may have occurred. That means the model needs to be inspected for changes externally, and that is exactly what Angular does: All watchers are run every time anything happens. Click handlers, HTTP response processors, and timeouts all trigger a *digest*, which is the process responsible for running watchers.

Running all watchers all the time sounds like a performance nightmare, but it can be surprisingly fast. This is mostly because there's usually no DOM access happening until a change is actually detected, and pure JavaScript reference equality checks are cheap. But when you get into very large UIs, or need to render very often, additional performance optimization trickery may be required.

Like Ember, Angular will also benefit from upcoming standards: In particular, the Object.observe feature in ECMAScript 7 will be a good fit for Angular, as it gives you a native API for watching an object's properties for changes. It will not cover all the cases that Angular needs to support though, as Angular's watchers can do much more than just observe simple object attributes.

The upcoming Angular 2 will also bring some interesting updates on the change detection front, as has been described recently in an article by Victor Savkin. *Update 7.3.2015: Also see Victor's ng-conf talk about the subject.*

React: Virtual DOM

> "I have no idea what changed so I'll just re-render everything and see what's different now."

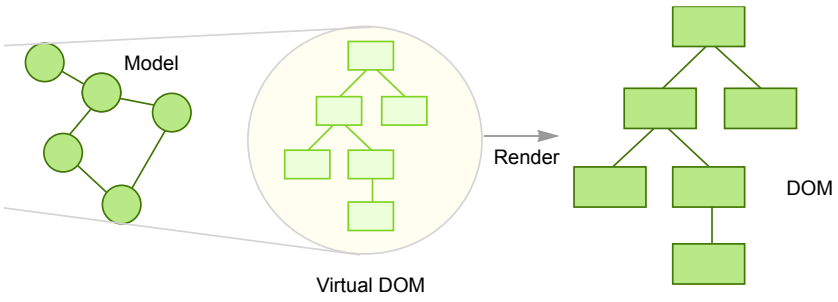
React has many interesting features, but the most interesting of them in terms of our discussion is the virtual DOM.

React, like Angular, does not enforce a data model API on you and lets you use whatever objects and data structures you see fit. How does it, then, solve the problem of keeping the UI up to date in the presence of change?

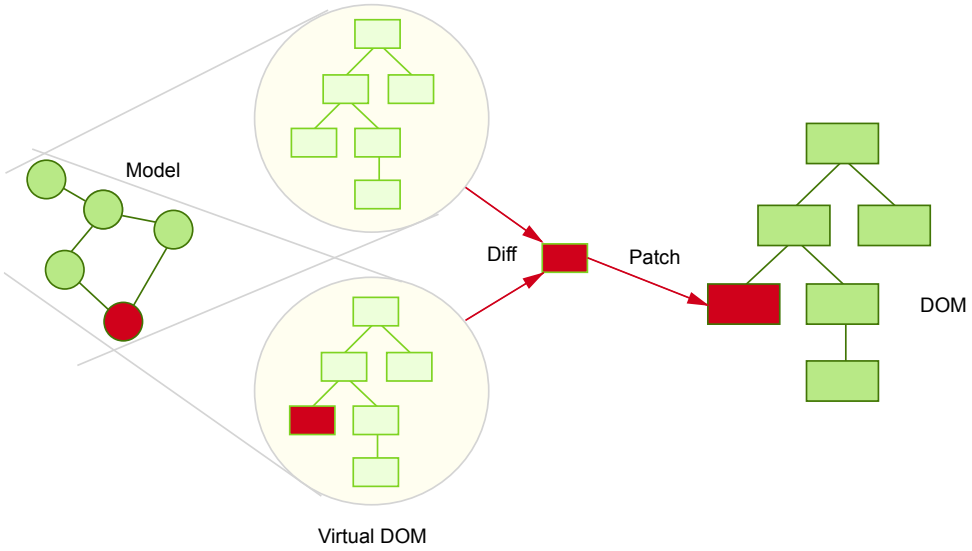
What React does is effectively take us back to the good old server-side rendering days, when we simply didn't care about state changes: It renders the whole UI from scratch every time there may have been a change somewhere in it. This can simplify UI code significantly. You (mostly) don't care about maintaining state in React components. Just like with server-side rendering, you render once and you're done. When a changed component is needed, it's just rendered again. There's no difference between the initial rendering of a component and updating it for changed data.

This sounds immensely inefficient, and it would be if that was the end of the story. However, the re-rendering React does is of a special kind.

When a React UI is rendered, it is first rendered into a *virtual DOM*, which is not an actual DOM object graph, but a light-weight, pure JavaScript data structure of plain objects and arrays that *represents* a real DOM object graph. A separate process then takes that virtual DOM structure and creates the corresponding real DOM elements that are shown on the screen.



Then, when a change occurs, a new virtual DOM is created from scratch. That new virtual DOM will reflect the new state of the data model. React now has *two* virtual DOM data structures at hand: The new one and the old one. It then runs a diffing algorithm on the two virtual DOMs, to get the set of *changes* between them. Those changes, and only those changes, are applied to the real DOM: This element was added, this attribute's value changed, etc.



So the big benefit of React, or at least one of them, is that you don't need to track change. You just re-render the whole UI every time and whatever changed will be in the new result. The virtual DOM diffing approach lets you do that while still minimizing the amount of expensive DOM operations.

Om: Immutable Data Structures

> "I know exactly what didn't change."

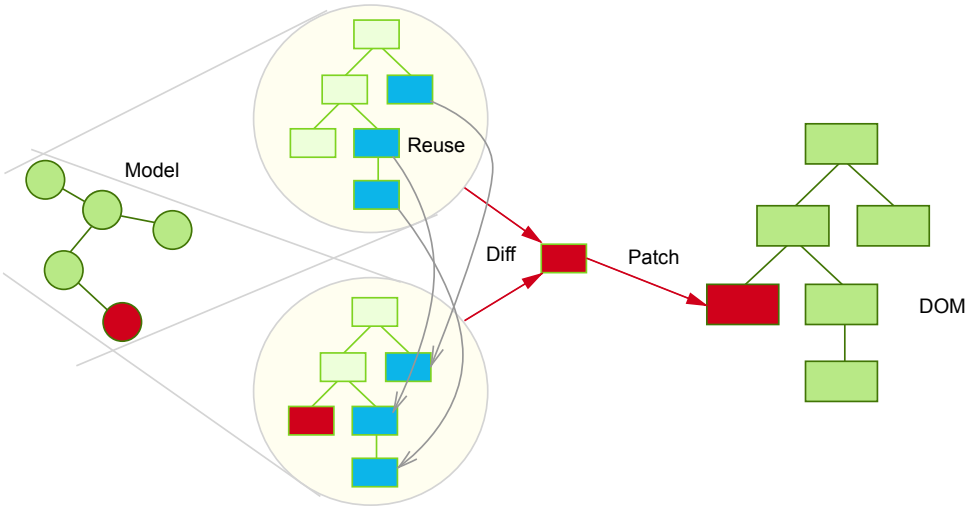
While React's virtual DOM approach is pretty fast, it can still become a bottleneck when your UI is big or when you want to render very often (say, up to 60 times per second).

The problem is that there's really no way around rendering the whole (virtual) DOM each time, unless you reintroduce some control into the way you let changes happen in the data model, like Ember does.

One approach to controlling changes is to favor immutable, persistent data structures. They seem to be a particularly good fit with React's virtual DOM approach, as has been shown by David Nolen's work on the Om library, built on React and ClojureScript.

The thing about immutable data structures is that, as the name implies, you can never mutate one, but only produce *new versions* of it. If you want to change an object's attribute, you'll need to make a new object with the new attribute, since you can't change the existing one. Because of the way persistent data structures work, this is actually much more efficient than it sounds.

What this means in terms of change detection is that when a React component's state consists of immutable data only, there's an escape hatch: When you're re-rendering a component, and the component's state still points to the same data structure as the last time you rendered it, you can skip re-rendering. You can just use the previous virtual DOM for that component and the whole component tree stemming from it. There's no need to dig in further, since nothing could possibly have changed in the state.



Just like Ember, libraries like Om do not let you use any old JavaScript object graphs in your data. You have to build your model from immutable data structures from the ground up to reap the benefits. I would argue that the difference is this time you don't do it just to make the framework happy. You do it because it's simply a nicer approach to managing application state. The main benefit of using immutable data structures is not to gain rendering performance, but to simplify your application architecture.

While Om and ClojureScript have been instrumental in combining React and immutable data structures, they're not the only game in town. It is perfectly possible to just use plain React and a library like Facebook's Immutable-js. Lee Byron, the author of that library, gave a wonderful introduction to the topic in the recent React.js Conf:

I would also recommend watching Rich Hickey's Persistent Data Structures And Managed References for an introduction to this approach for state management.

I myself have been waxing poetic about immutable data for a while now, though I definitely didn't foresee its arrival in frontend UI architectures. It seems to be happening in full force though, and people in the Angular team are also working on adding support for them.

Summary

Change detection is a central problem in UI development, and JavaScript frameworks attempt to solve it in various ways.

EmberJS detects changes when they occur, because it controls your data model API and can emit events when you call it.

Angular.js detects changes after the fact, by re-running all the data bindings you've registered in your UI to see if their values have changed.

Plain React detects changes by re-rendering your whole UI into a virtual DOM and then comparing it to the old version. Whatever changed, gets patched to the real DOM.

React with immutable data structures enhances plain React by allowing a component tree to be quickly marked as unchanged, because mutations within component state are not allowed. This isn't done primarily for performance reasons, however, but because of the positive impact it has on your app architecture in general.

- A Dash of Queueing Theory
- A Guide To Transclusion in AngularJS

comments powered by Disqus

- javascript
- web
- angularjs
- clojurescript

Contact: @teropa / tero@teropa.info