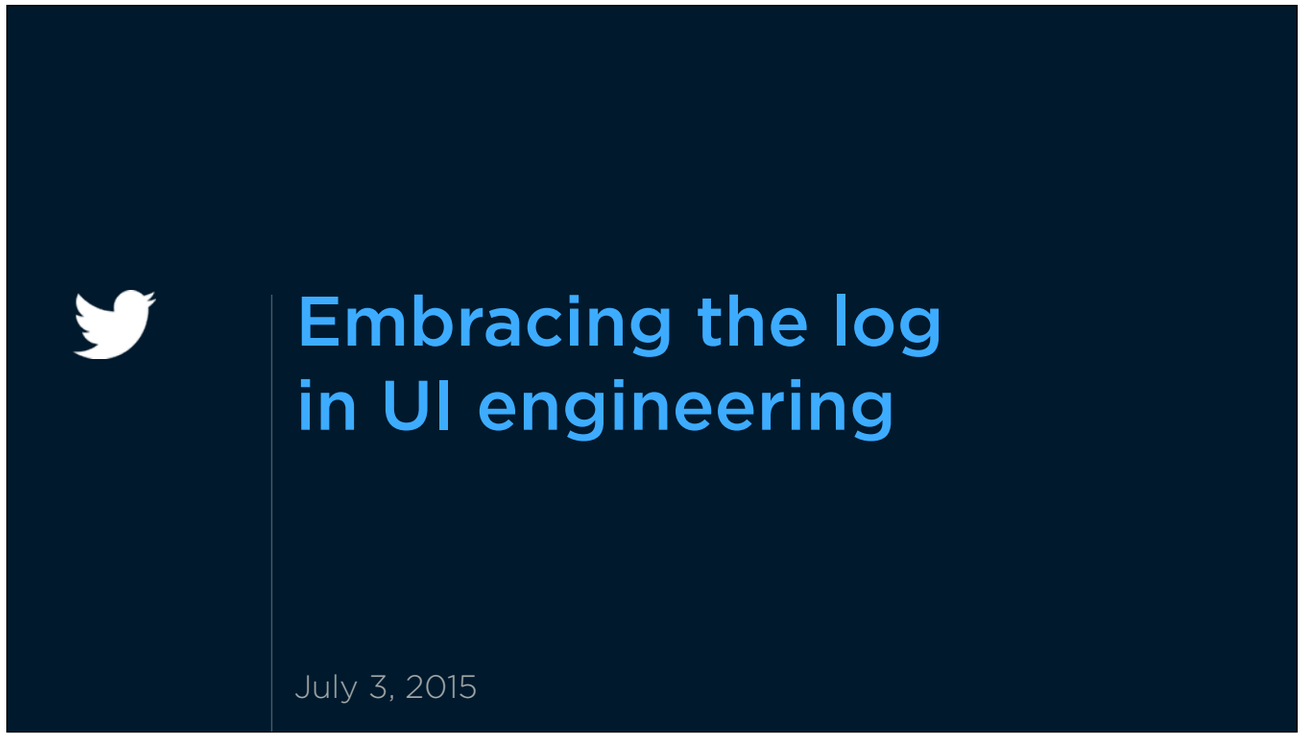
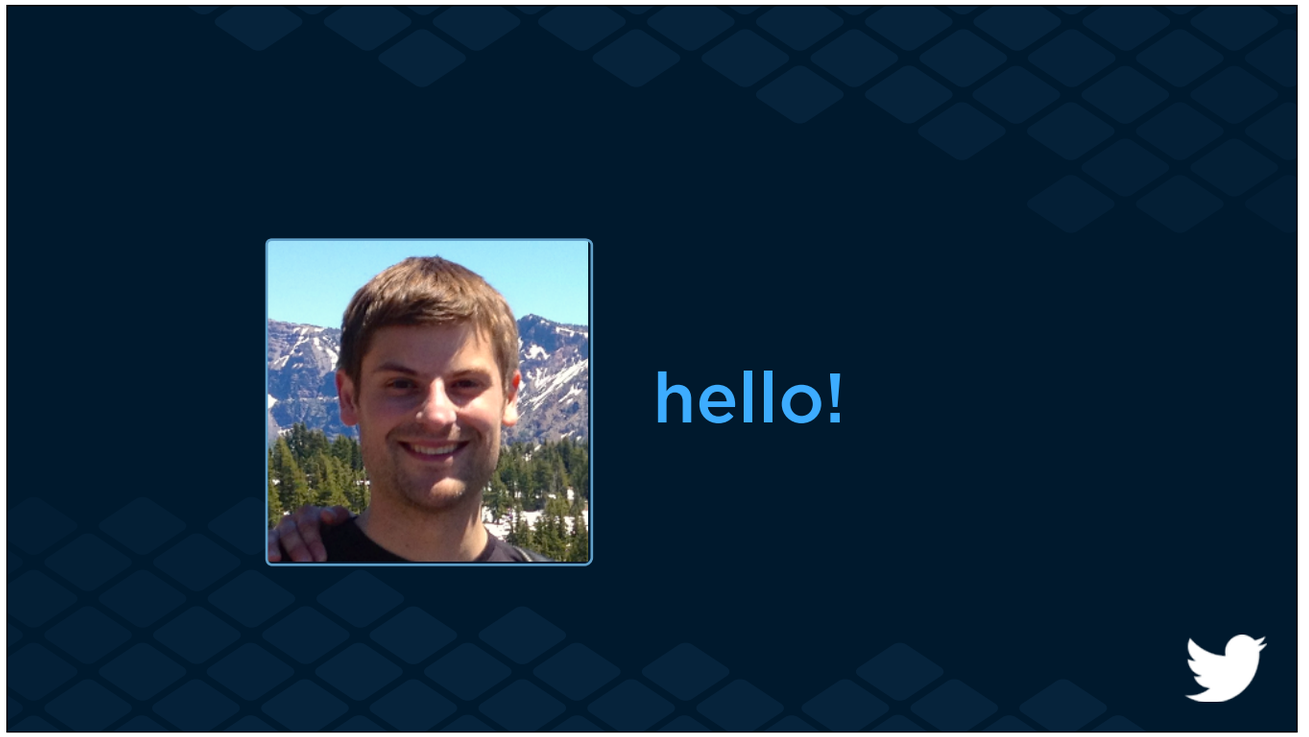


This is drawn from a part of my talk at React Europe on July 3rd, 2015. It's about why using a log to record data within a UI can enable some awesome product features.



Hi! My name's Kevin (@krob), and today I'm going to talk about why it can be useful to use a log as the central concept for organizing data and computation when building user interfaces.

First, let’s talk about what this looks like in backend systems, to set some context for what I’m talking about.

On the team I work on, Answers, we use immutable logs in several places in our backend systems. This has a lot of benefits, and I think some of these apply directly to UI engineering even though they are quite different problem spaces.

https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time

Embrace the log

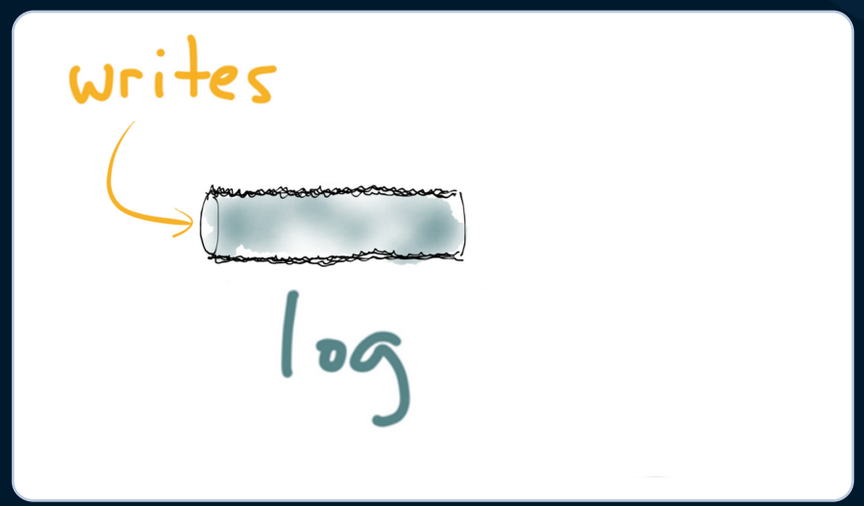
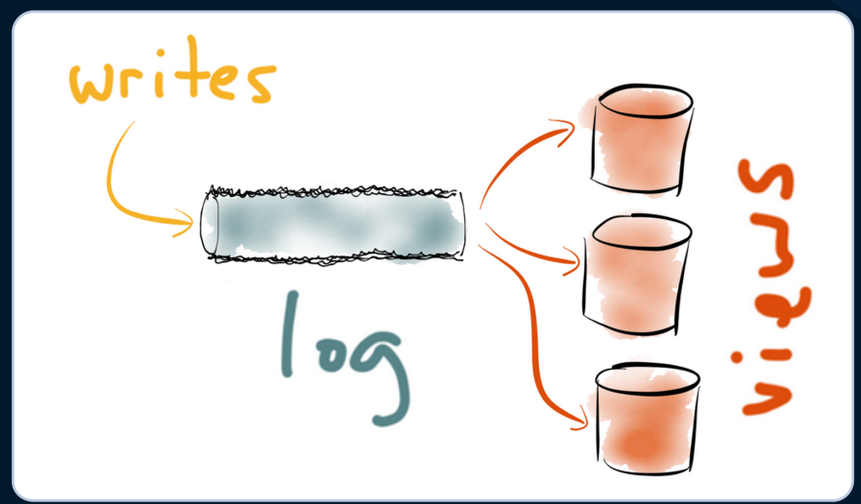


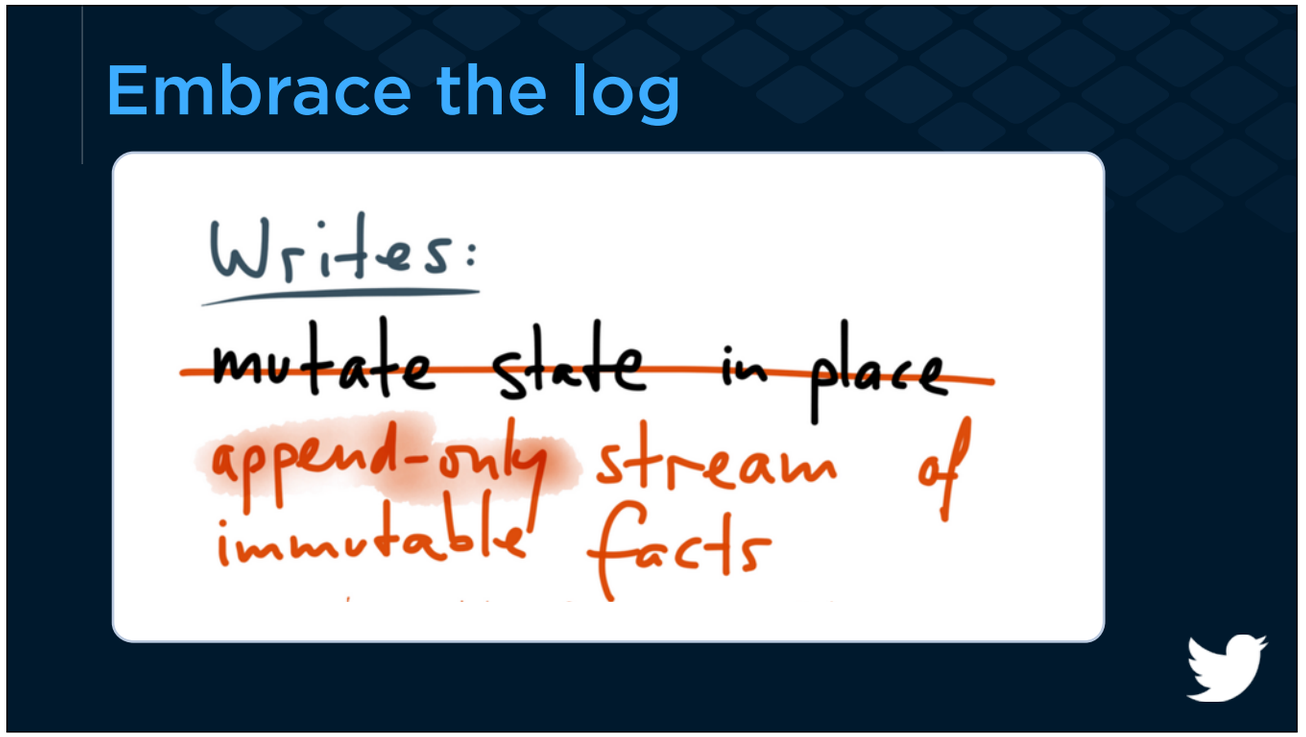
Diagram adapted from http://blog.confluent.io/2015/03/04/

As writes come in, they're written to a log. There's no computation or mutation.

Embrace the log

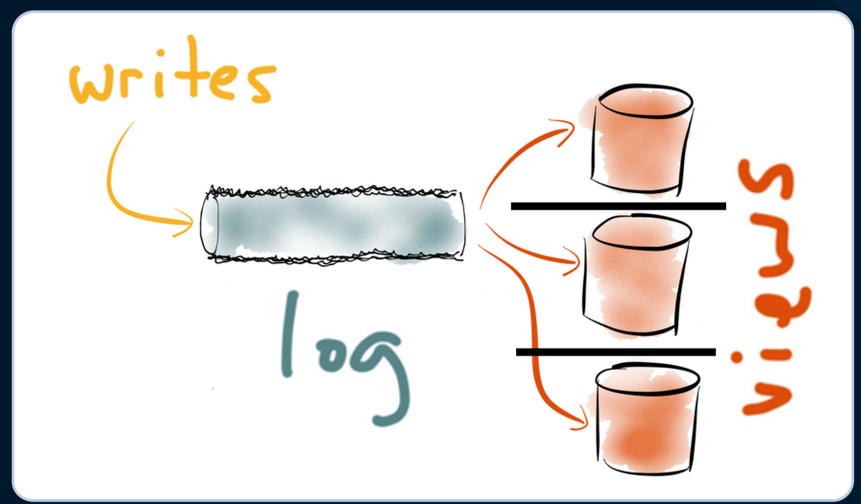


Then, separately, processing components can read from the log and compute whatever “views” they need of the data. These might be filters, aggregations, or special indexes.



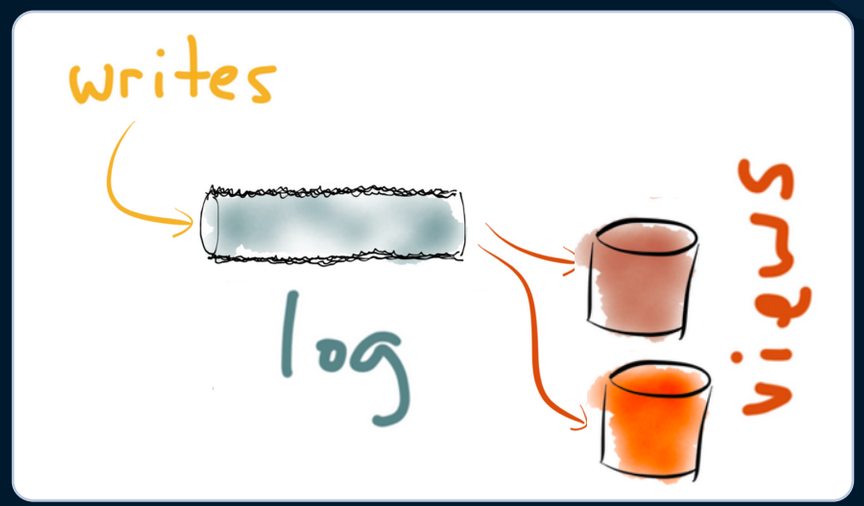
A central idea here is immutability. Instead of mutating state in a database record or field, the write is simply recorded, without affecting or destroying any existing information.

Isolation



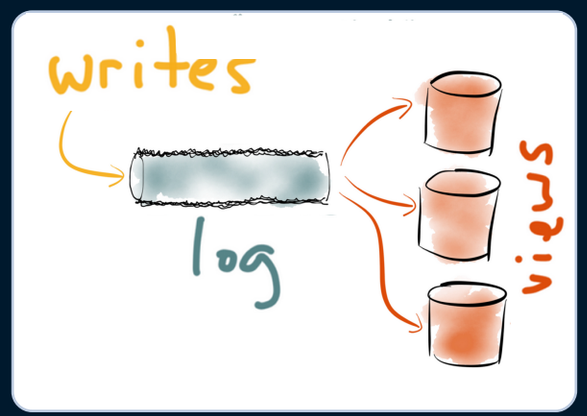
This has a lot of benefit on backend systems. One is strong isolation between different computations. If one computation has a bug, it doesn't affect the others. And if one computation becomes slow, we can optimize it in isolation.

Swapping computation



We can also swap out different computations. And since we preserve historical information in the log, we can even run computations retroactively, which is useful when building new features or fixing bugs.

More background



http://blog.confluent.io/2015/03/04/

https://twitter.com/jaykreps/status/412643213342568448

If you're a UI engineer and want more background, these articles are great reads - super clear and accessible.

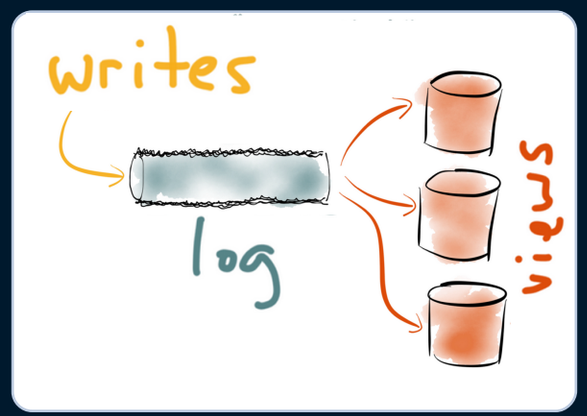
http://blog.confluent.io/2015/03/04/
https://twitter.com/jaykreps/status/412643213342568448

The beautiful diagrams here are adapted from Martin Kleppmann's (@martinkl) awesome work.

Applications to UI engineering

So, that's cool, but what does this have to do with UI engineering?

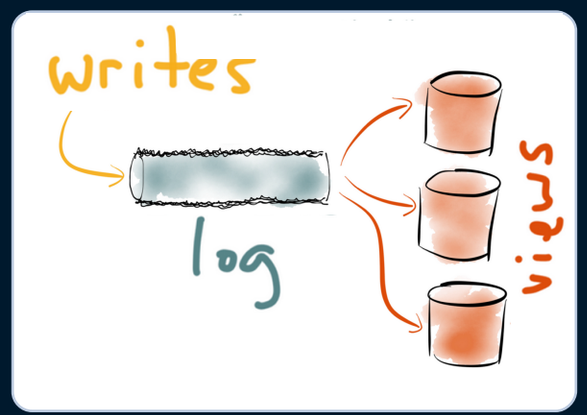
Some similarities...



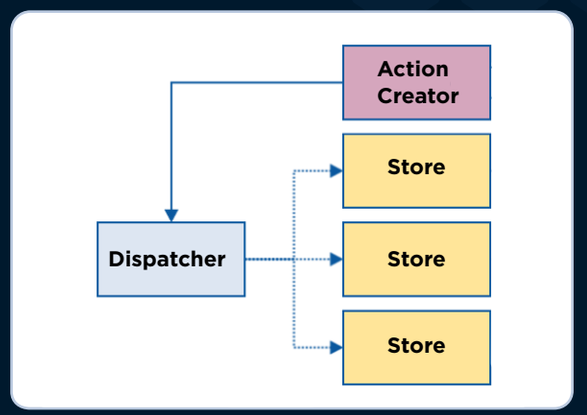
http://blog.confluent.io/2015/03/04/

Well, this is the diagram adapted from Samza, a stream processing system.

Some similarities



http://blog.confluent.io/2015/03/04/

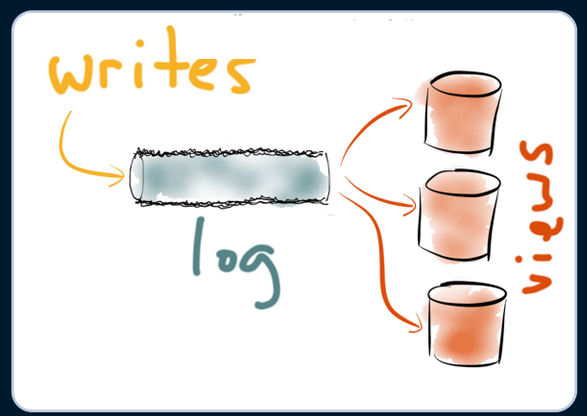


http://fluxxor.com/what-is-flux.html

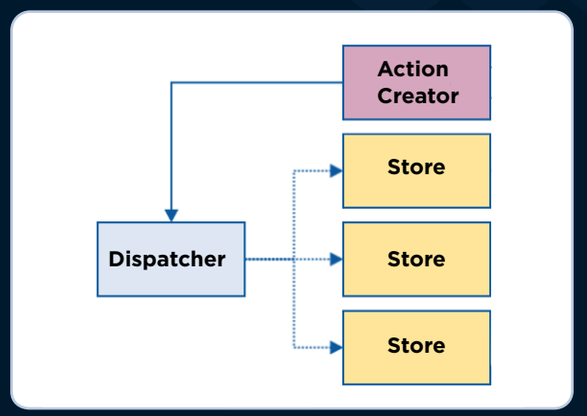
And this diagram is from the documentation for Fluxxor, a great Flux library.

http://fluxxor.com/what-is-flux.html

Some similarities



http://blog.confluent.io/2015/03/04/

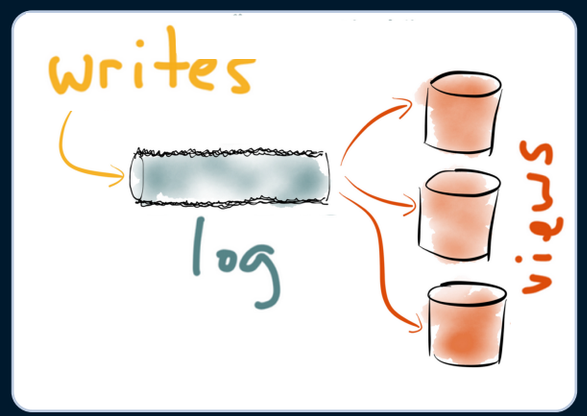


http://fluxxor.com/what-is-flux.html

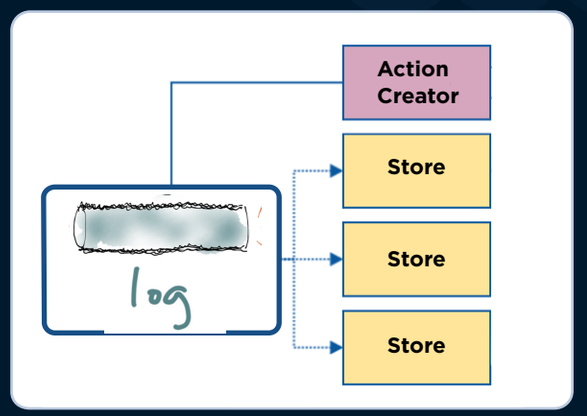
So with Flux, we get similar benefits of isolation between Stores, and the ability to write new Stores without affecting others.

But because Stores are mutable, and because we don't preserve the history of events through the Dispatcher, we can't swap out computation or Stores at runtime.

Some similarities



http://blog.confluent.io/2015/03/04/



http://fluxxor.com/what-is-flux.html

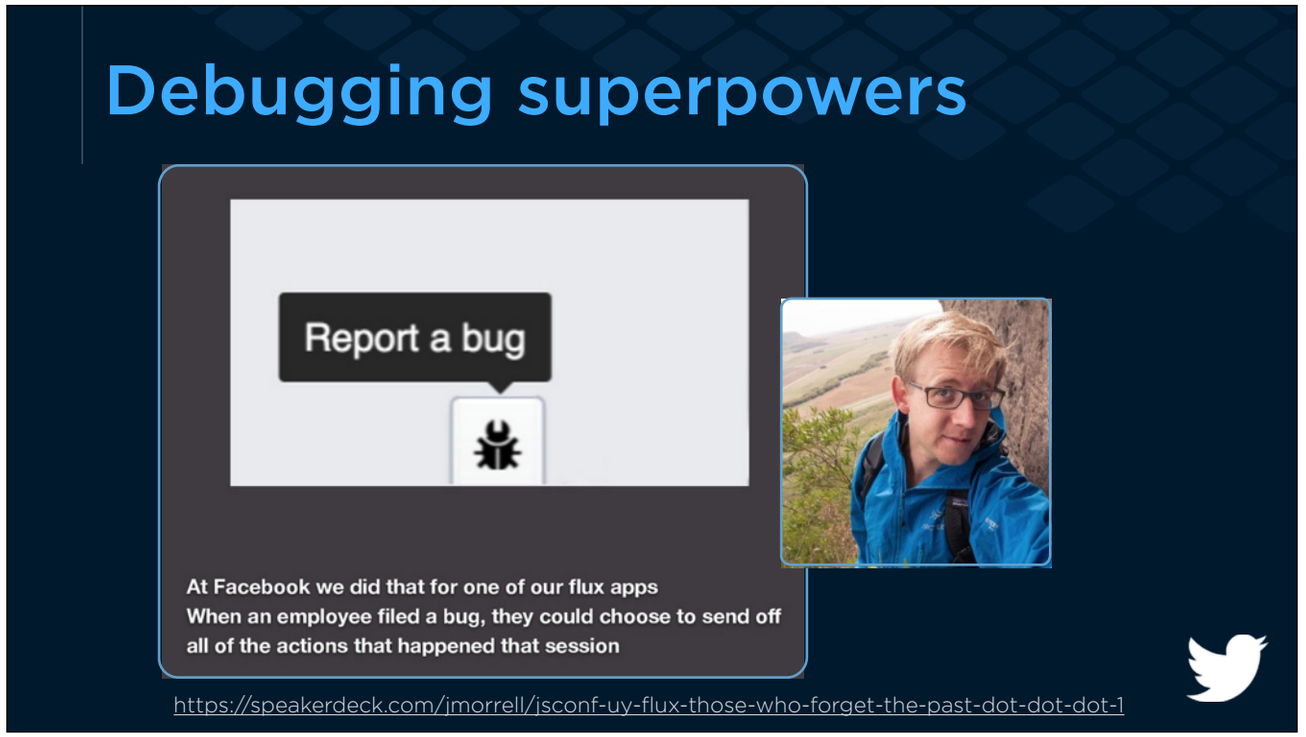
What if we could? :)

At React Europe, Dan Abramov gave an amazing talk about his work on Redux, which allows hot-reloading of Stores in development mode.

https://twitter.com/dan_abramov/status/617642370591510528?lang=en

This works because it's keeping a log of all actions flowing through the dispatcher, and because Stores are expressed as reducers instead of mutable objects.

After hot reloading, redux can re-reduce over the log, which enables hot-swapping Stores.

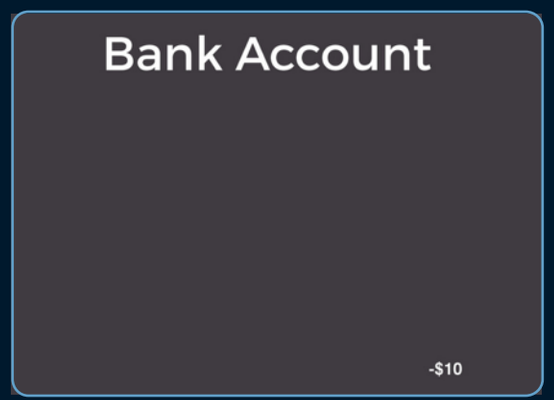


Preserving a log helps with debugging as well. Jeremy Morrell has great slides talking about how to take advantage of this in a Flux application.

A separate Store can log all actions flowing through, which lets user sessions be inspected or replayed later.

https://speakerdeck.com/jmorrell/jsconf-uy-flux-those-who-forget-the-past-dot-dot-dot-1

Debugging superpowers



https://speakerdeck.com/jmorrell/jsconf-uy-flux-those-who-forget-the-past-dot-dot-dot-1

Because when debugging, this is what we see using mutable models.

And this is what we want to be able to see. The history of what happened and how we got here.

We want to see the log.

Data layer superpowers

```
(aset js/window "undo"
  (fn [e]
    (when (> (count @app-history) 1)
      (swap! app-history pop)
      (reset! app-state (last @app-history)))))
```

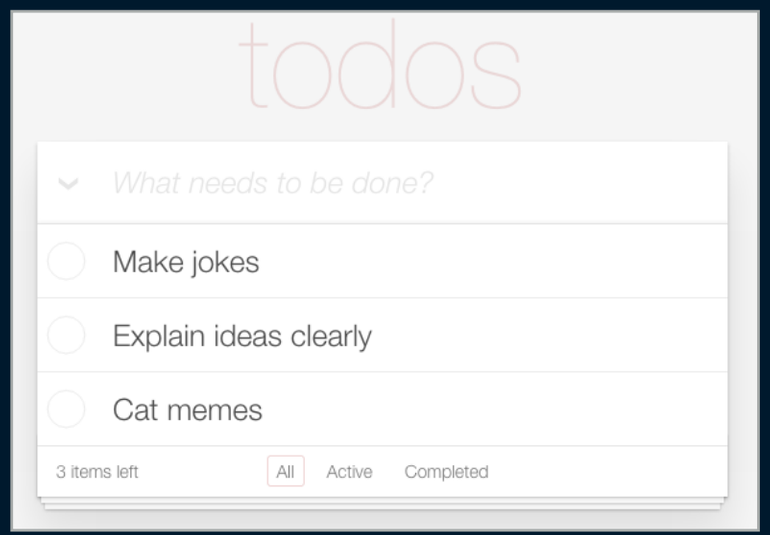
https://github.com/omcljs/om
swannodette.github.io/2013/12/31/time-travel/

But this can be even more powerful when it comes to building features for end-users like optimistic updates, real-time data systems, or systems for collaboration.

David Nolen has talked about how powerful a feature “undo” can be, and these kinds of features is really where using a log can really pay off.

http://swannodette.github.io/2013/12/31/time-travel/

Data layer superpowers



Let’s sketch out some examples.

We’ll use the universally popular TodoMVC example to see what this might look like.

Embrace the log

timestamp	key	payload
3000	USER_ADDED_TODO	{text: "Make jokes"}
3005	CREATE_TODO_REQUEST	{request: {text: "Make jokes"}}
3500	CREATE_TODO_RESPONSE	{request: {text: "Make jokes"}}, response: {id: 14}}
6000	USER_ADDED_TODO	{text: "Explain ideas clearly"}
8000	USER_ADDED_TODO	{text: "Cat memes"}
...		
12000	TODOS_LIST_RESPONSE	[{id: 14, text: "Make funny jokes"}]

Here's an example of a log. As the user is working, their actions are logged, and as requests are made to the server, the requests and responses are logged as well. Whenever the UI "learns" anything at all, it simply records that fact.

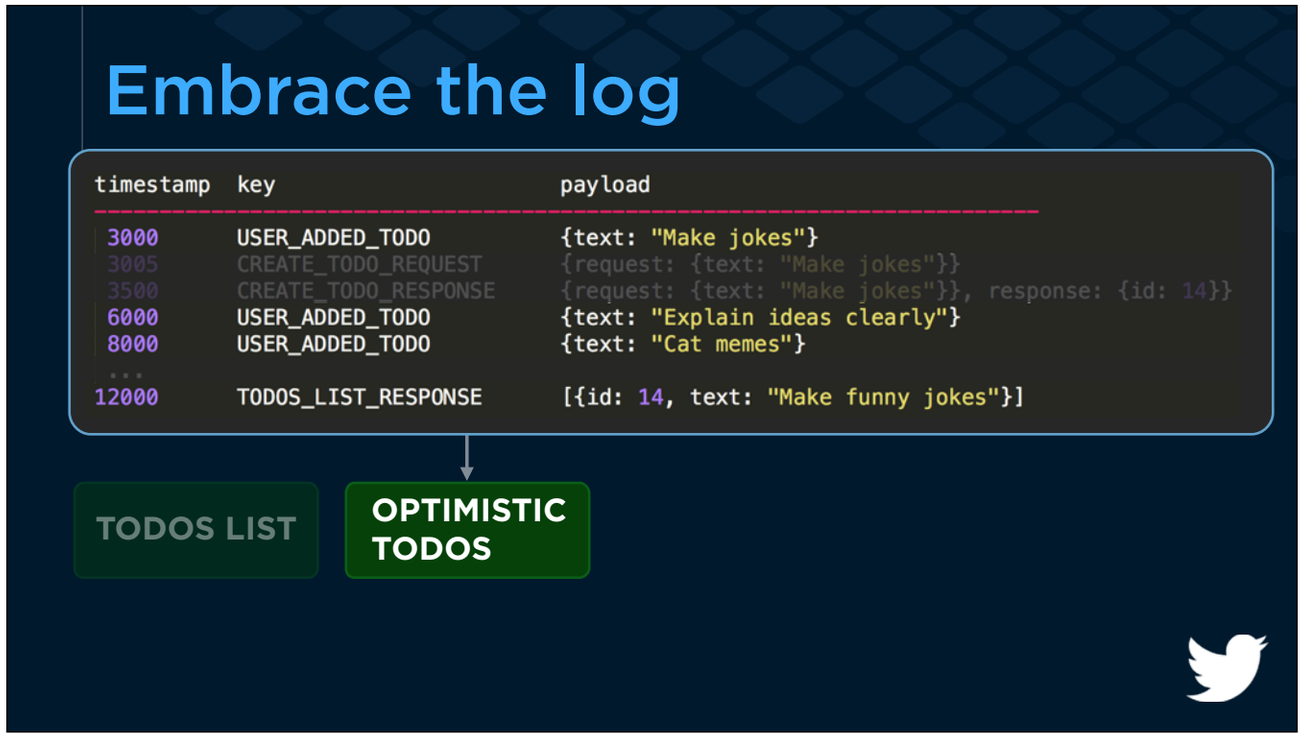
This isn't a server-side log being sent down to the UI. It's a log of everything happening in the world, from the UI's perspective.

And the log just sits there. It's immutable, inert.

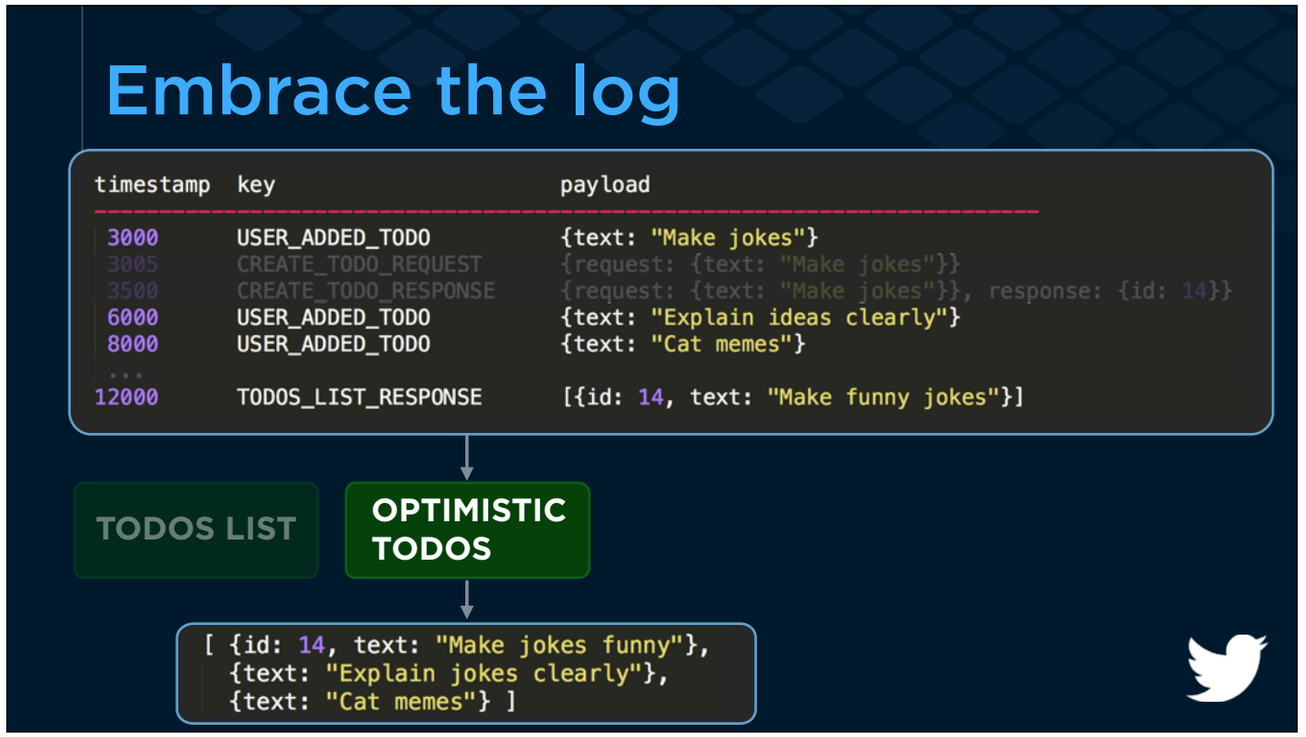
And we can perform any kind of computation on it that we want, decoupled from when the actions actually occurred.

If we want to simply show the user the list of Todos, we can reduce through the log and pull out the most recent TODOS_LIST_RESPONSE from the server.

This might have come from a specific request made in response to a user action, from polling a REST endpoint, or from data pushed down by a socket.

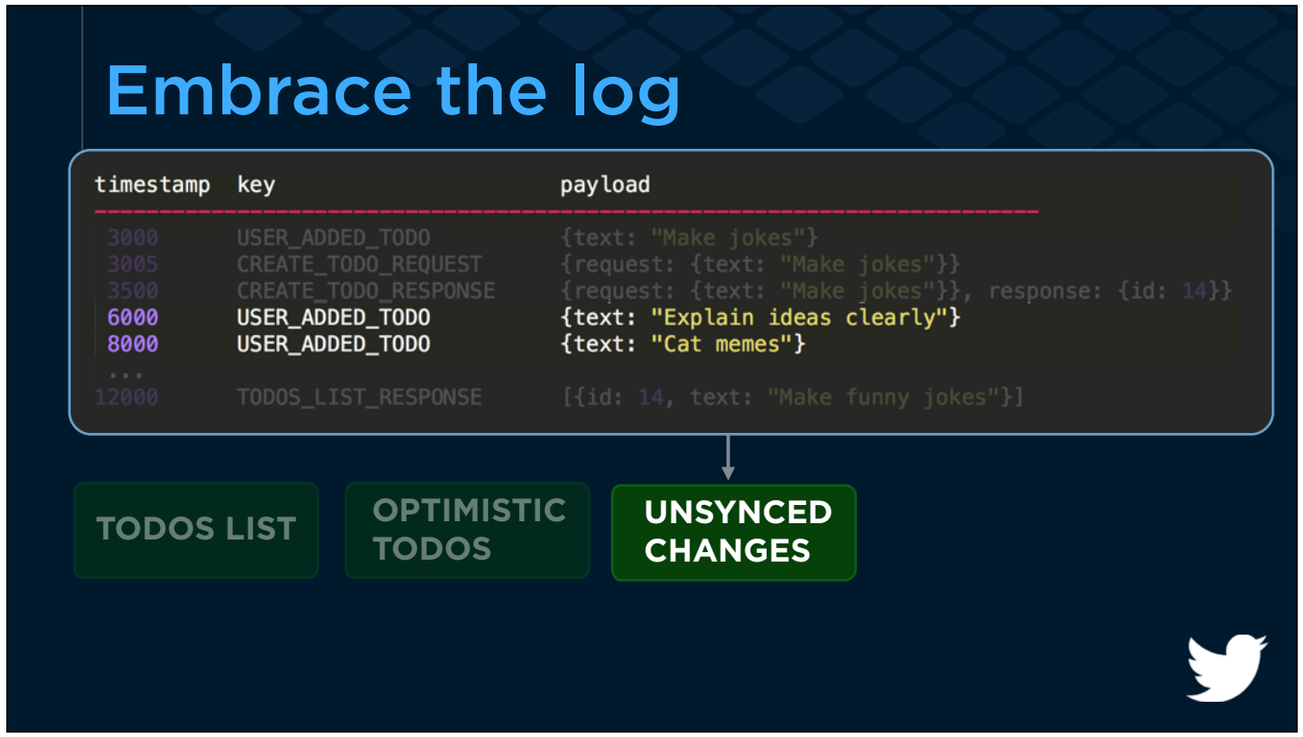


We can write a function that includes optimistic updates.

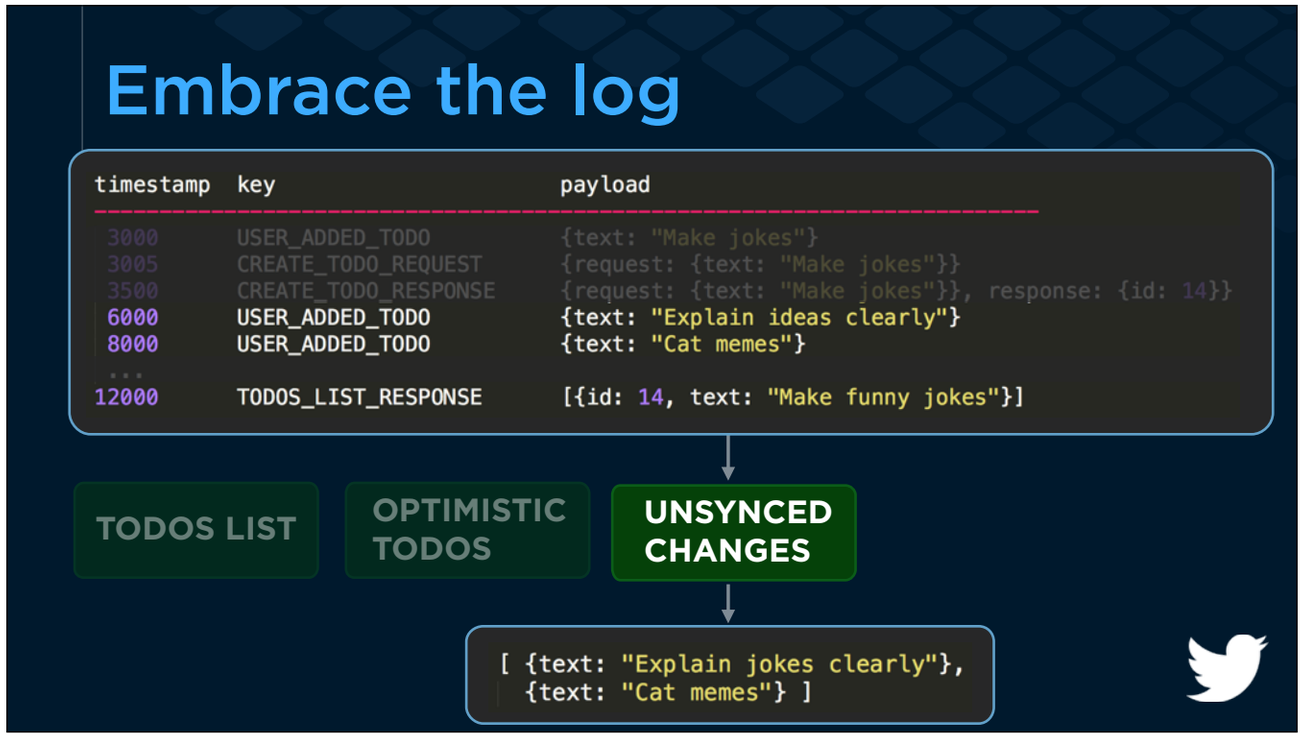


To do this, we reduce through the log, pulling out both the local user actions and server responses, and merge them both together.

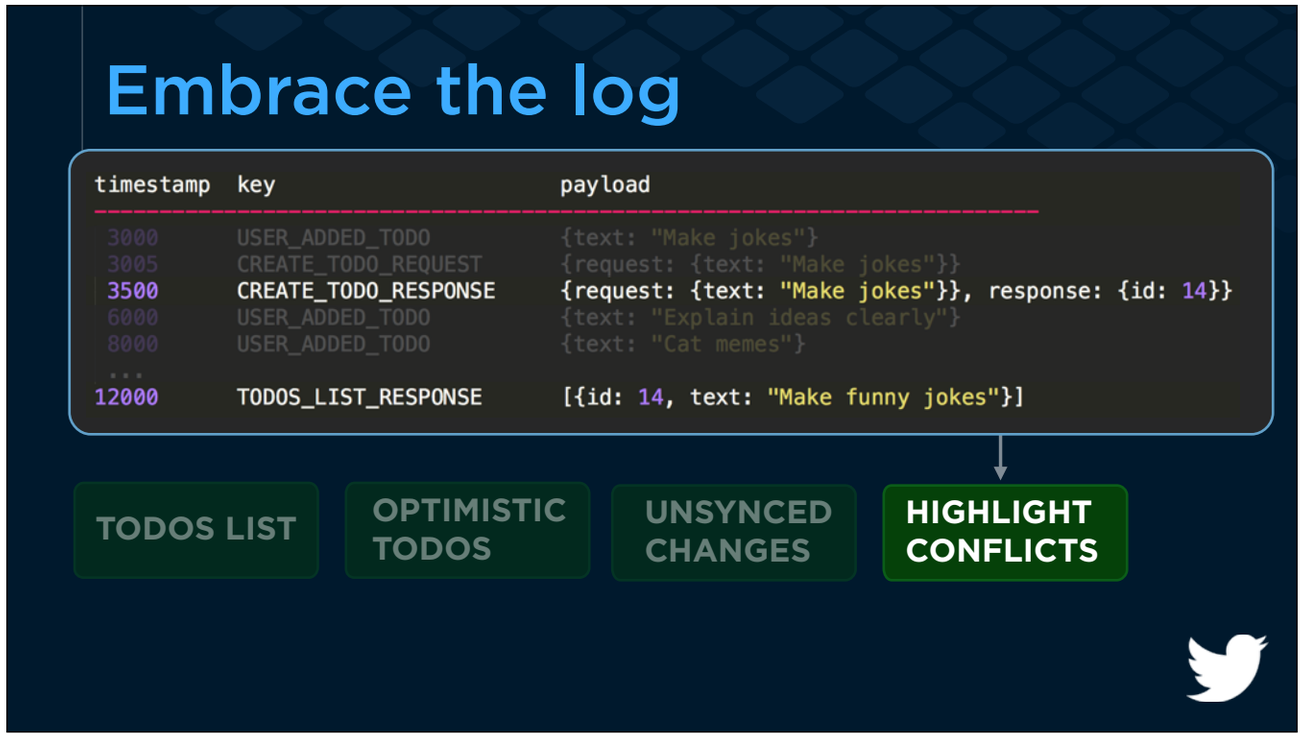
This isn't a super easy function to write, but the point is it's just a function and this is just plain programming.



For apps that work offline, we might want to let users see which changes they've made locally, but haven't been broadcast out to the world yet.

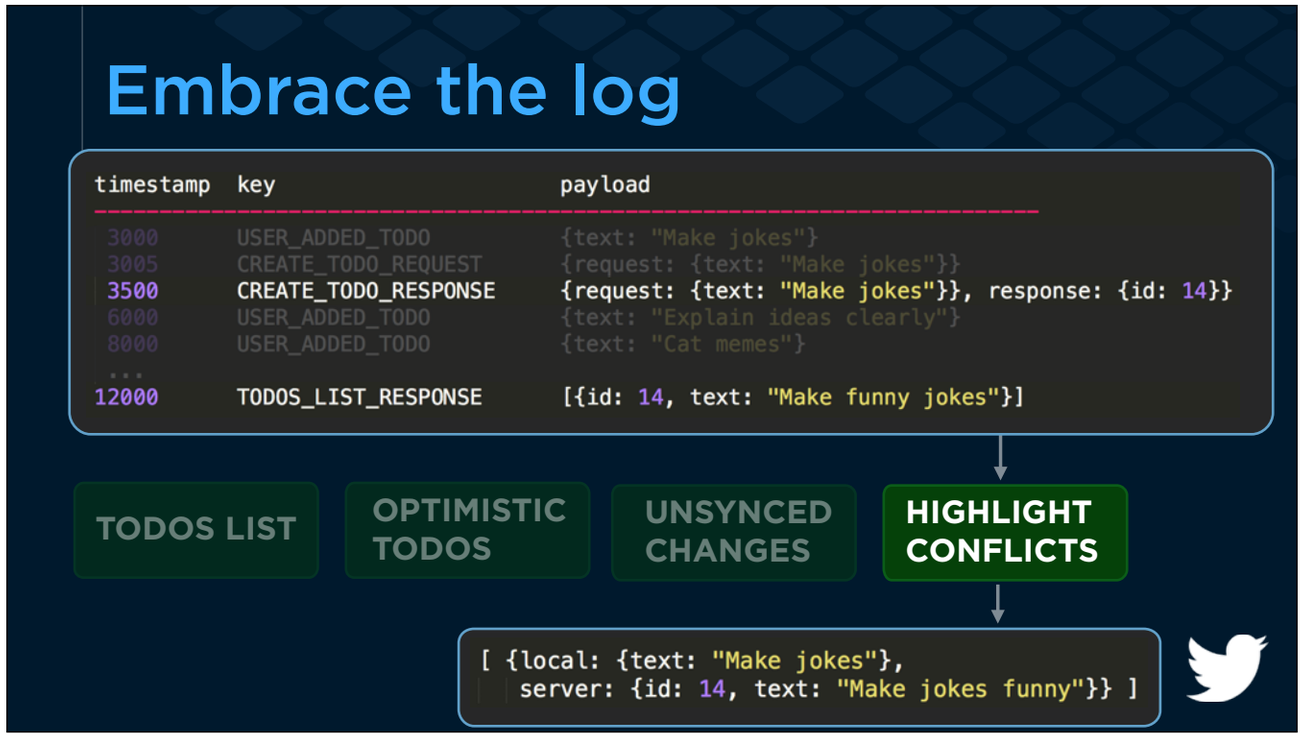


Here we just reduce through and collect local changes, and then filter out ones that have already been acknowledged by the server.



We can even provide a way for users to see where there have been conflicts between local actions and other users' actions from the server. This doesn't even require special cooperation from the server, even if we use simple polling of a REST API, we can see there's a conflict.

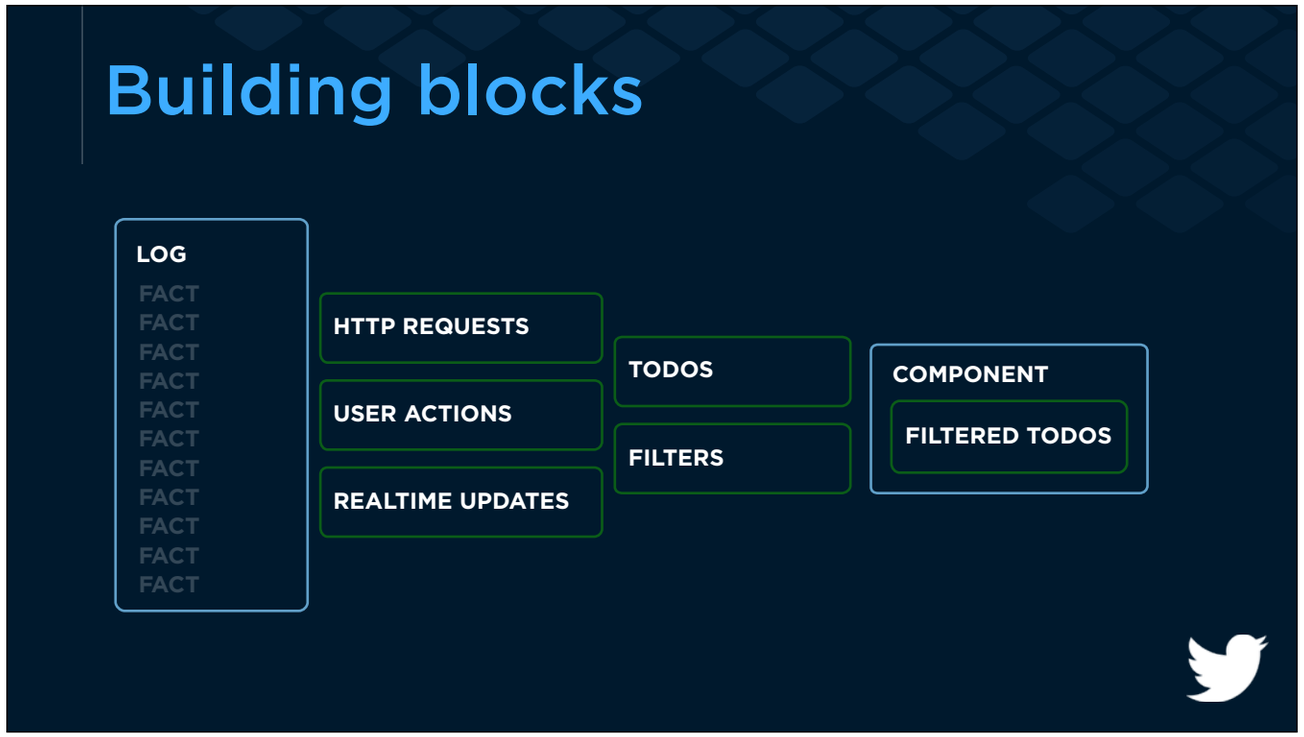
And since we have kept metadata about local timestamps, we have enough information to do more detailed computations. We can pull out the Todos that the user edited in the last ten minutes, and highlight ones where a server update came in that conflicted with the user's change.



All of this is possible because we kept the log and full history.

And importantly, this log if from the **user's** perspective. There's an authoritative server truth, but we have enough information in the UI to reconcile changes there with the user's **local perspective** of what has happened.

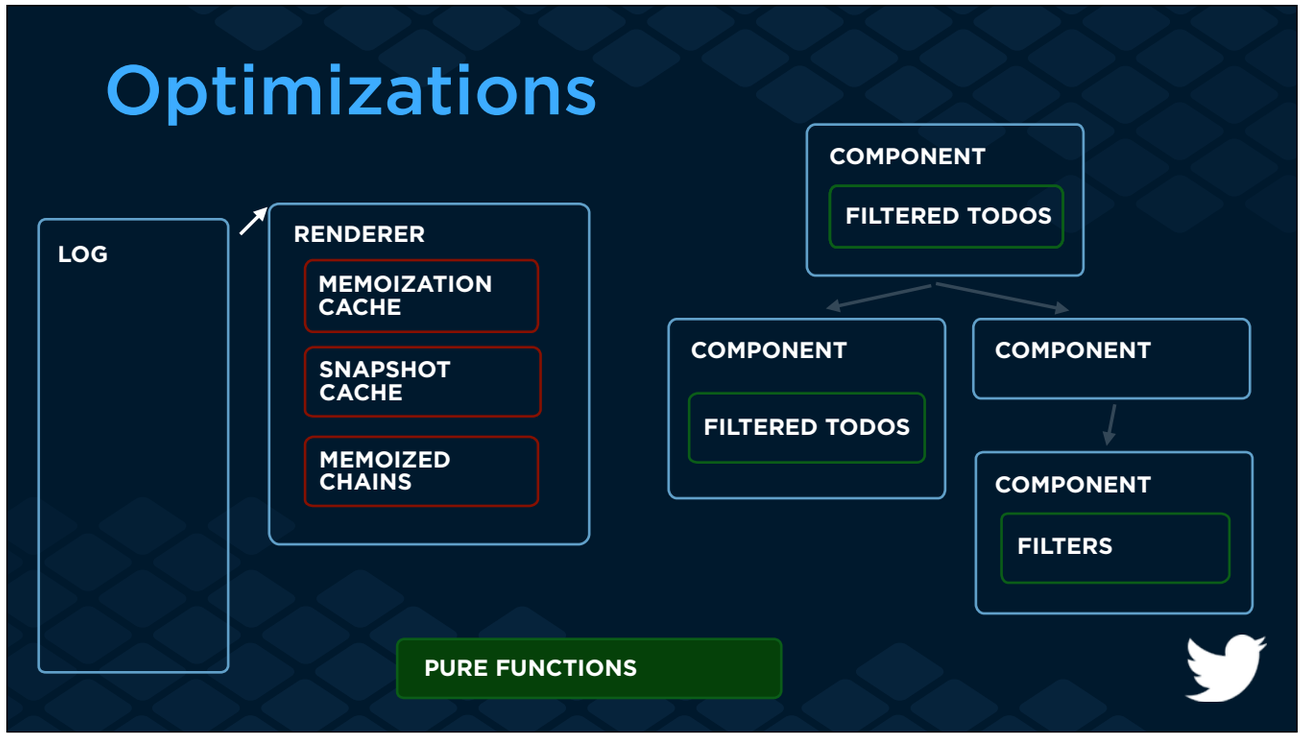
So that’s a sense of this idea. A few more slides, first to acknowledge the challenges here, and then to point you towards some experimentation.



It's one thing to say"well, we can compute anything we want from the log." But it's another to build these abstractions and make them into a developer-friendly API.

This might start with a layer of lower-level computation, but then build up to computing entities, or data for a particular component.

There's work to do here, but this is plain function composition, the easiest kind of programming to refactor, test, re-use, and shape to exactly what you need.



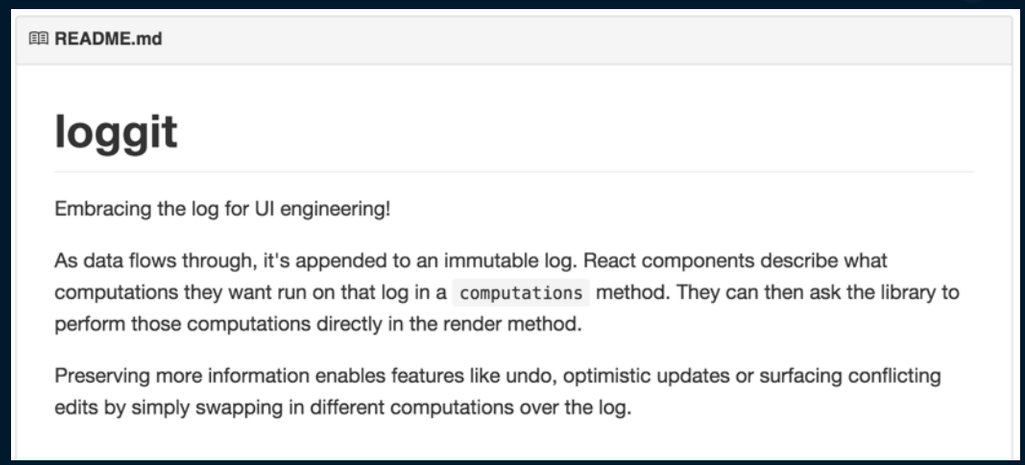
There also might be some performance challenges with a naive solution that keeps a log without bounding its size, or repeatedly reforms a full reduce over the log.

Fortunately, data storage systems in other spaces have tackled these problems, and there's a lot we can learn from there.

See https://twitter.com/krob/status/617868529082085376 for more information.

Code!

https://github.com/kevinrobinson/loggit/



I've put together some code demonstrating these idea, building off Dan's awesome Redux project.

It's on GitHub, and implements some of the optimization strategies mentioned on the previous slide.

https://github.com/kevinrobinson/loggit/

If you're curious, there's also some more explanation here: https://github.com/kevinrobinson/loggit/.

And my full talk from React Europe is here: https://www.youtube.com/watch?v=EOz4D_714R8

Thanks!
Kevin
@krob