React: Facebook's Functional Turn on Writing JavaScript

case study

A DISCUSSION WITH PETE HUNT, PAUL O'SHANNESSY, DAVE SMITH AND TERRY COATTA

One of the long-standing ironies of user-friendly JavaScript front ends is that building them typically involved trudging through the DOM (Document Object Model), hardly known for its friendliness to developers. But now developers have a way to avoid directly interacting with the DOM, thanks to Facebook's decision to open-source its React library for the construction of user interface components.

React essentially manages to abstract away the DOM, thus simplifying the programming model while also—in a somewhat surprising turn—improving performance. The key to both advances is that components built from standard JavaScript objects serve as the fundamental building blocks for React's internal framework, thus allowing for greatly simplified composability. Once developers manage to get comfortable with building front ends in this way, they typically find they can more readily see what's going on while also enjoying greater flexibility in terms of how they structure and display data.

All of which caused us to wonder about what led to the creation of React in the first place and what some of its most important guiding principles were. Fortunately for us, Pete Hunt, who at the time was an engineering manager at Instagram as well as one of the more prominent members of

Facebook's React core team, is willing to shed some light on React's beginnings. Hunt has since gone on to cofound Smyte, a San Francisco startup focused on security for marketplaces and social networks.

Also helping to tell the story is Paul O'Shannessy, one of the first engineers at Facebook to be dedicated to React full time. He came to that role from Mozilla, where he had previously worked on the Firefox front end.

The job of asking the probing questions that drive the discussion forward falls to Dave Smith and Terry Coatta. Smith is an engineering director at HireVue, a Salt Lake City company focused on team-building software, where he has had an opportunity to make extensive use of both Angular and React. Coatta is the CTO of Marine Learning Systems, where he is building a learning management system targeted at the maritime industry. He is also a member of the acmqueue *editorial board.*

DAVE SMITH What is it exactly that led to the creation of React?

PETE HUNT Of all the web apps at Facebook, one of the most complex is what we use to create ads and manage ad accounts. One of the biggest problems is keeping the UI in sync with both the business logic and the state of the application. Traditionally, we've done that by manually manipulating the DOM using a centralized event bus, whether by putting events into the queue or by having listeners for the event and then letting them do their thing.

That proved to be really cumbersome, so a few years ago we implemented what we then considered to be a

state-of-the-art DOM-monitoring system called Bolt. It was kind of like Backbone with observables, where you would register for computed properties that would eventually get flushed to the DOM. But then we found that also was pretty hard to manage since you could never be sure when your properties were going to be updated—meaning that if you changed a value, you couldn't be sure whether it was going to cause a single update, cascading updates, or no updates at all. Figuring out when those updates might actually occur also proved to be a really hard problem.

The whole idea behind React initially was just to find some way to wire up those change handlers such that engineers could actually wrap their heads around them. That hadn't been the case with Bolt, and as a consequence we ended up with lots of bugs nobody could solve. So the engineers who started working on a way to remedy that ended up going wild for a couple of months and came out with this weird-looking thing nobody thought had any chance whatsoever of working. If you're even vaguely familiar with React, you already know that whenever there's a change in your underlying data model, it essentially re-renders the whole application and then does a diff to see what actually changed in the rendered result. Then it's only those parts of the page that get updated.

Some people here had some performance concerns about that, so an early version of React ended up being run through a serious gauntlet of engineering tests where it got benchmarked against pretty much everything that could be thrown at it. As part of that, of course, we looked at how this new programming model fared against both

the Bolt model and our old event model. React ended up really surprising a lot of people—enough so, in fact, that it was shipped almost immediately as part of our "liking and commenting" interface on News Feed. That was the first big test for React, and that came a few years ago.

Then we tried it out on Instagram.com, which is where I entered the picture since I was the person at Instagram responsible for building a few things using React. We were really happy with it since it proved to be capable of running our whole page instead of just one small widget here or there. That gave us a pretty good indication it was actually going to work. Since then, it has essentially become the de facto way people write JavaScript at Facebook.

TERRY COATTA I've heard React takes a different approach to data binding. What sets React apart there?

PH The way I think about data binding in a web context is that you've got some sort of observable data structure down to the DOM nodes. The challenge is that when you're implementing some sort of observable system, you're obliged to observe this data structure wherever your application touches the data model.

For example, if you use something like Ember, everything you do is going to use getters and setters, meaning you're going to need to remain aware of this observable abstraction throughout the entire application. So if you want to compute a value, you're not going to use a function only; you're going to use a computed property number, which is a domain-specific thing for Ember.

Angular, I think, does a much better job of this since it uses dirty checking, which means you can actually take advantage of regular JavaScript objects. The problem with

Angular, though, is that it makes it difficult to compose your application. That's because, instead of using regular functions or objects to build up abstractions (as you would do with JavaScript), you have to pass everything through a scope in order to observe those changes. Then you end up with this data binding that couples different parts of your program in ways that aren't necessarily all that clear or obvious.

For example, let's say we're looking to sort a list of your top friends—which is the kind of thing we do all of the time here. In order for us to do that with an observable system, we would have to set up an observer for every one of the thousand friends you've listed, even if all we're really looking to do is to render the top ten. So, as you can imagine, it's going to take a good chunk of memory to maintain that whole representation.

Obviously, there are ways to get around that, but people typically just break out of the data binding abstraction altogether at that point so they can proceed manually. Now, I generally hate to say something isn't going to scale, but it's fairly obvious this is going to present some scaling issues. It's clear that the bigger your application gets, the more you're going to run into this sort of edge case.

TC I agree completely about the Angular situation since I also find composition there to be tricky for just the reason you mentioned—that is, you end up having different parts of your application essentially coupled silently via two-way data binding. But I see that React also has data binding, so I'm curious about how you've managed to provide for better composability despite that coupling.

PH Let me zoom out a little here to observe that, at a very

high level, React essentially treats your user code as a black box while also taking in whatever data you tell it to accept. That basically allows for any structure. It could be something like Backbone. It could be plain JSON. It could be whatever you want. Then your code will just go ahead and do whatever it's supposed to do, backed by the full power of JavaScript.

At the end of that, however, it will return a value, which we call a virtual DOM data structure. That's basically just a fancy handle for JavaScript objects that tell you which kinds of elements they are and what their attributes are. So if you think of data binding as a way to keep your UI up to date with your underlying model, you can accomplish that with React just by signaling, "Hey, something in my data model may have just changed." That will prompt React to call the black-box user code, which in turn will emit a new virtual DOM representation. Then, having kept the previous representation, React will look at the new version and the old version and do a diff of the two. Based on that, it might conclude, "Oh, we need to build a className attribute at this node."

The advantage of this approach is that it involves no actual tracking of your underlying data model.

The advantage of this approach is that it involves no actual tracking of your underlying data model. You don't have to pay a data-binding cost up front. Most systems that require you to track changes within the data model and then keep your UI up to date with that are faced with a data-binding cost driven by the size of the underlying data model. React, on the other hand, pays that cost relative only to what actually gets rendered.

TC If I understand you correctly, you're saying React is in some sense a highly functional environment that

takes some arbitrary input, renders an output, and then computes the difference between the two to determine what it ought to be displaying on the screen.

PH Exactly. I like to describe this as "referentially transparent UI." Which is to say your user interface is generally a pure function of some set of inputs, and it emits the same kind of virtual DOM structure every single time for some given data input.

TC So the data bindings that have caused us grief in Angular run in the other direction here in the sense that they reflect the value of DOM elements that are bound to underlying model objects or scope variables. Any changes there effectively become visible at multiple locations throughout your code at much the same time, meaning the composability issues surface since different locations in your code are made aware almost simultaneously of changes that propagate backwards from the UI.

PH Another problem is that you might have multiple bindings to the same data source. So then which piece of code is going to be treated as the authoritative source for determining what the value ought to be?

This is why, with React, we emphasize one-way data flow. As I said earlier, data in our model first goes into this application black box, which in turn emits a virtual DOM representation. Then we close the loop with simple browser events. We'll capture a KeyUp event and command, "Update the data model in this place based on that KeyUp event." We've also architected the system in such a way as to encourage you to keep the least possible mutable state in your application. In fact, because React is such a functional system, rather than computing a value

and then storing it somewhere, we just recompute the value on demand with each new render.

The problem is that people sometimes want to have a big form that includes something like 20,000 fields that then bind to some simple keys and data objects. The good news is that it's actually very easy for us to build an abstraction on top of a simple event loop that basically captures all the events that might possibly update the value of this field, and then set up an automatic handler to pass the value down from the data model into the form field. The form and the data model essentially get updated at the same time. This means you end up with a system that looks a lot like data binding, but if you were to peel it back, you would see that it's actually only simple syntactic sugar on top of an event loop.

TC One of the things I've observed about React is that it seems to be what people would call fairly opinionated. That is, there's a certain way of doing things with React. This is in contrast to Angular, which I'd say is not opinionated since it generally lets you do things in several different ways. Do you think that is an accurate portrayal?

PH It depends. There are certain places where React is very opinionated and others where it's quite unopinionated. For example, React is unopinionated in terms of how you express your view logic since it treats your UI as a black box and looks only at the output. But it's opinionated in the sense that we really encourage idempotent functions, a minimal set of mutable state, and very clear state transitions.

I've built a lot of stuff with React, and I have a team that's run a lot of stuff with it. From all that experience, I

can tell you that whenever you run into a bug in a React application, nine times out of ten you're going to find it's because you have too much state in there. We try to push as much mutable state as possible out of applications to get to what I like to call a fully normalized application state. In that respect, yes, we're very opinionated, but that's just because a lot of React abstractions don't work as well if you have too much mutable state.

I think Angular is actually less opinionated in that regard, but it certainly has opinions about how you need to compose your application. It's very much a model-view-presenter type of architecture. If you want to create reasonable widgets, you're going to have to use directives, which are very opinionated.

TC Another thing I noticed right away about React is that it's very component oriented. What was the reason for going in that direction?

PH We actually think of a component as being quite similar to a JavaScript function. In fact, the only difference between a function and a component is that components need to be aware of a couple of lifecycle hooks about themselves, since it's important they know when they get added to or removed from the DOM as well as when they're going to be able to get their own DOM node. The component is a fundamental building block on top of which we've built our own internal framework. Now a lot of other people out in the open-source world are also building on top of it.

We emphasize it because it's composable, which is the one thing that most separates React components from Angular directives and web components like partials and templates. This focus on composability—which I see as the

ability to build nested components on multiple layers—not only makes it easier to see what's actually going on, but also gives you flexibility in terms of how to structure and display data, while also letting you override behaviors and pass data around in a more scalable and sensible way.

PAUL O'SHANNESSY This also has a lot to do with how we build applications on the server, where we have a core library of components that any product team can use as the basis for building their own components. This idea of using components is really just a natural extension of the core way we build things in PHP and XHP, with the idea simply being to compose larger and larger components out of smaller components.

PH Those product teams tend to be made up of generalists who work in all kinds of different languages, which is to say they're not necessarily experts in JavaScript or CSS (Cascading Style Sheets). We strongly discourage the average product engineer from writing much CSS. Instead, we suggest that they take these components off the shelf, drop them into whatever it is they're doing, and then maybe tweak the layout a little. That has worked really well for us.

PO That way we end up writing good code pretty much across the board since there are fewer people going off into crazy land writing CSS. Basically, this just gives us a way at the top level to control all that.

For all the ways in which React simplifies the creation of user interfaces, it also poses a learning curve for developers new to the environment. In particular, those who have worked primarily on monolithic systems in the past may find it challenging to adopt

more of a component-oriented mindset. They will also soon find that React is opinionated about how state should be handled, which can lead to some hard lessons and harsh reminders whenever people stray.

TC There's a lot about React that's appealing, but where are the sharp edges that people ought to look out for before diving in? What kinds of mistakes are likely to make their lives more painful?

PH Most of the pain points are almost certain to be about state. State is the hardest part of building applications anyway, and React just makes that super-explicit. If you don't think about state in the right way, React is going to highlight those bugs for you very early in the process.

Most of the pain points are almost certain to be about state. State is the hardest part of building applications anyway, and React just makes that super-explicit.

TC Give me a concrete example of how people might think about state in the wrong way.

PH OK, I'm looking at a site powered by React that was launched earlier today. It looks like the page has four main components: side navigation, a search-results list, a search bar, and a content area containing both the search bar and the search-results list.

When you type in the search bar, it filters the results to be shown in the results grid. If I were to ask you where that filter state should live, there's a good chance you would think, "Well, the search-results list is what's doing the filtering, so the state probably ought to live there." That's what intuitively makes sense.

But actually the state should live in the common ancestor between the search box and the search-results list, sort of like a view controller. That's because the search

box has the state of the search filter as well as the search results. Still, the search-results list needs access to that data as well. React will quickly let you know, "Hey, you actually need to put that in a common ancestor."

PO If you were building that same UI with Angular and used a directive for the search box and then another directive for the search results, you would be encouraged in that case as well to put your state in a common ancestor. This would mean having a controller hold the scope variable, since that's where you'll find the search text to which both of those directives would then bind. I think you're actually looking at a pretty similar paradigm there.

PH Good catch. But I think there's still a distinction to be made in that React components are building blocks that can be used to construct a number of conceptually different components or objects. You could use a React component to implement a view controller or some pure view-only thing—whereas with Angular, the controller is distinct from a directive, which in turn is distinct from the "service," which is how Angular describes those things you shove all the other logic into. Sometimes it makes sense just to make all those things React components.

DS In this case, if you were building the UI with React, what would be the common ancestor? A React component?

PH Yes. I would use React components for everything.

DS When I was starting out with React, I think one of the hardest things for me to grasp was this idea that everything is a component. Even when I walked through an example on the React website that included a comment box and a comment list, I was surprised to learn that

even those were treated as components. I also found myself getting lost in the relationships between those components. I wonder if you find that to be a common problem for other new React developers.

PO For people who are used to building more monolithic things, that often proves to be a problem. At Facebook, where we've always coded in PHP, we're accustomed to building microcomponents and then composing them, so that hasn't proved to be such a huge problem here. Anyway, what I think we've always encouraged is that, whenever you're thinking about reusing something, break it down into its smallest elements. That's why, in the example you cited, you would want to separate the comment box from the comment list, since you can reuse both of those things in other parts of your application. We really do encourage people to think that way.

PH We also encourage that you make stuff stateless. Basically, I like to think people are going to feel really bad about using state. I know there are times when it's a necessary evil, but you should still feel dirty whenever you have to resort to doing that. That's because then you'll start thinking, "OK, so I really want to put this search state in only one place in my app." Generally, that means you'll find the right spot for it since you're not going to want to deal with having to synchronize states throughout your application. And you won't have to if it lives in only one canonical place.

DS What other major differentiators set React apart from other JavaScript frameworks?

PH We haven't yet talked about the idea that React, as a general way of expressing user interface or view

hierarchies, treats the DOM as just one of many potential rendering back ends. It also renders to Canvas and SVG (Scalable Vector Graphics), for example. Among other things, this means React can render on the server without booting up like a full-browser DOM. It doesn't work like it's just some other domain-specific language on top of the DOM. Basically, React pretty much hates the DOM and wants to live outside a browser as much as possible. I definitely see that as a huge differentiator between React and the other JavaScript frameworks.

PO We've basically seen the same thing happen with WebGL or any other generic rendering platform. It just goes back to the question of immediate vs. retained mode, where you soon discover that as long as you can output something, it really doesn't matter. You just blow away whatever was there before.

DS I'm also curious about the functional programming aspects of React. In particular, I'm interested in knowing more about which specific functional principles you've adopted.

PH The truth is, we're actually a bunch of functional programming geeks. In part, that's because if you truly subscribe to the Church of Functional Programming, you can get a lot of performance benefits for free. For example, if your data model is serializable and you treat your render method as a pure function of your properties, you get server-side rendering and client-side rendering for free since both of those end up being pure functions of the same data on both sides of the wire. That way, you can guarantee that when your application initializes, it will get into the same state on both sides automatically. That

The truth is, we're actually a bunch of functional programming geeks. In part, that's because if you truly subscribe to the Church of Functional Programming, you can get a lot of performance benefits for free.

can be really important if you have a very stateful kind of object-oriented mutative system, since then it becomes much, much harder to synchronize those two states otherwise.

The other advantage has to do with optimizing your apps. We have a hook called Chute Component Update, where you can replace React's diff algorithm with a faster custom one. Also, many functional programmers really like to use immutable data structures since they let them quickly figure out whether something has changed—just another example of how you can get free performance benefits this way.

TC In the immutable data structures vein, one really powerful library I've heard about is David Nolen's Om.

PH That's a very cool piece of technology. It's for ClojureScript, the version of Clojure that compiles to JavaScript. What makes Clojure really cool is its persistent data structures, which basically are really fast and easy-to-use immutable data structures.

What that means for us is that if you have a post on Facebook and somebody likes it, that gives you a new *like* event that should be reflected on the like count appearing on that post. Normally, you would just mutate that, but then you would have no way of detecting whether the change actually happened or not, which means you would basically need to re-render the whole thing and then diff it. From that diff, you would learn that only that particular part of the UI actually changed. But if you were using immutable persistent data structures, instead of mutating the like count, you could just copy the story object and, within that copy, update the like count.

Normally, that would be a very expensive way to go, but in Clojure the copy isn't expensive since it has a way of doing it where it shares the pointers with all the other parts of that data structure and then allocates new objects only for whatever actually changed. That's a good example of an abstraction that's quite complicated under the hood and yet manages to present a very, very simple user interface—something that's extremely easy for people to reason about.

TC I assume that could also help with undo/redo capabilities.

PH Right. When everything is immutable, everything gets simpler. Om undos and redos basically just keep around pointers to the previous state and the next state. When you want to undo, you just pass the old object into React, and it will update the whole UI accordingly.

TC The whole thing?

PO When your state is serialized into one object at the top level, all you do is pass that through and re-render it—and you're done. With some of the Om examples I've seen, it just snapshots the state at every point and then gives you a UI that indicates how many states you have. Then you can just drag back and forth on that. Or you could start doing some fancier things with the help of trees to produce a really advanced undo system.

PO I should also point out that React clearly is not purely functional. We also have some very imperative steps and hooks that let you break out of the functional paradigm. But in an ideal world, you don't have any other sources of data, so everything is at the top and just flows through—

meaning everything ends up being a very pure output of these render functions.

DS A bit earlier, you used the term "referential transparency" to describe the way React renders UI. Can you explain what that means?

PH Basically, React components have props, parameters that can be used to instantiate those components. You might think of them as function parameters. In order to say, "I want to create a type-ahead with these options," you can just pass in the options list as a prop.

The idea is that if you render a component using the same props and states, you'll always render the same user interface. This can get a little bit tricky, though. For example, you can't read from the random-number generator because that would change the output. Still, if you handle this as a pure function of props and state and make sure you don't read from anything else, you can probably see that this is going to make testing really fast and easy. You basically say, "I just want to make sure my component looks this certain way when it gets this data." Then, since you don't have to take the Web-driver approach of clicking on every single button to get the app into the right state before double-checking to make sure you've got everything right... well, it becomes pretty obvious how this makes testing a whole lot easier—which, of course, makes debugging easier as well.