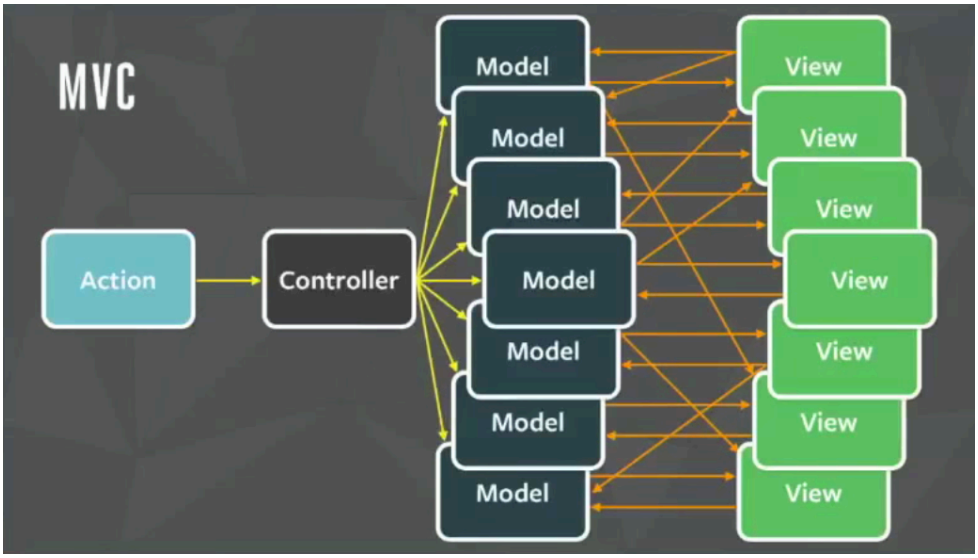
## Reasons not to use MVC

- MVC works well for small apps but not big apps
  - Creates a fragile system:
    - It doesn't make room for new features because when you add a lot of models and views there's an explosion of arrows
    - Model triggers something in the view and the view triggers something in a different model
      - Having a lot of this going on are those arrows you see going all over the place here and if your app was going in constant loops for something and you wanted to debug and see how that infinite loop between all these relationships is happening and where, it's really difficult:

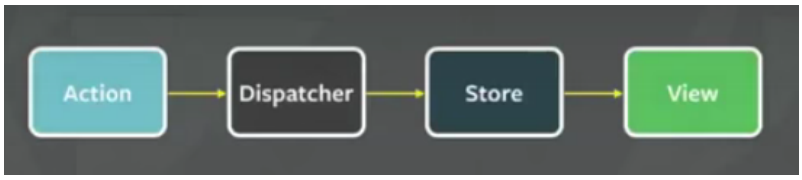



Dave: so just like when UB or others talk about having your dependencies point downward and no dependencies pointing upward, this is the same concept except it's for the web and for views and view models

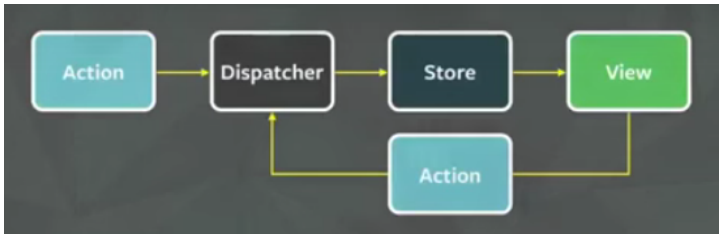
## Flux

Is a single direction data flow

- Avoids all the double arrows that go in both directions that make it really difficult to understand the system




**Action** comes into the system --> **Dispatcher** which is the traffic controller for the whole thing --> **Store** which is the data layer that updates whenever you get a new action --> Then the **View** is re-rendered whenever the store is saved or something has changed



**Views**

- Can throw another action into the system

**Dispatcher**

- The main piece that enforces that there's no cascading effects
  - So how this is done: once an action goes into the store you can't put another one in until the store is completely done processing

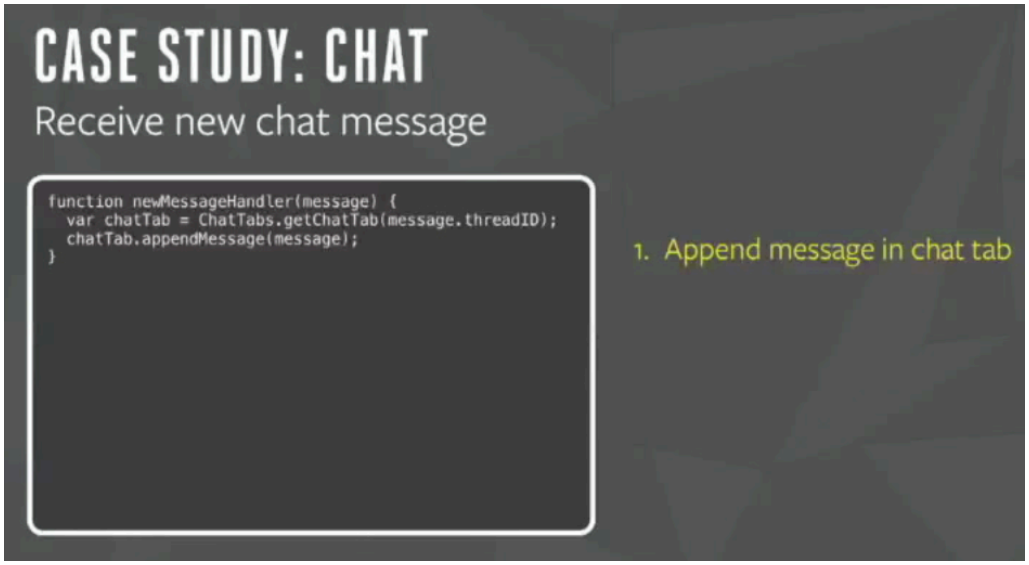
## Real World Use Case

We knew we had a problem with chat in 2011 when we introduced the subscribe button on our blog and the first comment on that blog post was "please improve chat system" with 898 likes. And this was one of our more polite comments. We got a lot of comments that just said "Fix Chat". And that become a pretty frequent refrain in all of our user comments.

So how did we get to this place? How did we get to the point that we were annoying our users so much they just wanted us to fix chat. They didn't care what else we were launching, they didn't care what we were improving, they just wanted us to fix chat.

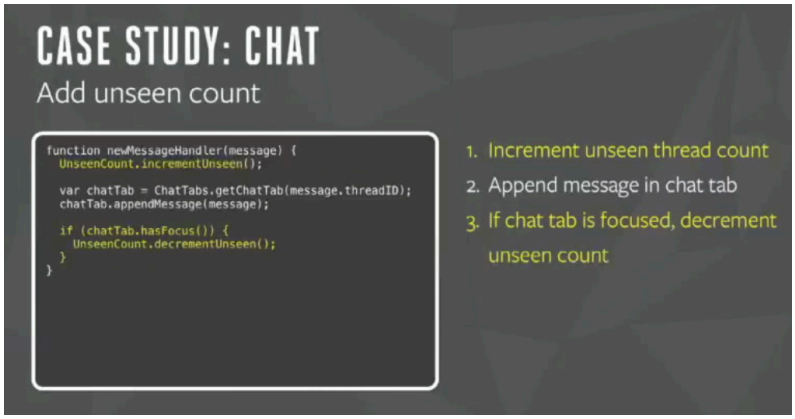
Well, to answer that we have to go through sort of the chat features as it evolved.

We started with something pretty simple. You have a chat tab, shows a list of message. Whenever you get a new chat message, you just append it. It's pretty simple, the code was pretty manageable.

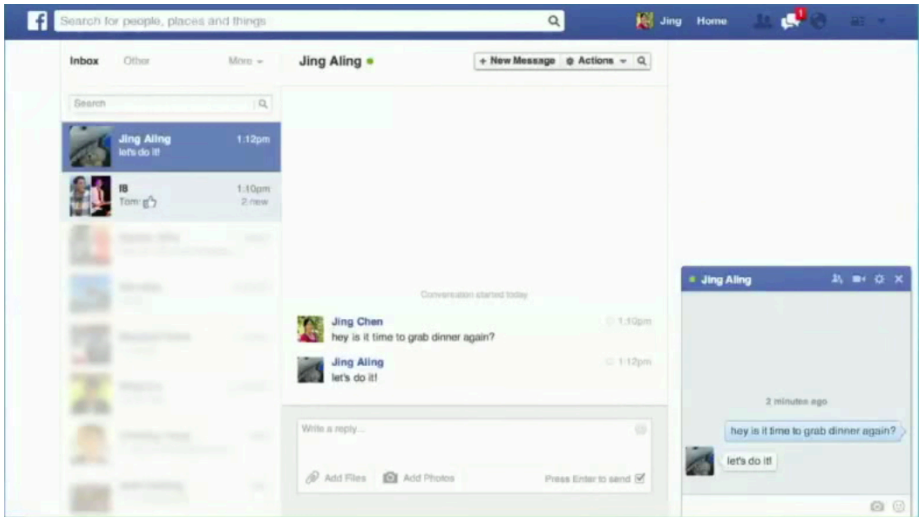
Later on we also launched the messages tool that shows you the number of threads that have unseen messages on them.


That was kind of developed independently from the chat tab, but whenever a new message comes in, we also want to handle that unseen count. So this code gets a little more complicated:




We have an unseen thread count that we want to bump every time we get a new message. And if we think that the user is looking at it we also want to decrement it.

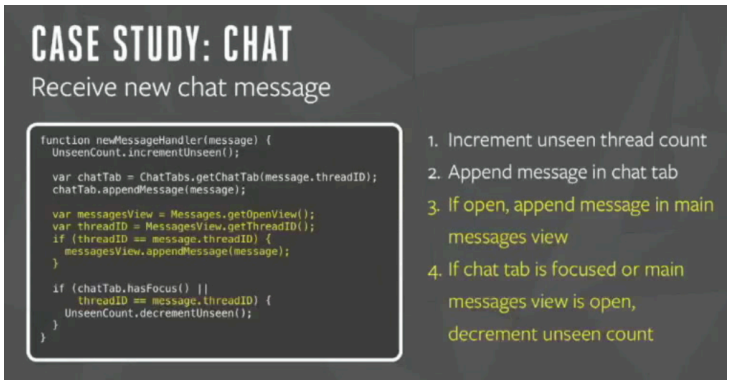
A couple months later we also introduced the larger messages view



And with that we made this handler even more complicated.

This is what we had before:


And now we have to account for the messages view and decrement the count under the right conditions: if we think the user is looking at either one of the views:




So what are the problems here?

- One is that the code really has no structure
- It's very imperative which makes it fragile
- We lose a lot of the original intent behind it
  - It's hard to tell just by looking at the code that you want to keep the messages view and the chat view in sync
  - It's hard to tell that you only want to increment and then decrement. You don't want to do either of those more than once
  - As we add more features, this code just only gets longer

And having something imperative like this led us to the situation where we had our most annoying chat bug happen over and over. That is…users would get an unseen count and there would be no unseen messages behind it.


Everyone is used to seeing these numbers, clicking on it, and seeing something behind it. That's exciting but when they get that number and they click and there is nothing there, they refresh the page and it's gone…that's really annoying.

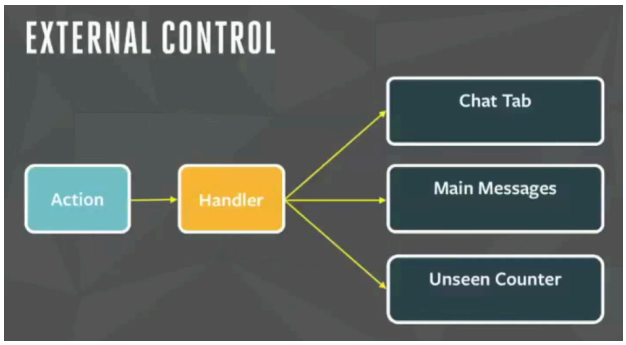
And that's what lead to the whole fix chat user feedback.

And it wasn't like we didn't fix the problem. We would always fix some particular edge case but this problem would keep coming back just because the whole system was fragile. And that was a good sign that something was fundamentally wrong here.

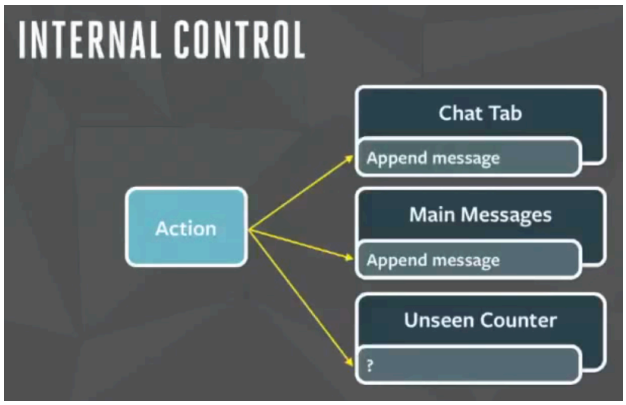
## So how can Flux help with something like this?

If we're looking at the structure of the code we had before we have sort of the fundamental pieces. We have chat tab, main messages, and unseen counter.

And then we have a handler that goes in and changes each one of their state. They each have their on state but the handler goes and modifies them. All of the logic to modify state based on a new message coming in is on the handler itself. And so those 3 different pieces no longer have the ability to maintain internal consistency. They basically don't have that information; they've already given that control to the handler.




So instead what we want to do instead is internalize that control and move all of the control into individual pieces so that the state is right next to the logic that updates that state.




So for chat tab and main messages it's pretty easy, you just want to append a message whenever you get a new message.

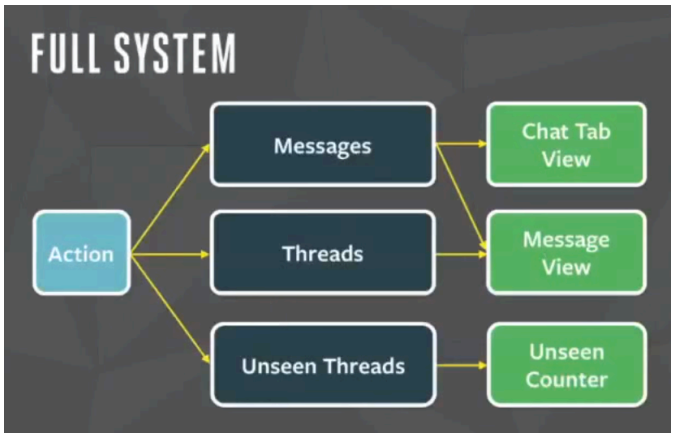
But it's a little less clear what to do with the unseen counter. We know that you still want to increment that counter when you get a new message…but where do you decrement that counter? You don't have enough information at that point. What this structure makes clear is that we need more explicit data.

What we really want to do is to **keep track of the unseen threads**. Then what we can do is whenever a new message comes in we can add that thread to the list of unseen threads. And then further down the line, we can have the UI decide when to mark that thread as seen and then remove it from the thread list.

Having more information this way means that thread list stays more consistent. If we put in 2 more unseen messages…no harm done. The second one is a no out because that thread has already been removed.

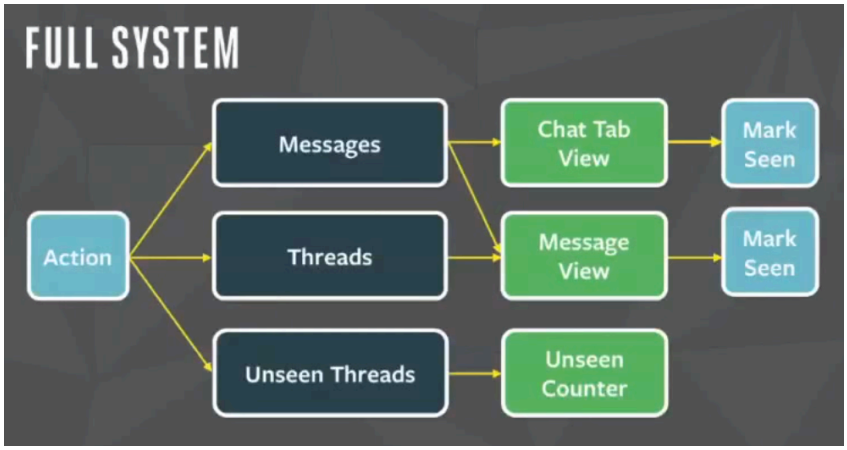
What we learned here is that we should use more explicit data on the client instead derived data. The unseen threads list is the explicit data. And the unseen count is the derived data which is the length of that list. Doing it this way gives the client more control and better ability to stay consistent overall.

We also **wanted to separate data from view state**. This isn't much different than MVC…because it's very non-controversial, but it does add to our system diagram a bit:



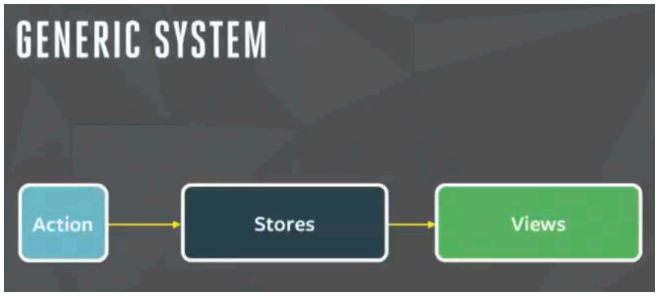

So above, we have an action coming into the messages threads and unseen threads state. And whenever those update, they tell the views that something changed and that you should go re-render.

The biggest difference between **Flux** and **MVC** is that we want to avoid cascading affects by preventing nested updates. So what does that mean? Well if we look at our system, that means we want to let the data layer completely finish processing before we trigger any additional actions. In this case, the actions come in, the messages threads and unseen thread stores finish processing that, and then tell the views to update. And only then can the views say I want to mark something as seen



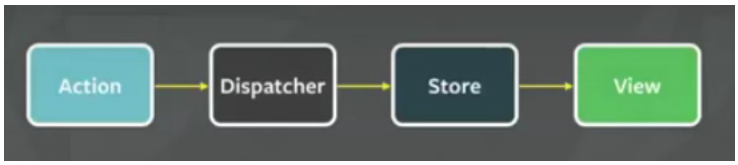

and that goes back and through the system. The benefit here is that you have no arrows going in the other direction.

More generally, what this diagram looks like is that you have an action that comes into the stores, the stores update the views, and all of your actions take the same structure…nothing goes in the other direction.



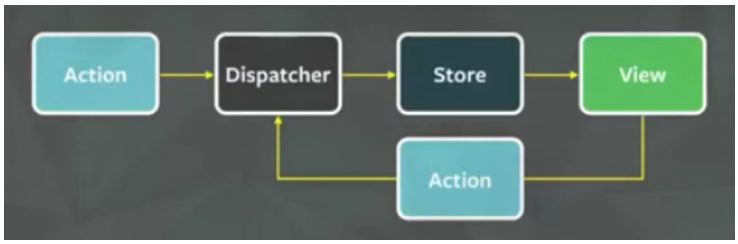

And having all of your actions going through the same flow, makes it really easy to onboard a new person…we found that we didn't need to sit down with them and show them what's going on because all they had to do was trace how one action flowed through the entire system and have a pretty good idea therefore how the rest of the system worked.

So going back to the original diagram, we still have the action, the store, and the view. We also have a dispatcher in here that is the traffic controller of the whole system. What it does is that it ensures that until your store layer is completely done, the views or anything else can't put an action through the system:




Instead, if it gets an action while the data layer is still processing you can throw. And it's a good constraint to have because it's a guarantee for all future devs and yourself, that if you understand where your action starts, and know what the changes are in the Data Layer, you know all of the downstream effects. It doesn't have anything that propagates and cascades on down that crazy graph with all the errors. Instead this is goes in a single direction and once you finish a single layer, you're done.

If we sort of imagine this **Flux** diagram with more stores and views, it all process down to the same thing because all of the arrows go in a single direction. With MVC you can't sort of form a simple mental model because the arrows go in both directions. But with arrows going in a single direction you can collapse it down to exactly this diagram, and that's your mental model for the whole thing:

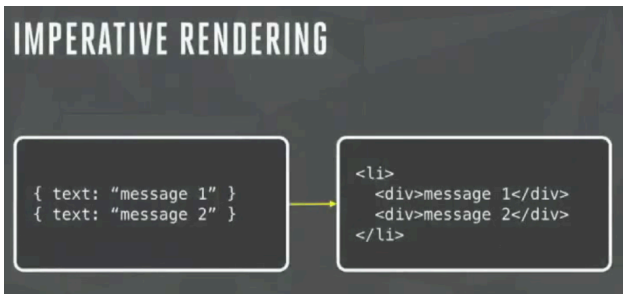



So when we switched over to **Flux**, in our chat system what we found is that it improved our data consistency so when you looked at your chat tab, and your messages, they'd always be in sync. It also became much easier to find the root of a bug. We could usually say well, it's probably that module is the most likely suspect. So that shortened our iteration cycle.

We were also able to write more meaningful unit tests as a result. Because all of the state and all of the logic that updates that state is in one place…we could basically test that module by saying assume you're in state A. Given an input, now we can test that it's in an expected state B and that state B is consistent. This was effective and helped us to avoid making the same mistake twice.

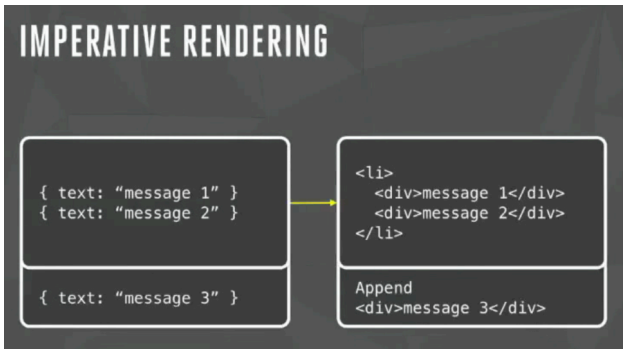

The real-world impact of this is that before when we had the bug, it'd always come back to haunt us. We'd make some changes like adding features, it would re-introduce that bug again but that doesn't happen now.

So at this point we're in a pretty good state with our data flow. But our rendering layer was still a little bit difficult:




The initial render is pretty easy. We have a list of messages coming in from the data layer that renders into the (something…she mumbles) list of messages. So still pretty manageable. But whenever we wanted to update it, we didn't want to just re-render the entire thing. We wanted to optimize it a little bit for the new message case because that happens so frequently and you might have a really long list of messages and the users might notice when you start re-rendering every time, the UI flickers.

So what we had instead is that the render layer would look at the messages that the data layer gave it and then it finds that it has one more message than it had previously and so it would translate that into an append operation:




The problem with this code is that it became really fragile over time. We had to account for different message types. We had to account for messages coming in out of order. And this code got more fragile over time.

We had one guy on the team who knew it and how to handle it but nobody else on the team wanted to touch it. Anyone else was willing to jump in on any other bug in the data layer because they were comfortable with it. But intuitively the knew that this code was fragile and too difficult to change without breaking something else and they just wanted to infer to the one guy on the team who understood everything.

So that's a bad sign.  What we really wanted for our rendering system was to be able to always re-render no matter what.  So we wanted to give it a list of message. We know how to register that list into the dom.

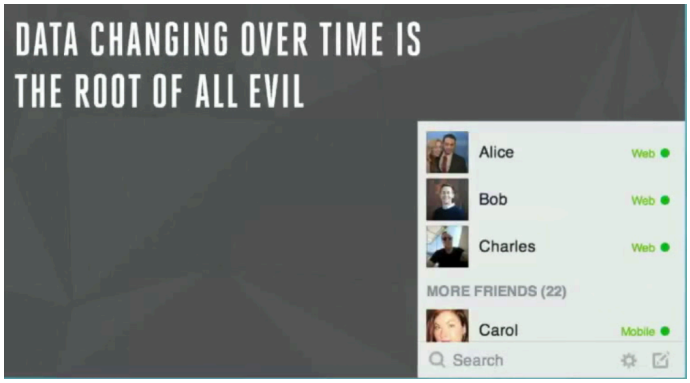
**What we wanted is a rendering library that would do that efficiently for us without interrupting the user experience.**

## React

This is where **React** comes in to solve this problem.

**Flux** gets your data model into a really predictable state.  It's really easy to understand what's going to happen in your system.  But Jing brought up a very important point which is we have to render this consistent data model to the user interface and that's a pretty difficult problem.  So difficult in fact that we built a user interface library called **React** designed to solve the hardest problem in this space….which is that data changing over time is the root of all evil.

So here's an example piece of user interface from our chat product, it's the buddy list.

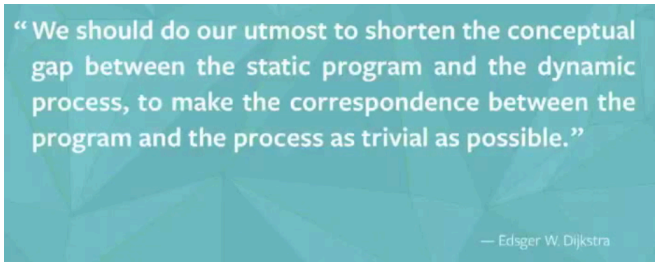



Now I'm going to read off a series of mutations to this buddy list.  And I'm going to ask you what the state of this should be at the end.  So I'm going to ask you what it should look like.

So...Alice went offline, Bob went offline, Steve went online, Bob went online, Charles is idle, and Charles is on mobile.

Now…what does this user interface look like?  Does anybody know?  I don't know, I have no idea…because processes that are spread out over time are very difficult for humans to understand and hold in their head.  Just these imperative actions that mutate state; it's very difficult to figure out what that state is at the end of the sequence of those mutations.

And I'm not the first person to discover this.  Dijkstra has a really great quote about this:

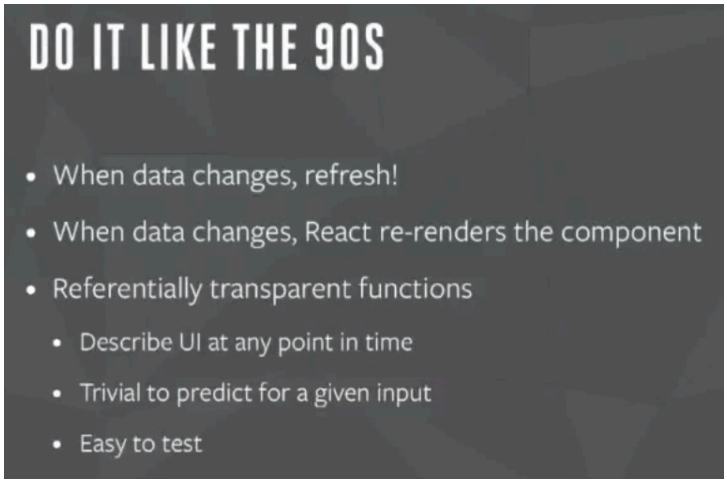



Now what Dijkstra is saying here, is he's saying we need to take this series of imperative mutations…these things that are changing over time…and try to make it look like a program that executes at a single point in time.

So if you go back to that buddy list example, if I presented a consistent snapshot of the data at any point in time, it's very easy for you to visualize what that interface should really look like. You can then imagine how we would draw this on the screen:

```
[
  {name: 'Bob', client: 'web', idle: false},
  {name: 'Steve', client: 'web', idle: false},
  {name: 'Charles', client: 'mobile', idle: true}
]
```

So back in the 90s when we were doing server rendered apps all the time and we didn't have a lot of JavaScript running on the client, this is actually how we wrote user interfaces. So when the data changed in your app, you simply clicked the refresh button in your browser. And what would happen is that the browser would go to the server, fetch some data from the DB, which would be a consistent snapshot of the state of your application. It would then turn it into an HTML page and flush it down to the browser.




So what we've done with **React** is we've taken the mental model of a 90s era server rendered web app and we've brought it to the client. So whenever data changes whether it's from **Flux**, or parse, or firebase, or backbone or whatever you're using to get data on the client, **React** will render the entire component every single time, just like a 90s era web app.

Now when I say a 90s era web app, what I really mean is a RESTful web service. So if you're bringing that model to the client you get a lot of the benefits of RESTful architecture for free. So one of the important parts of RESTful architectures is item potency. And a similar concept is referential transparency. So when you model your UI as a referentially transparent function, you get a lot of benefits. So think about how the web works today. When you type a url in you always get the same resource no matter what. With **React** when you put in data, you always get the same user interface rendered at any point in time. And whenever the data changes you can simply re-fetch and refresh just like in the 90s.

So one of the reasons people use RESTful architectures so much is they're so predictable, it's very easy to cache them, you understand the performance characteristics and how to scale them out simply because for the same set of inputs you get the same set of outputs.

And a lot of great stuff falls out of this like testability. A lot of our unit tests for our user interfaces at FB started out as click on this button, mutate this state, a series of imperative actions, and then we would make some assertions along the way.

Since adopting this model, we do a lot less of that and a lot more of sending in a consistent state and doing some assertions on actually what was rendered based on that state.

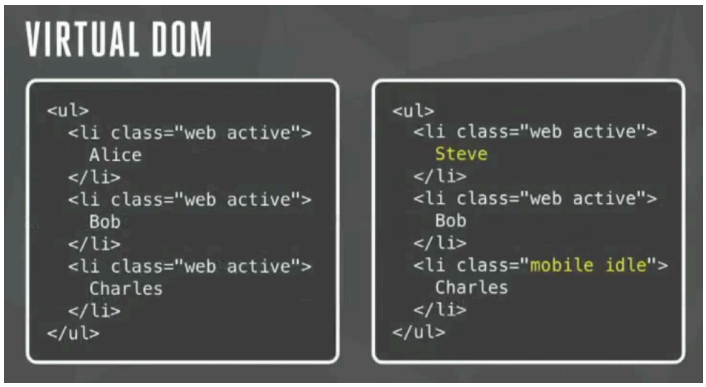
Now this sounds like a pretty great model for building applications but we can't do this with the regular browser APIs that we have today. And the reason for that is because the DOM is stateful. So if were to re-render all of the DOM nodes when anything changed in our application, we would have a really poor performance profile and a really bad user experience. The browser was not designed to be constantly creating and throwing out DOM nodes. And additionally if someone is typing into a text field or has an inertial scroll going on, if we were to throw out those DOM nodes and re-create them, we would kill that scroll momentum and scroll position and it would just be a really bad user experience.

And this actually isn't limited to the browser. This is any stateful UI toolkit. So imagine implementing this in iOS and WebKit. People generally don't destroy and re-create view hierarchies on every change because of the performance characteristics and because of the user experience problems.

So what we did at FB is we spent the past couple of years building what we like to call a virtual DOM abstraction which is implemented in this library called **React**. And what the virtual DOM lets us do is it lets us model our UI in the way that I just described…conceptually re-rendering everything in an inexpensive and in a good way for user experience.

So here's how it works. Whenever the data changes, we re-render your application to the virtual DOM and we create a new tree of virtual DOM nodes. Then we diff that new tree of virtual DOM nodes with the previous tree…the one that existed before the data changed. Then we can isolate what were the changes between the two virtual DOM trees, and we can compute the minimal set of DOM mutations necessary to make the real DOM look like the virtual DOM. We put that into a queue and we flush that queue to update the UI.


So here's an example of how that works:



Remember the buddy list example, we have the initial state that I showed you in the screenshot on the left, and we have the state that we want to get to after all those mutations on the right. Now again, we can't just throw out the DOM nodes and re-render them for the performance and user experience reasons I just mentioned.

So we have a virtual DOM diffing algorithm that goes through and actually highlights what changed between these two data structures. So I've colored those changes in yellow.

And what the system will do is that it will eventually turn it into imperative DOM mutations just like people write on the web today.

Now…remember…this is actually the most difficult part of the engineering of the UI…these imperative mutations that change over time. If you think back to that buddy list example, this is the same thing. This is just the code representation of those mutations:

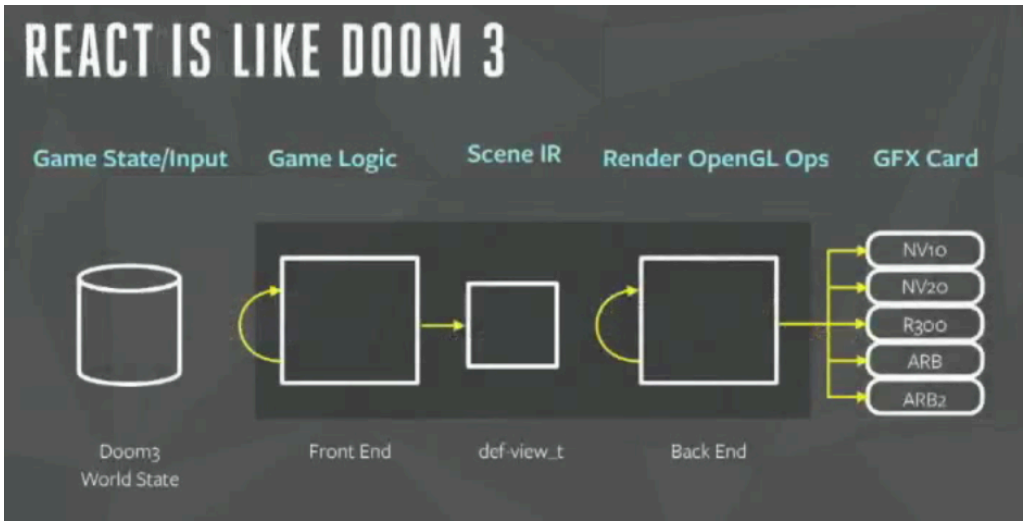
```
buddyList.childNodes[0].textContent = 'Steve';
buddyList.childNodes[2].className = 'mobile idle';
```

But the key here is, we push the complexity of those mutations into the framework. So the end user of the library doesn't even need to think about it.

So they think about how their UI looks like at any point in time as a complete snapshot, and the system figures out the imperative mutations needed to update the UI, to make it look correct.

So if you've been following so far, this might not seem like a very performant system because we're re-rendering the UI all the time to this virtual representation, and we're diffing a lot of things that may not have actually changed. In this example, only two pieces of data changed out of that whole DOM structure.

The good news is that this is how high performance games work:

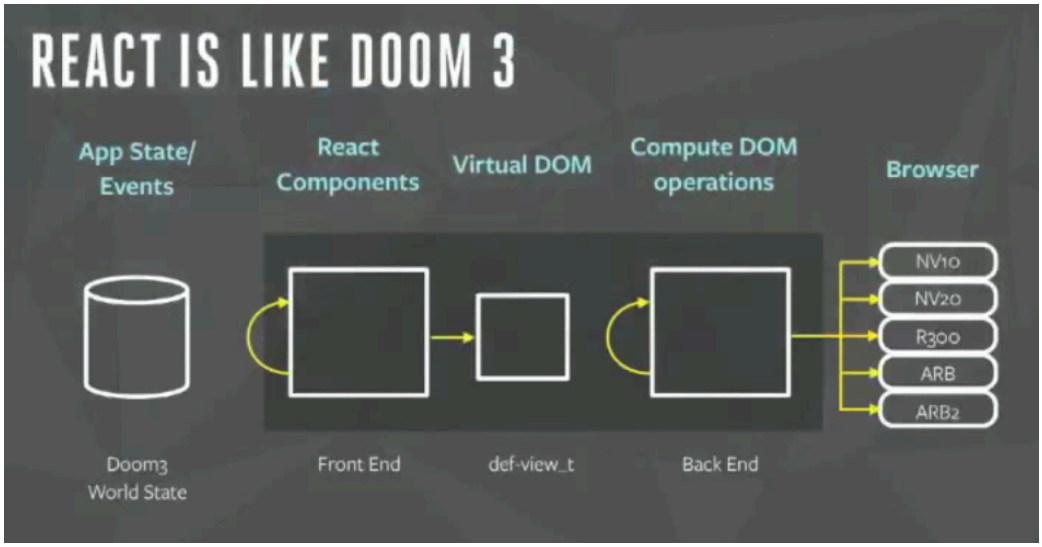



One of my colleagues built a game a couple years ago called DOOM 3, and here's how it works:

- We have a game state that represents a consistent state of the world
  - So we have user-input events that come in and mutate that state and we have network events that come into the game server
- That world state goes into the front-end of the game

- The front-end of the game represents all of the rules of the world, so how players move around, how weapons work, that type of thing
- And out of that component comes what we like to call a scene intermediate representation which is a logical declarative description of what actually needs to be drawn
- Now they take that logical description of what needs to be drawn and they put it into the backend which turns it into OpenGL operations that are imperative and go over the bus to the graphics card

Now the architecture we came up with **React** and this virtual DOM abstraction is very similar:

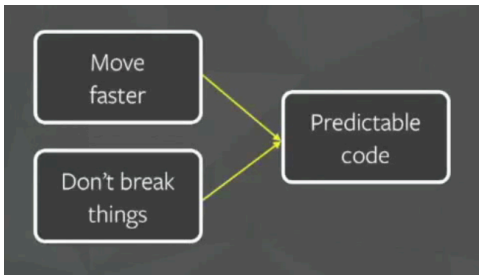



- We have your application state…which is updated by server events and by browser events
- That goes into your **React** component, written in JavaScript, which describes the business logic of your application, the rules of your app
- And then that generates a virtual DOM representation, rather than a logical representation of a 3D world
- We pass that virtual DOM representation into the backend, which then computes imperative DOM operations rather than Open GL operations and renders them to a browser rather than a graphics card

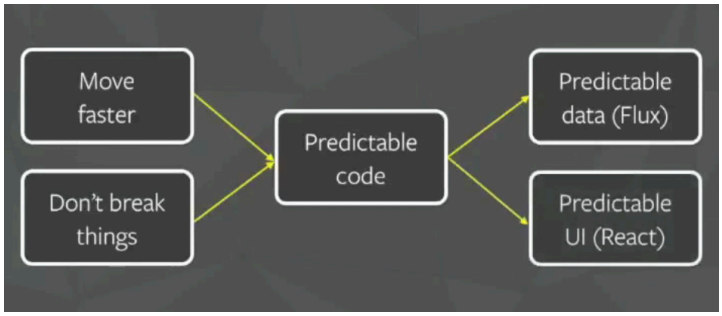
So I think that this is pretty cool technology.

But Facebook doesn't build technology for the sake of technology, it has to have impact, it has to make our developers and our users lives better.

Now Tom touched on this earlier, our whole goal with all of this is to improve quality for the same amount of engineering time. So we have to basically move faster without breaking things.

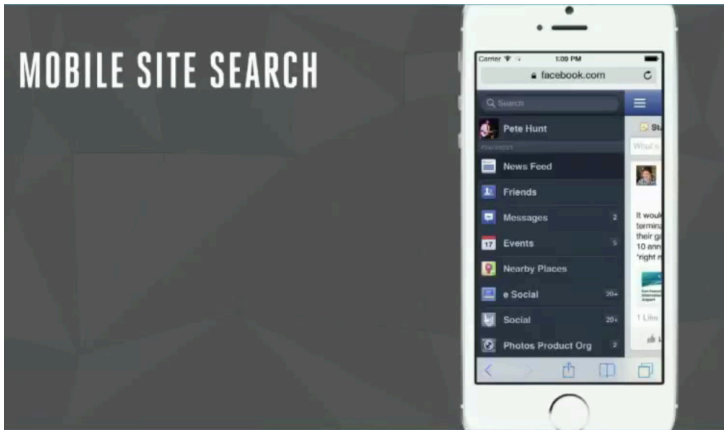



We identified that focusing on making your code predictable is a really great way to improve these things. And the two systems we came up with are **Flux** and **React** for making your data layer and your UI more predictable.




So…lets take a look at how **React** in particular has made our lives a little bit better at Facebook.

So we have a search product on our mobile site:

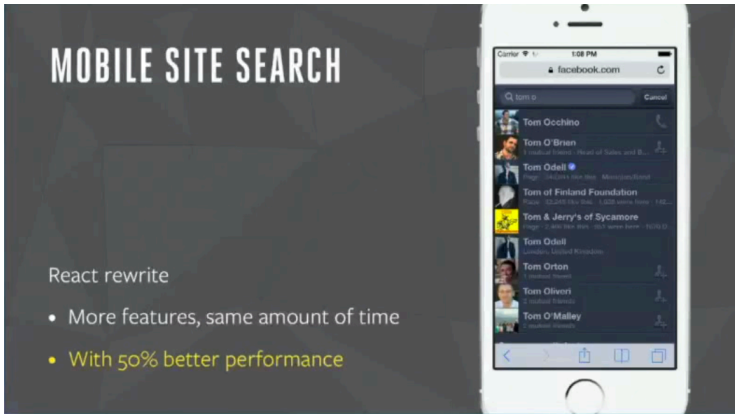



And it's kind of been the same thing for a while and we wanted to add some features a couple of months ago. So we decided to use our best of breed web technologies which right now is **React**, to re-write the system, and add these new features.

And what the team found is they were able to build more features in about the same amount of time it took to build the previous version. And they were learning **React** while they were doing this.

And this is a really great benefit of having predictable code and fundamentally solid ideas underlying your architecture.

And by the way, it performed about 50% better than the older version. So a lot of great things fall out having predictable correctness and performance characteristics of your code.




But it's not just helping us here at Facebook. It's also become a pretty great open source project for us as well.

So a couple of years ago, we started building this at Facebook. Then we acquired **Instagram**.


**Instagram** is a great native mobile company, but there's not a lot of web expertise there. Facebook on the other hand has a lot of web expertise.

So we wanted to build Instagram.com which was a web experience for **Instagram**, and we wanted to use **React**.

So in order to do that we had to extract **React** out from the Facebook specific use-cases running on our big PHP platform and custom JavaScript packaging, and make it work on **Instagram's** DJango stack that runs on Amazon EC2.

It was very difficult for us to kind of reuse the underlying backing infrastructure at that time. But by the time we open sourced it to the world on GitHub, we had already dog fooded this with millions of people for many many months and already figured out a lot of the kinks.

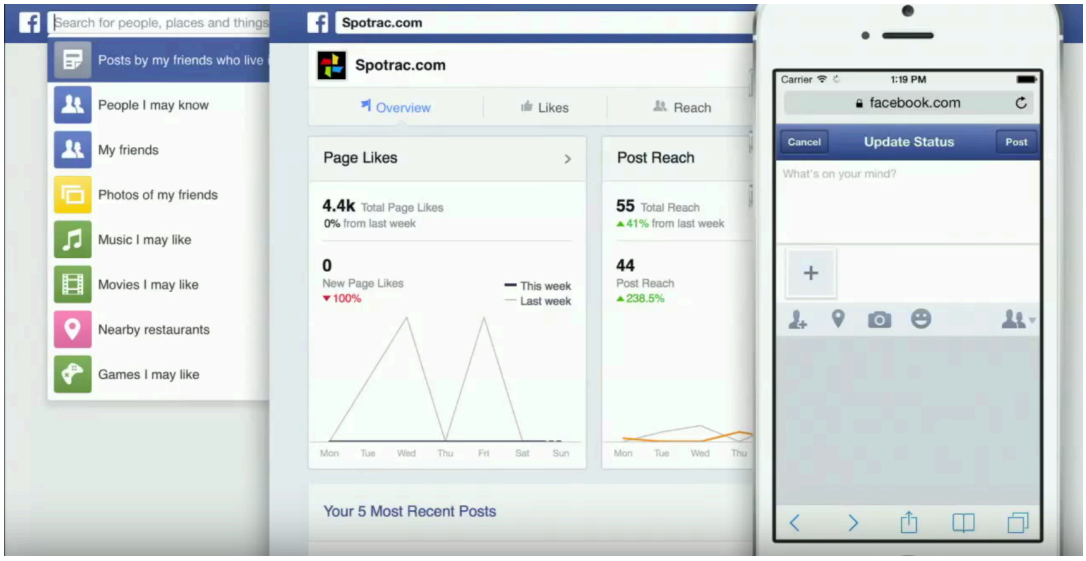

And today we have a lot of great companies in the community using this. A couple off the top of my head are **Airbnb**, **Vemeo**, the **New York Times**. They're all pushing this to production. And a really great story is Khan Academy, they were one of our early users and now an engineer at Khan Academy is I think our #2 contributor to **React** which is really great.

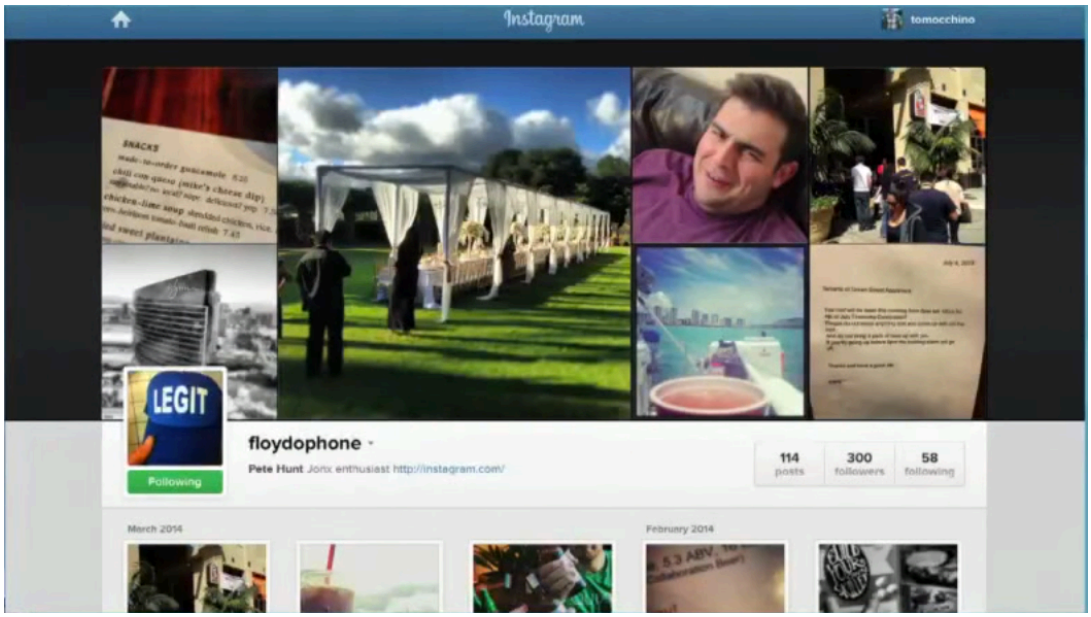
And you're going to be hearing about a lot more of our open source projects later today, so keep that in mind.

But it's not just open source. We're starting to bet big on **React** everywhere on the web.

So our graph search product is powered by **React**. Our page insights product which has a lot of high performance data visualization is also powered by **React**.

And on our mobile site I did mention our search system but also uploading photos and navigation are powered by **React**. If you go to Instagram.com, the entire page is one big **React** component. That includes basically all of the features.




## Conclusion

So I've talked a lot about how this has made our lives a lot better on the web and about the browser DOM and JavaScript, but these are fundamental software architecture ideas.  And we're starting to see these ideas appear in our native mobile apps as well.

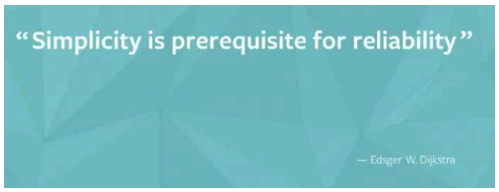
So our data layers are starting to move away from a traditional MVC observable system, and they're starting to present consistent snapshots of the entire data model at any point in time.

And our UI toolkits are starting to resemble **React**.  They're starting to be reactive components that present a completely declarative view of how the UI should look.

And if you stick around for Scott Goodson's talk later you're going to see even more influence and cross pollination of ideas here.

I'd like to leave you with a quote from my favorite computer scientist, Dijkstra:

He says Simplicity is a prerequisite for reliability




We want to build better systems in less time. In order to do that we need to be able to predict what our systems are going to do. And you can't predict what you don't understand. So we need to make our systems simple enough to fit inside our heads when we're working on a problem, so we can understand them, predict what they're going to do, and build higher quality software in less time.

## Questions

**Question**: **How do you guys handle Routing in React**?

**Response**: We have like a much more open source stack on Instagram and we glue it into Backbone. The way that we generally model everything is in terms of this state that goes into your React component, and one of those components of state is the route. Another component of that state for example is the underlying data model. We don't really distinguish between route and underlying data model.

**Question**: **How long did it take you to refactor all your code to Flux and React**?

**Response**: So Pete did most of that, he did it in a couple months. We've been using React internally for actually a really long time now. Jordan Walk is somewhere in the back and it was actually his idea. He actually first presented this to me and kind of came over to my desk and said dude I have this great idea. And I said this is never going to work, you're nuts. And then he showed me some stuff and it actually did work. So we started using it internally for a lot of internal tools and stuff like that and then shortly after we acquired Instagram, Pete was going to work with them and build out their web presence and really wanted to use this stuff so I'd say over the course of 2-3 months, he basically divorced it completely from our Facebook build stack and kind of turned it into some that could very easily be put onto **GitHub**. And it was a team effort, it wasn't just Pete but he was kind of the driving force.

It's one of those things where we generally don't like rewriting code for the sake of rewriting code. So if there's a new feature we'll generally try it out on that and then it will gradually kind of effect the rest of the system as we start moving more and fixing more bugs.

**Question**: **I've used Angular, it looks quite similar, can you compare and contrast React + Flux to Angular**

**Response**: They do solve some of the same problems but they go at it at very different ways. React is focused a lot on treating your code as a black box. So there's no sort of observable abstraction within React. You simply say hey re-render the UI and you present a consistent view of what you want your UI to look like. With Angular you are basically passing data throughout these things called scopes which observe your data model. I think that's a very leaky abstraction. It forces you to compose your application not in terms of function and objects but in terms of directives and MVC and their flavor of MVC. So…while it does work for a certain class of applications, as you scale up you start to miss the past 40-50 years of research into how to abstract a program. So if you can push that kind of data binding concern out toward the edges of your system like React does, I think it leads to fast iteration time.

**Question**: **How does it compare with AJAX? Like jQuery where it pulls data and sort of brings it into the page without reloading the url and stuff like that**?

**Response**: There are actually two different kind of systems. So question was how does this sort of architecture compare to traditional jQuery and Ajax sort of thing. So Flux kind of manages your data flow and your stores in that system will manage the Ajax calls for you. And you can actually implement that with jQuery, you can implement that with Parse, you can do basically do anything that goes over the network and dump it into that architecture. So we kind of push that concern to a different system and Flux is the integration point.

As far as the comparison with jQuery, you can think of React as kind of filling a similar role as jQuery. So the idea of jQuery is like you want to make some changes to the DOM and put the UI in a certain state. I like to think of React as the next step in that. So with jQuery you have to think in terms of this is the initial state, this is the next state, this is the next state. With React you only think of one state and it's the current state.

We've given various presentations about this too and what we say is that mutation is really hard. jQuery makes this really easy across browsers, it kind of smooths out all the inconsistencies and it makes it so it's very easy to manipulate the DOM. React tries to abstract it a little further than that and say look…if you didn't have to worry about mutations ever at all and you just describe declaratively what your component and program looks like at any point in time or what your views look like at any point in time, it actually allows you to move a lot faster because you don't have to think about intermediary states…it's a little bit different approach