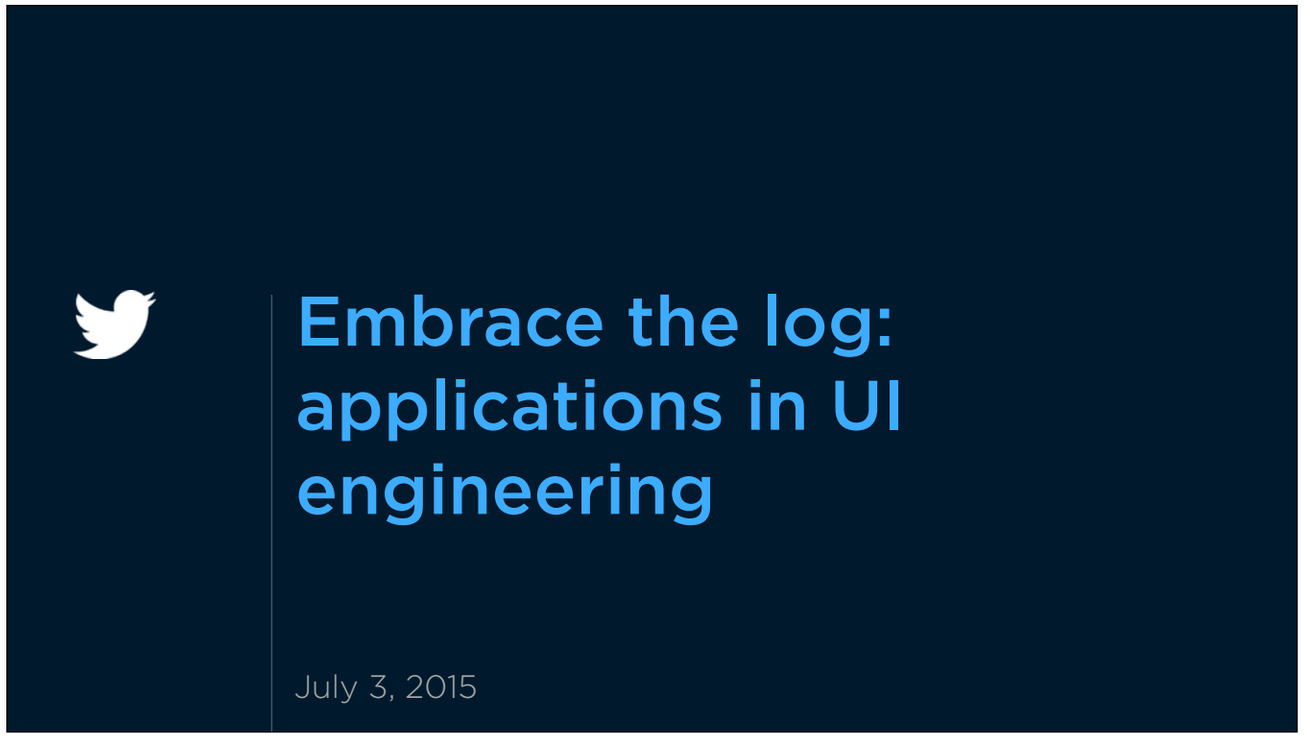
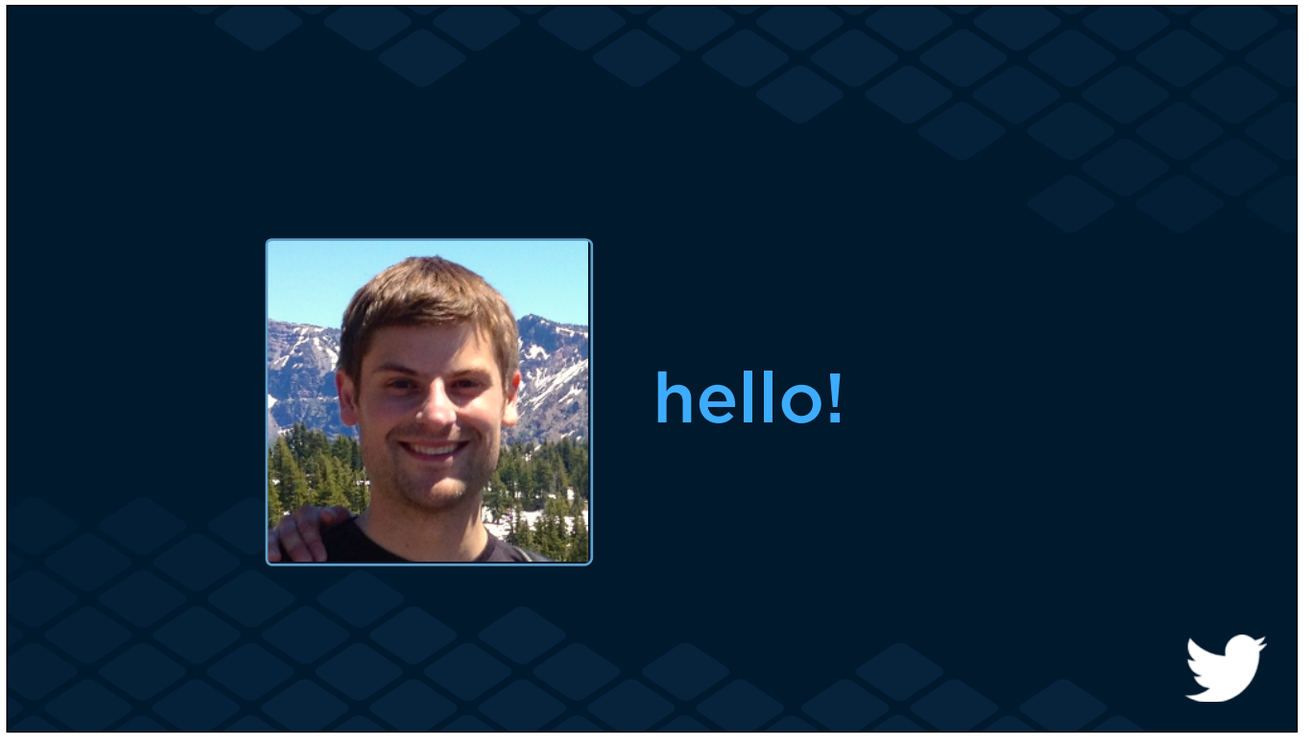
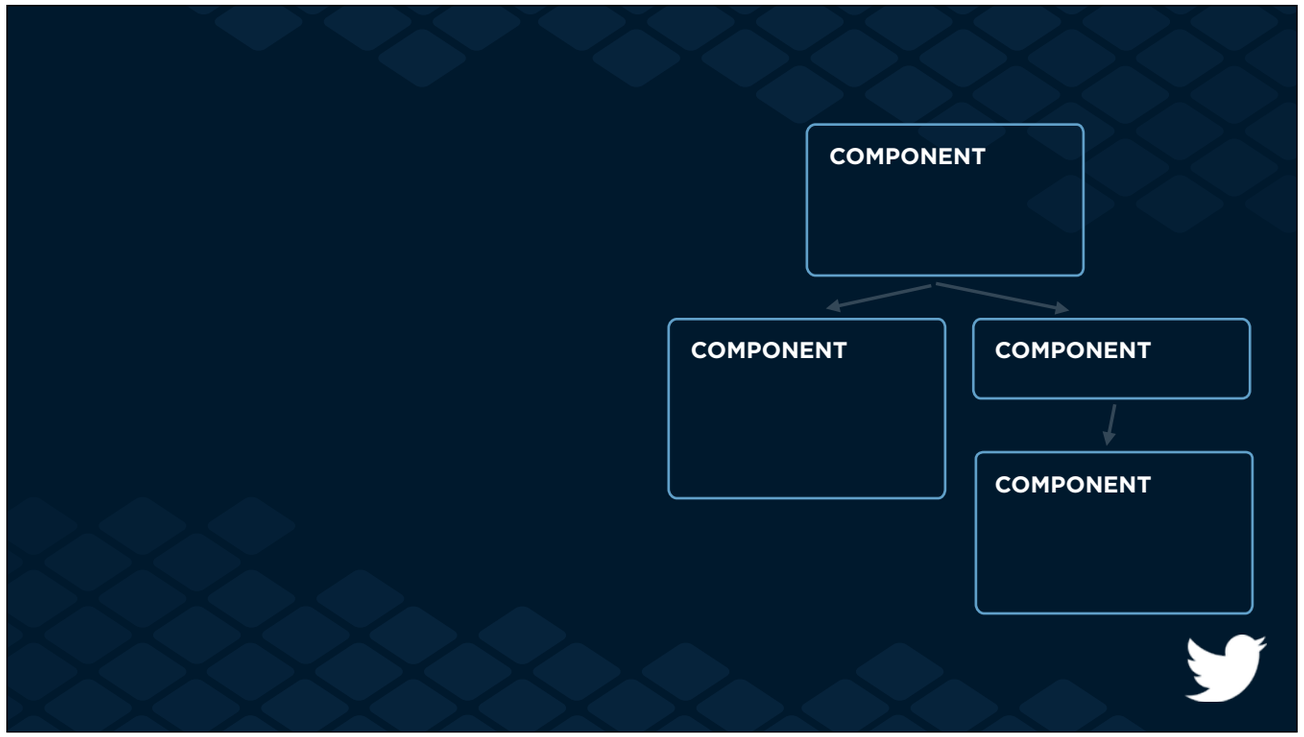



This is drawn from a part of my talk at React Europe on July 3rd, 2015.  It focuses on one part of the talk and is intended to be read.

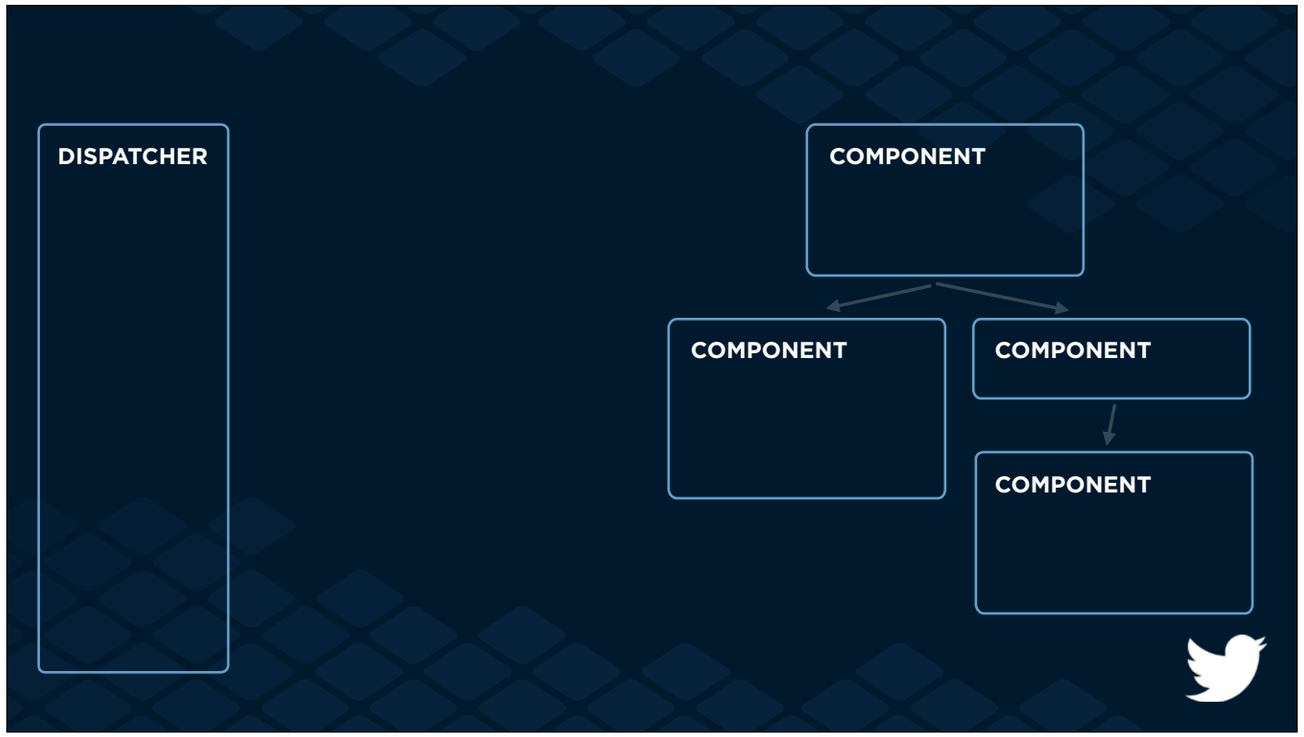



Hi!  My name's Kevin (@krob), and I'm going to share some thoughts about using a log as the central concept for organizing data and computation in a UI application.  This is more of a sketch than a proper design doc or article, but I'm pretty convinced these are ideas worth exploring.  I'd love input.

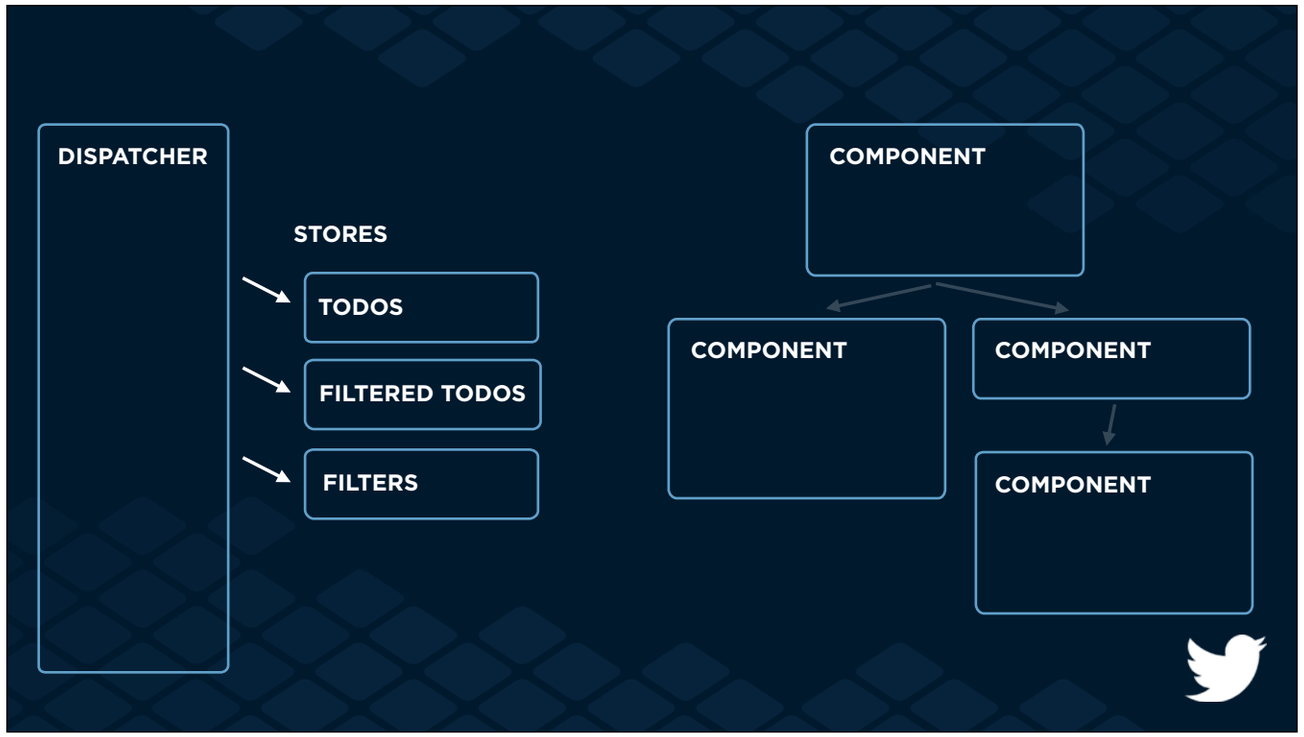
There is related code and explanation here: https://github.com/kevinrobinson/redux/pull/1



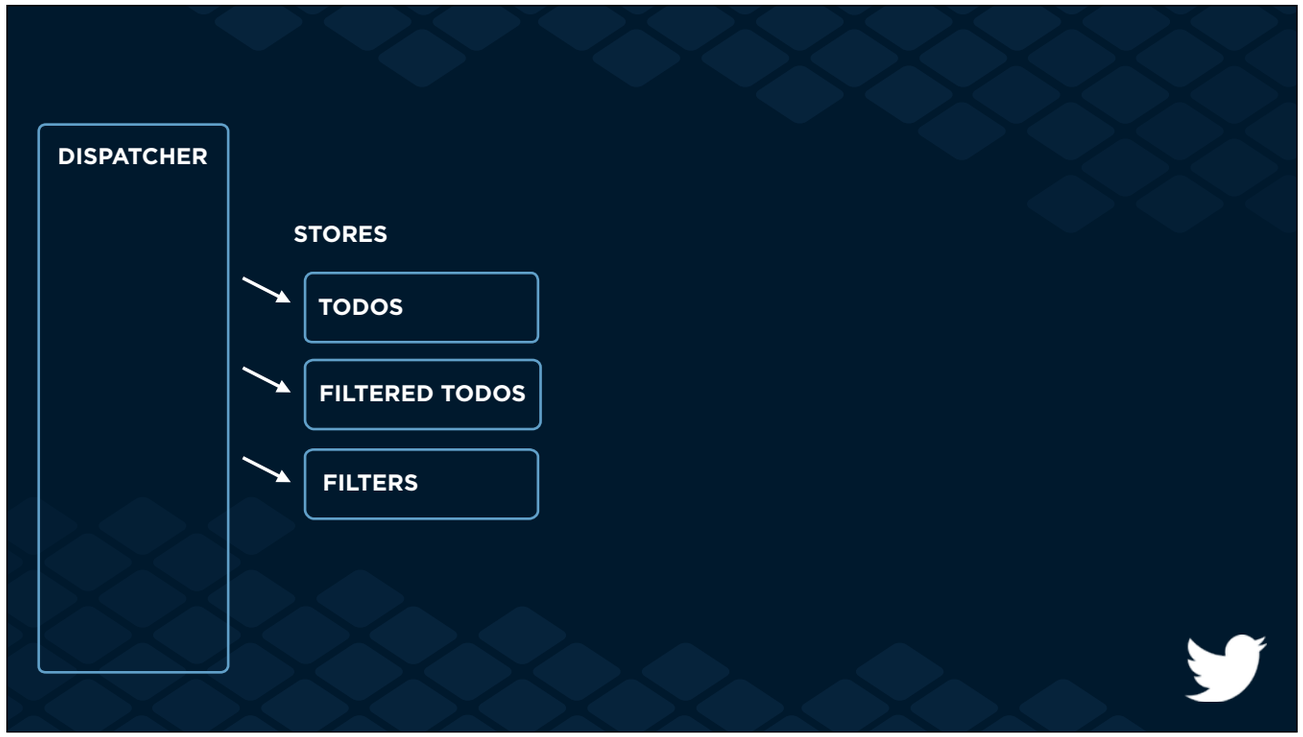

We start with a React component tree.



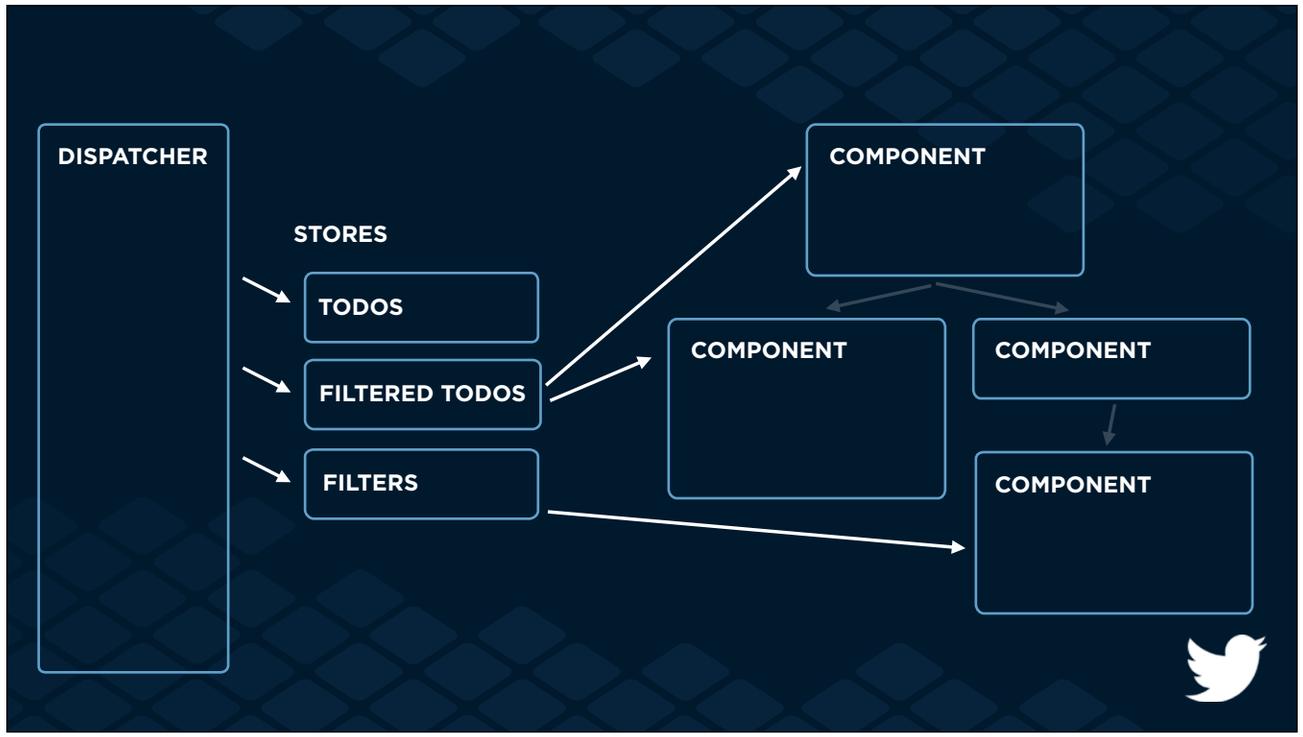

Then we add a Dispatcher, that actions flow through.  Like Flux.




Stores listen for actions flowing through the Dispatcher.  Their sequence could be chained.

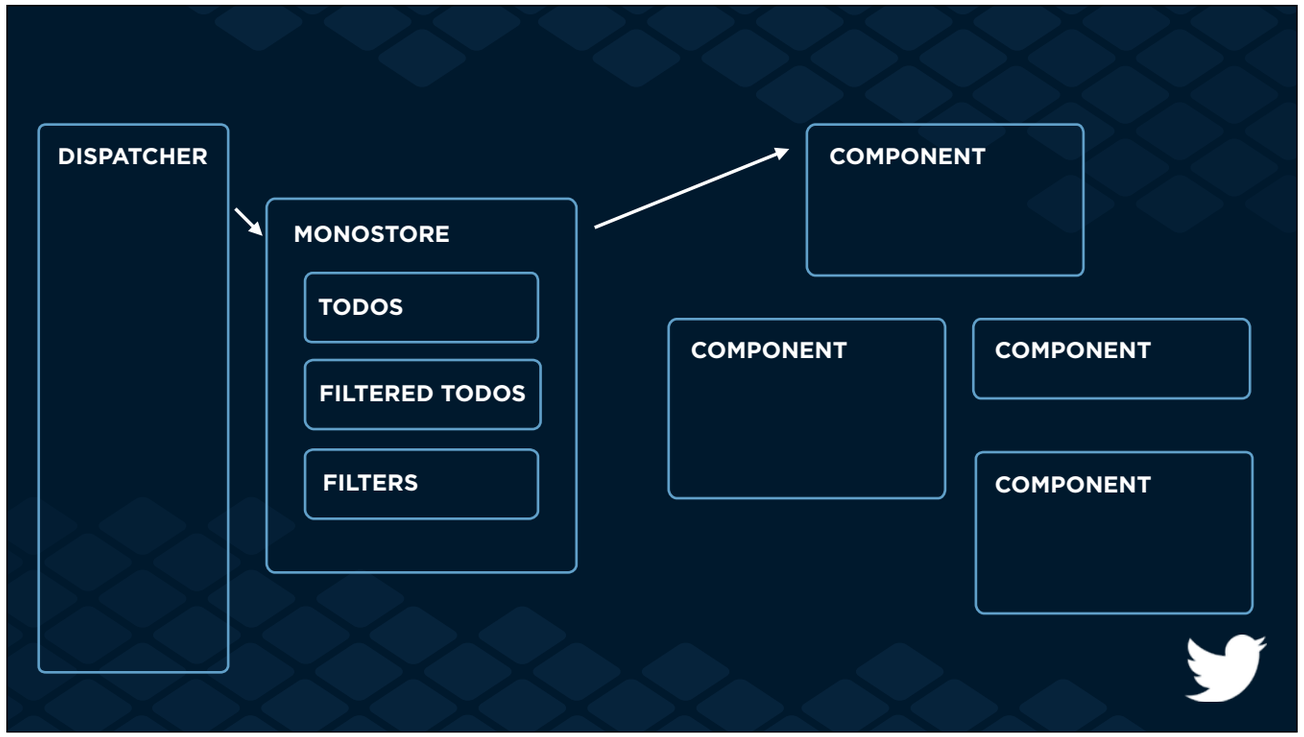



Note that these Stores will have to know about the component tree to some extent, since they need to compute everything the component tree needs.

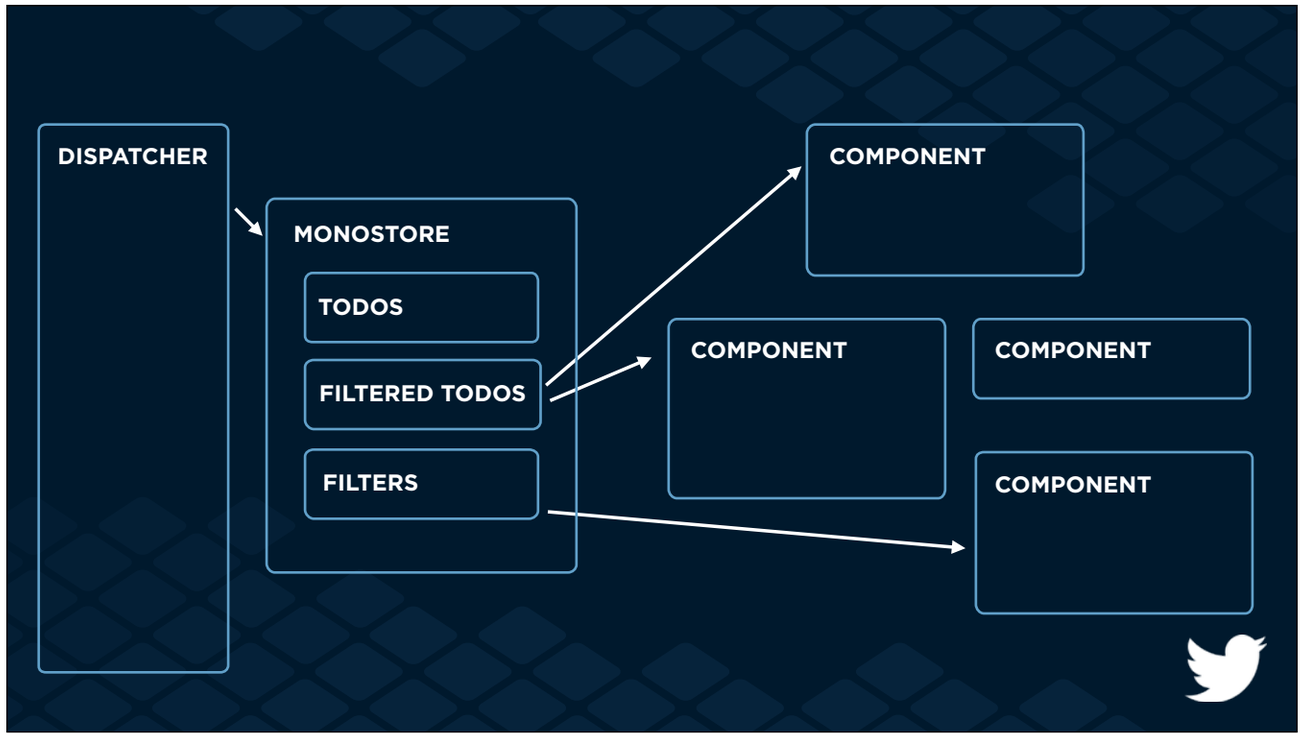



Components listen for updates to Stores.  On updates, they pull the values from Stores into their own state and re-render.

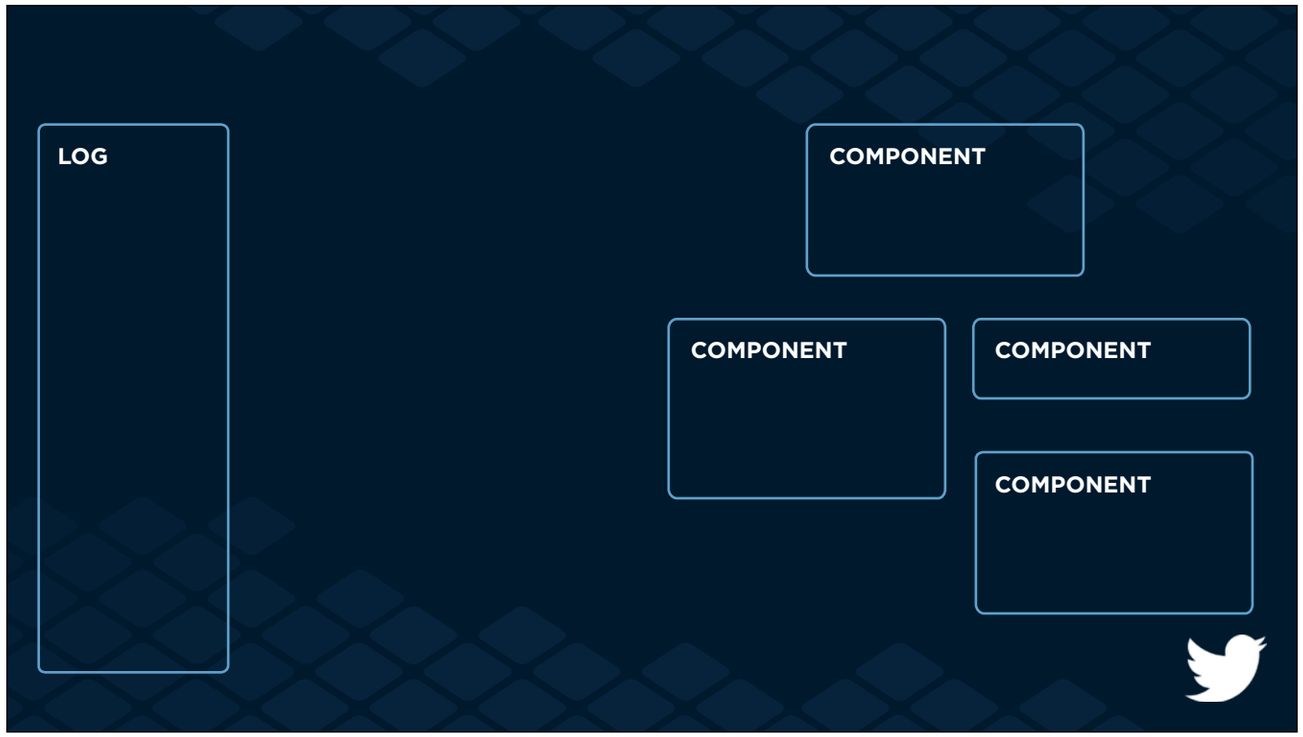
But sequencing computations across Stores can get complicated, and changes in multiple Stores might trigger multiple updates to the same component.



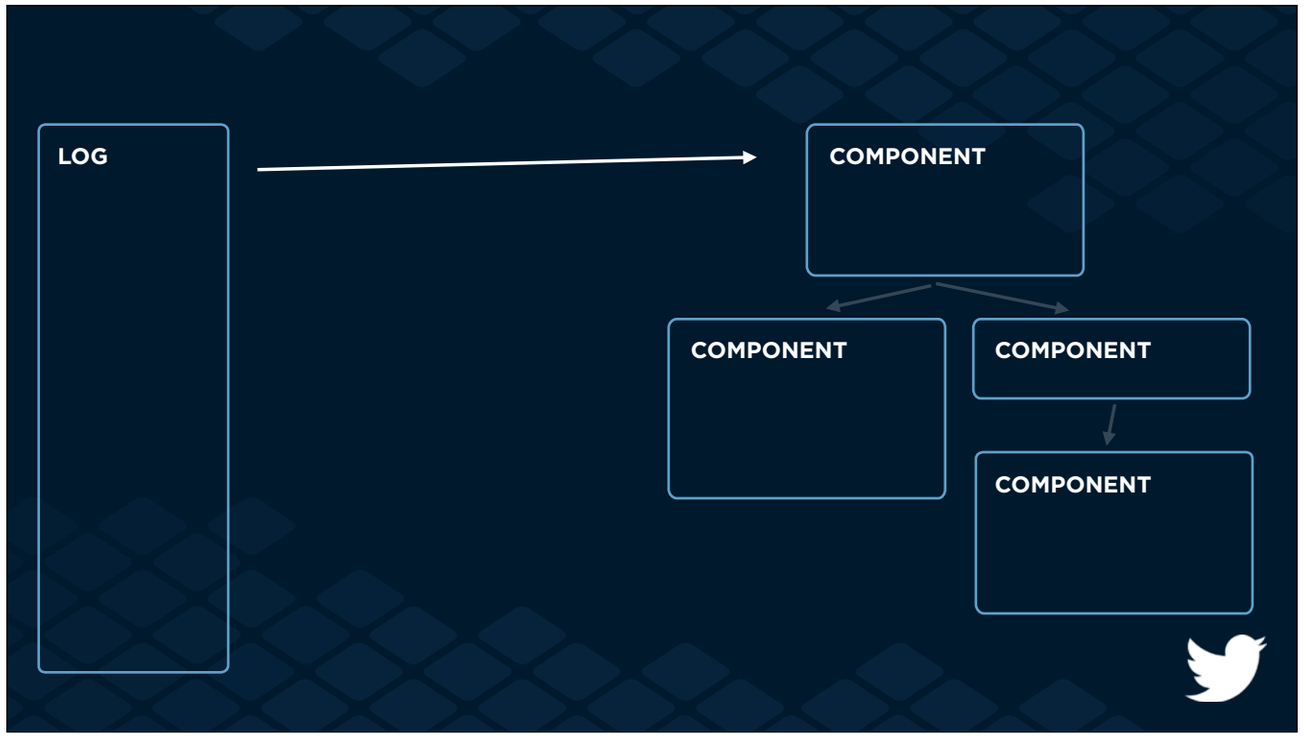

One solution is to make a Monostore that wraps individual stores.  This might be a single structure, like the state atom in Om, or it might compose other Stores, like in Redux.



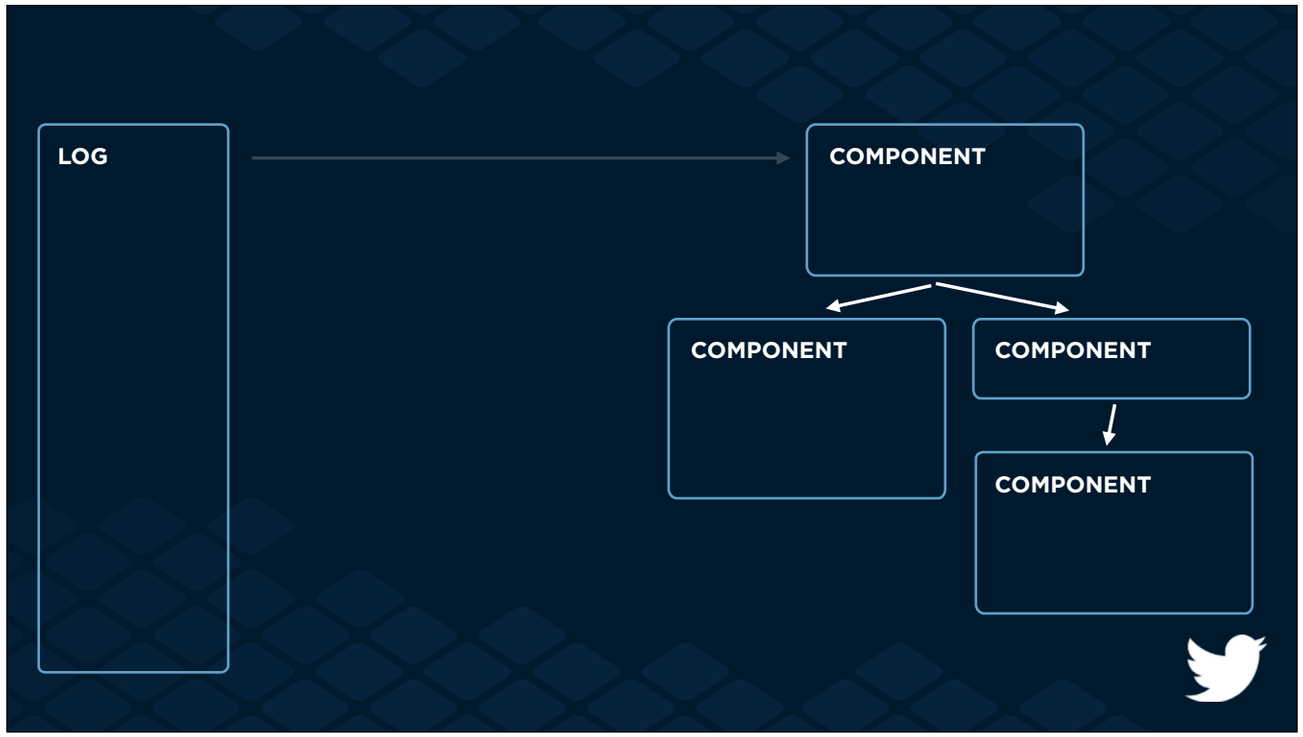

If we want to only present a slice of the Monostore to components, we could use lenses or cursors.  This also lets the Monostore emit change events at a finer granularity, to optimize rendering instead of doing a full re-render and diff over the entire component tree.



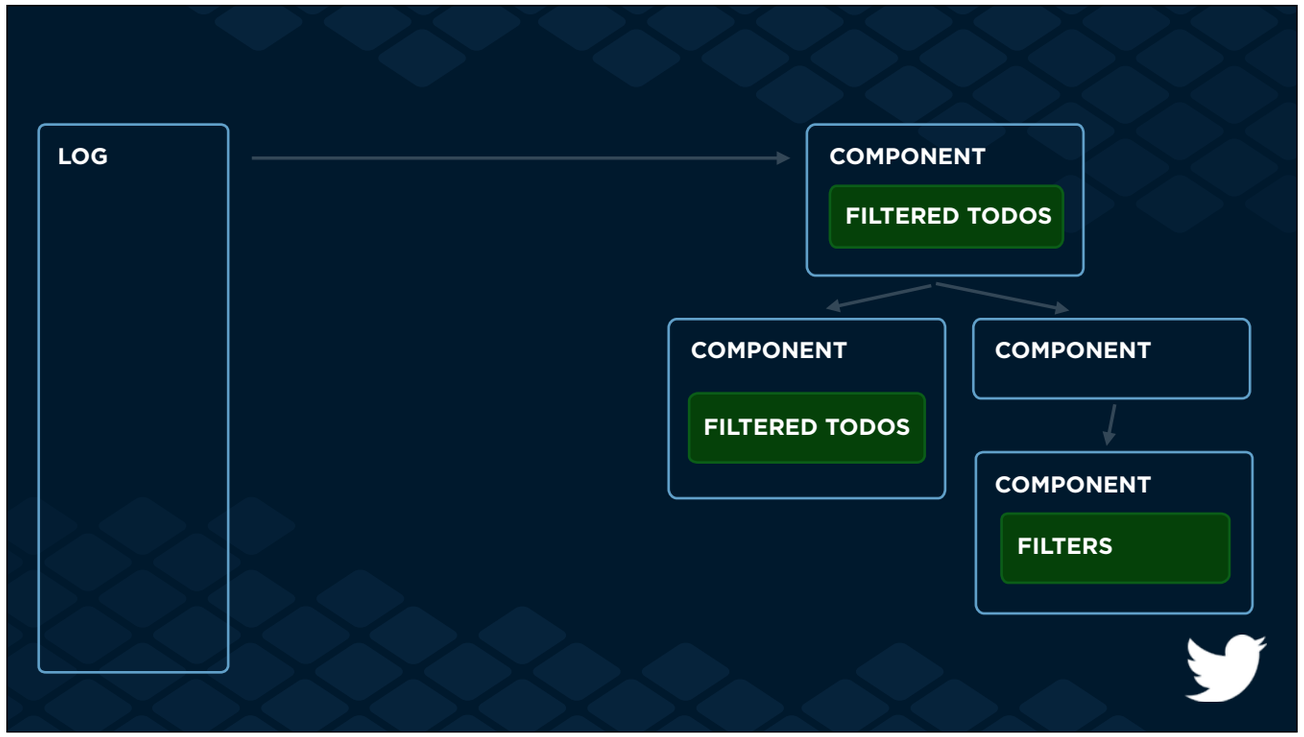

Let’s look at what this might look like using a log as the central organizing concept.



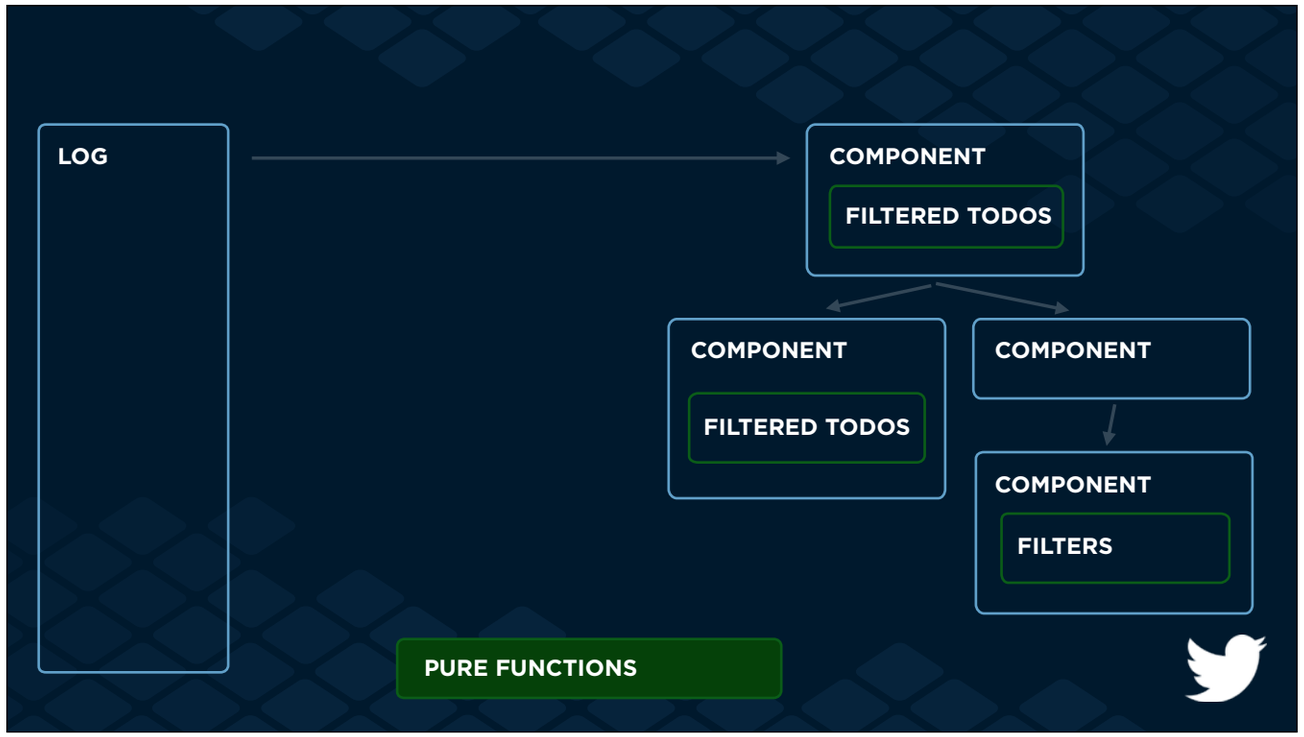

Let’s start simple and naive.  When the log is written to, we can call setState and pass the list of facts to the root component.



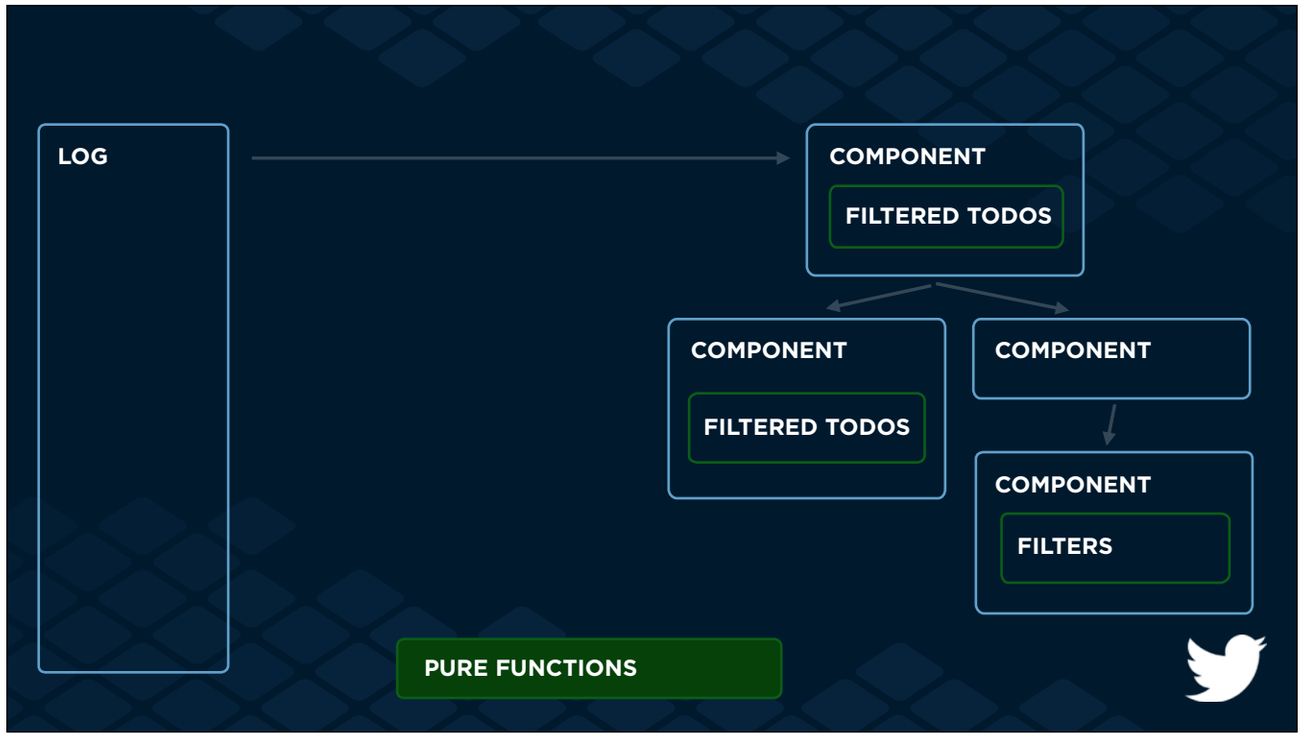

This list of facts is a prop (or context) that each component threads throughout the component tree.



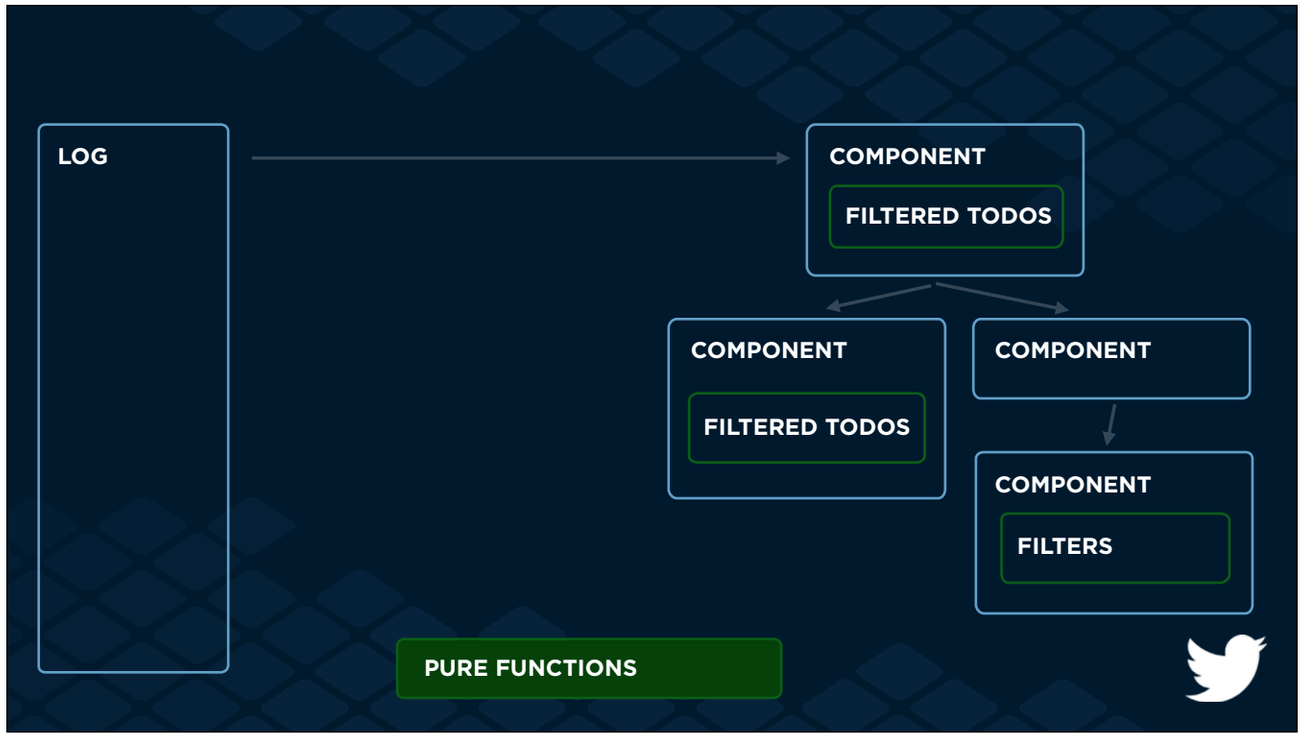

Each component then has a reference to all facts in the log.  And they naively apply functions to compute what they need from the log.  It's the same as if we were using reducer functions as Stores, applied when each event comes in.



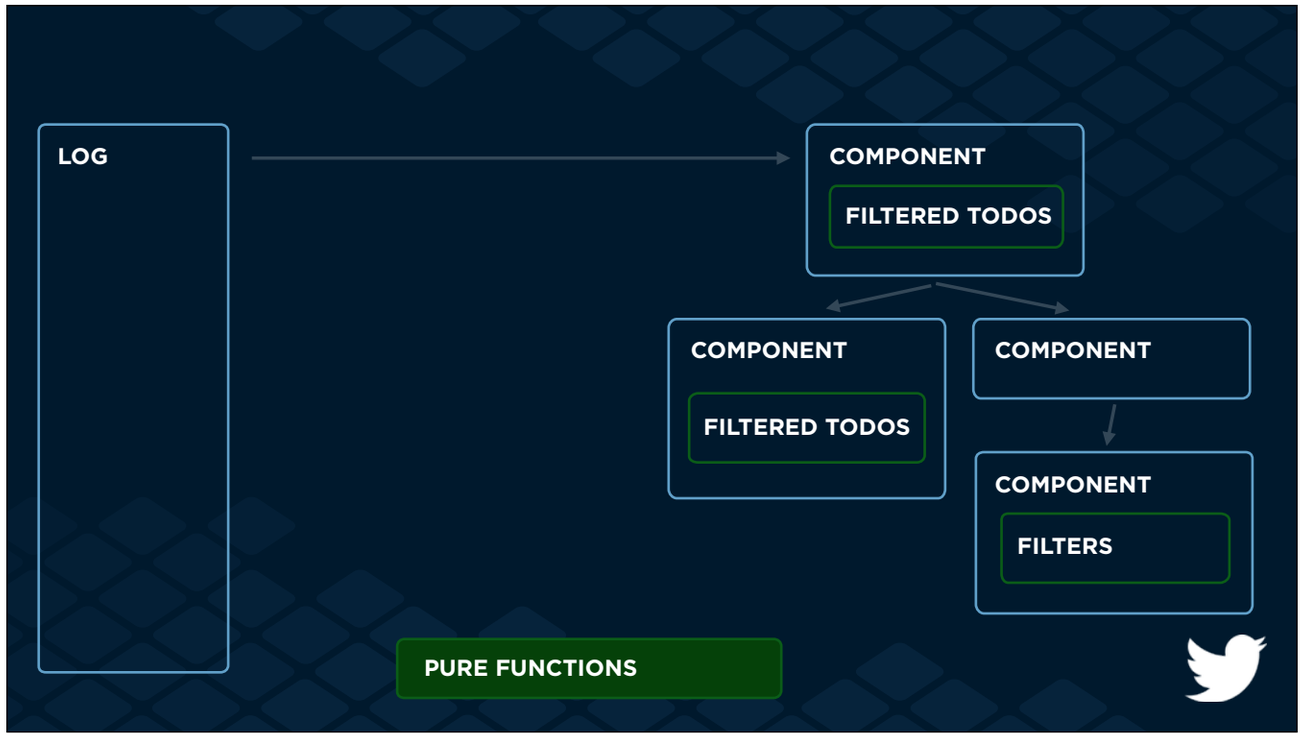

If computations are repeated, like computing the current list Todos, or the current state of the filters, the mechanics of how to perform these computations can be extracted out.  But what each component needs is still expressed directly in the component, where its purpose can best be understood.



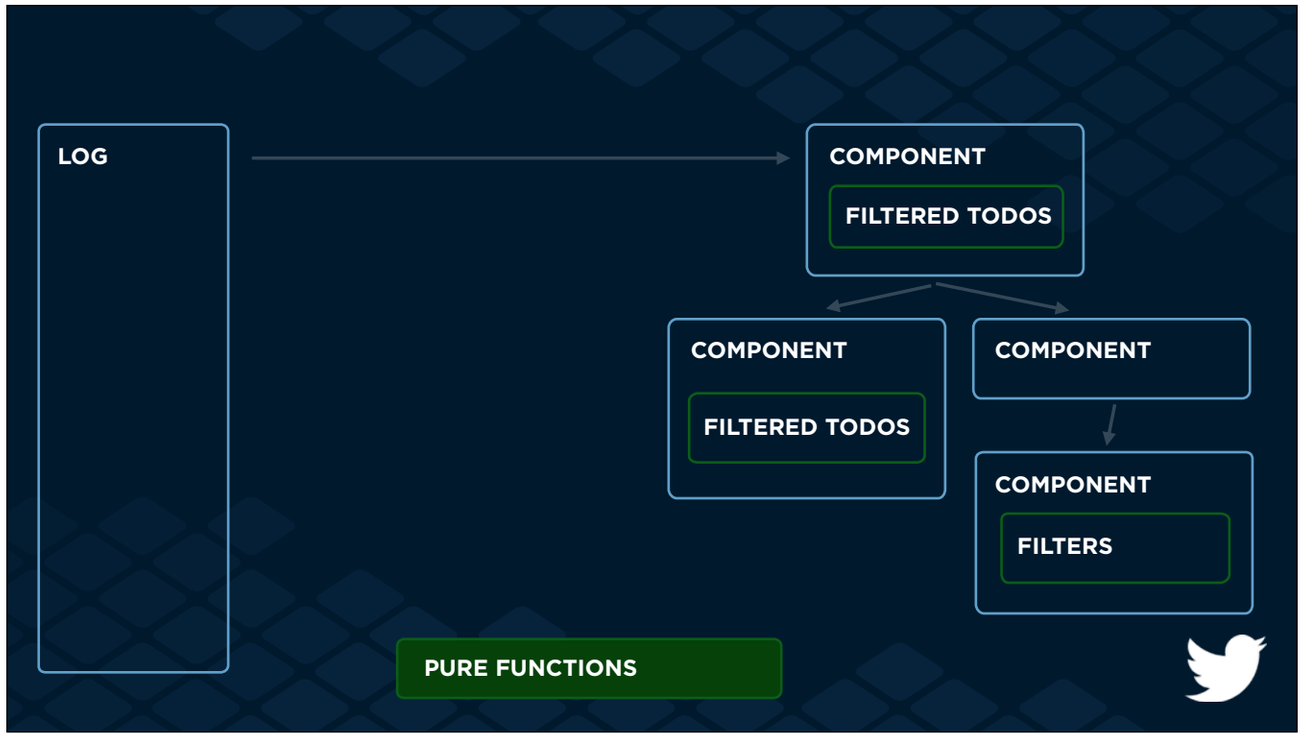

These pure functions are easy to reason about and factor however we like.  They might take a list of facts and reduce them, or they might take two data structures that were computed from the log and merge them together.



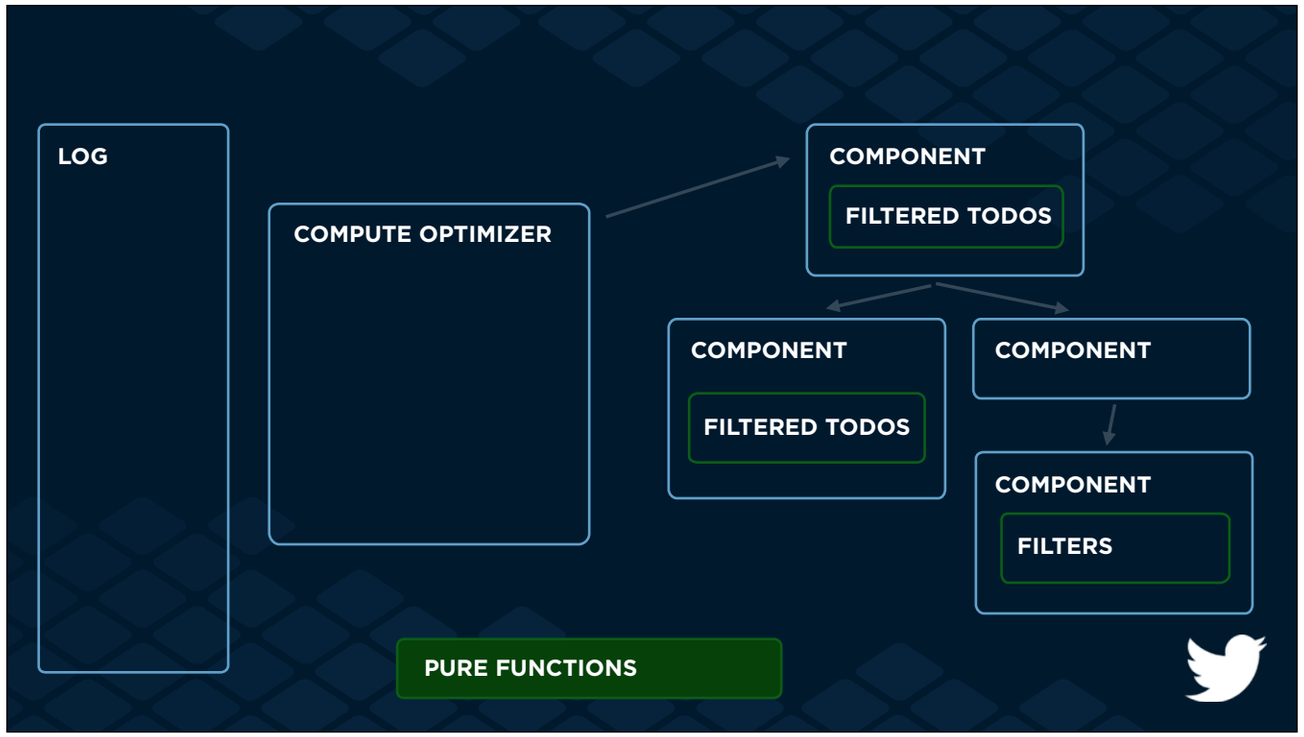

I’d like to suggest that at this point, we already have an interesting approach that provides an improvement in terms of simplicity and communicating intent by co-locating computation with components.  Here are a few other strengths.



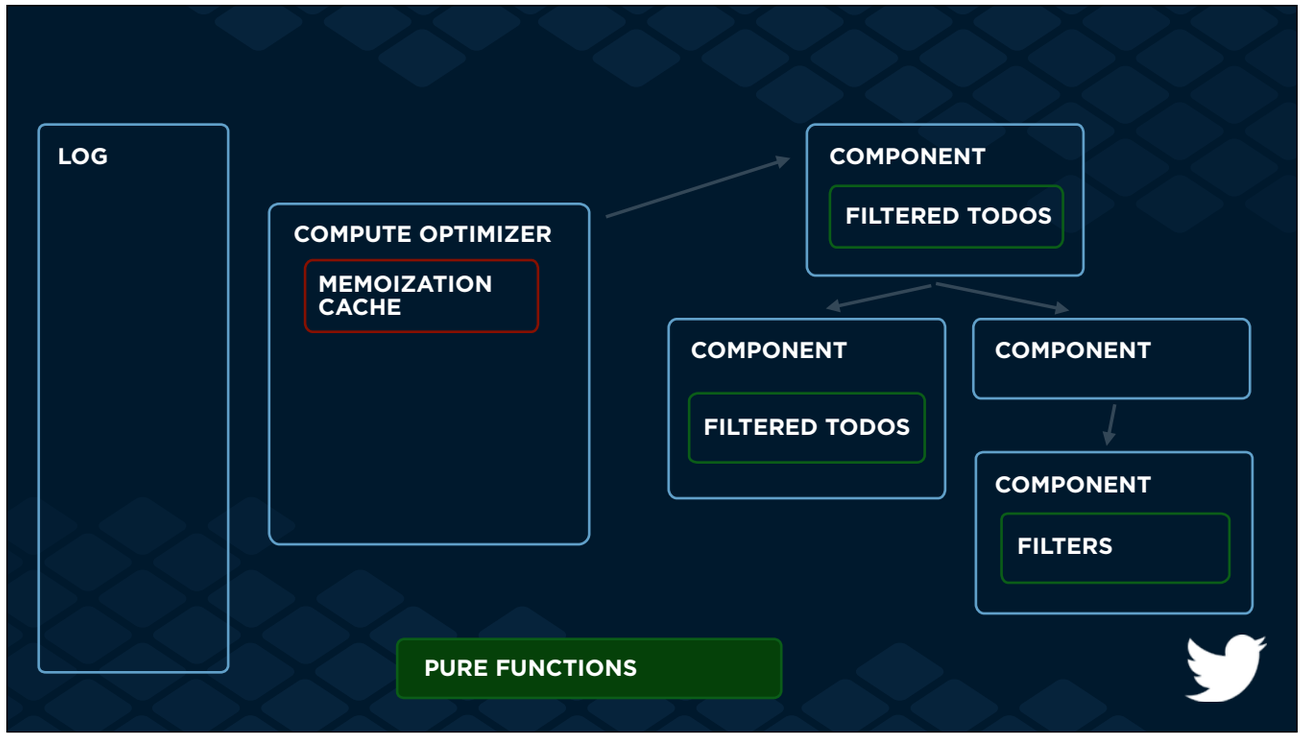

One, by describing the computation we need inside components, when the component tree changes, React will naturally swap in an entirely different set of computations on the log.  This is akin to swapping different Stores in and out in standard Flux.



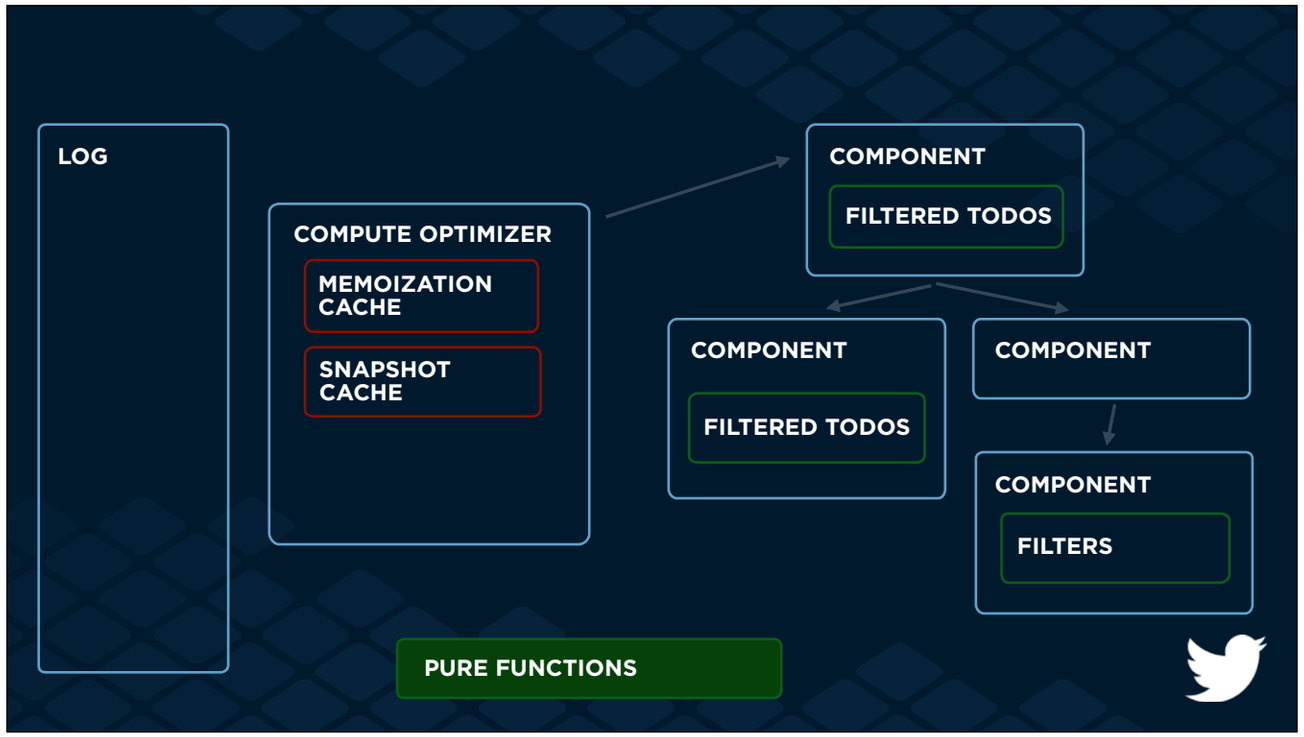

Two, by preserving more information in the log, and deferring performing computations on it until we need to, this design has set us up well for more complex features like collaborative editing and partial connectivity.  These kinds of features don't fit well with mutating Stores in-place.



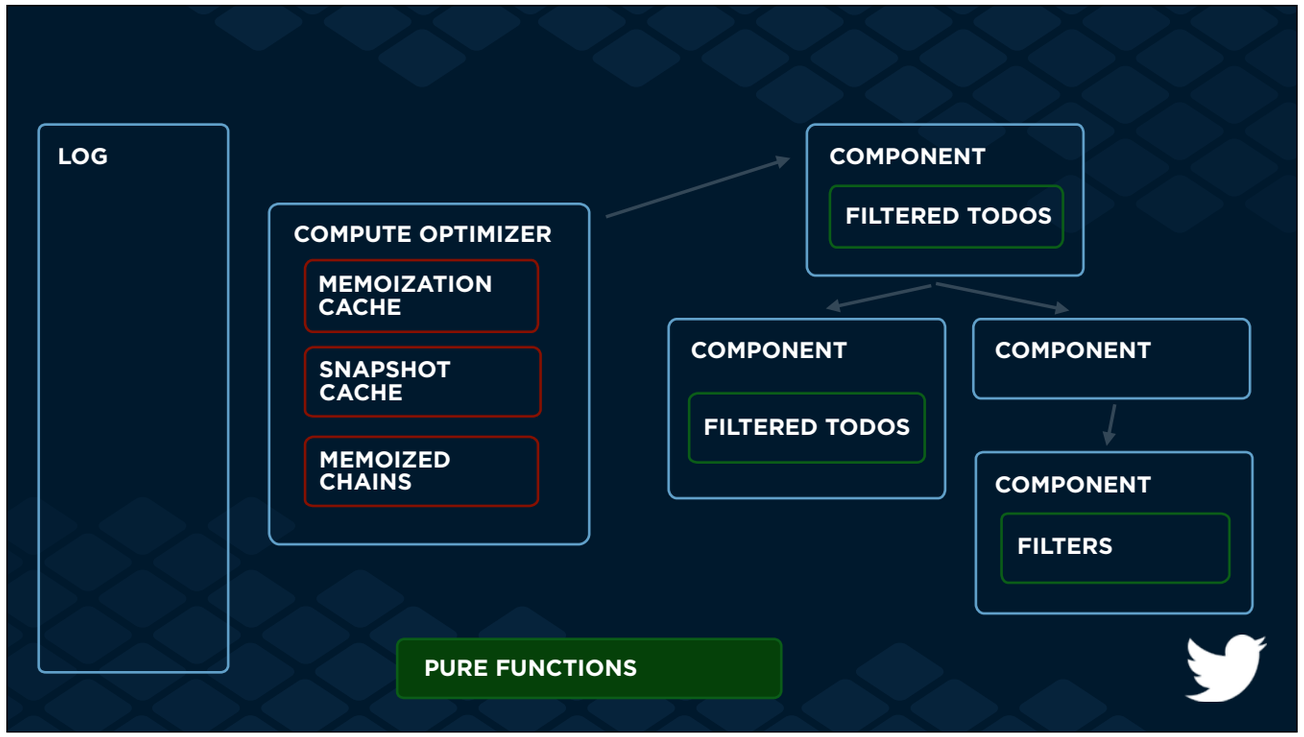

But the naive solution might not have great performance.  That's an important concern.  Profiling is critical here, and needs to ground any optimization discussion, but I'd like to offer a few thoughts about why, even if performance does need to be optimized, it's not a fundamental obstacle.



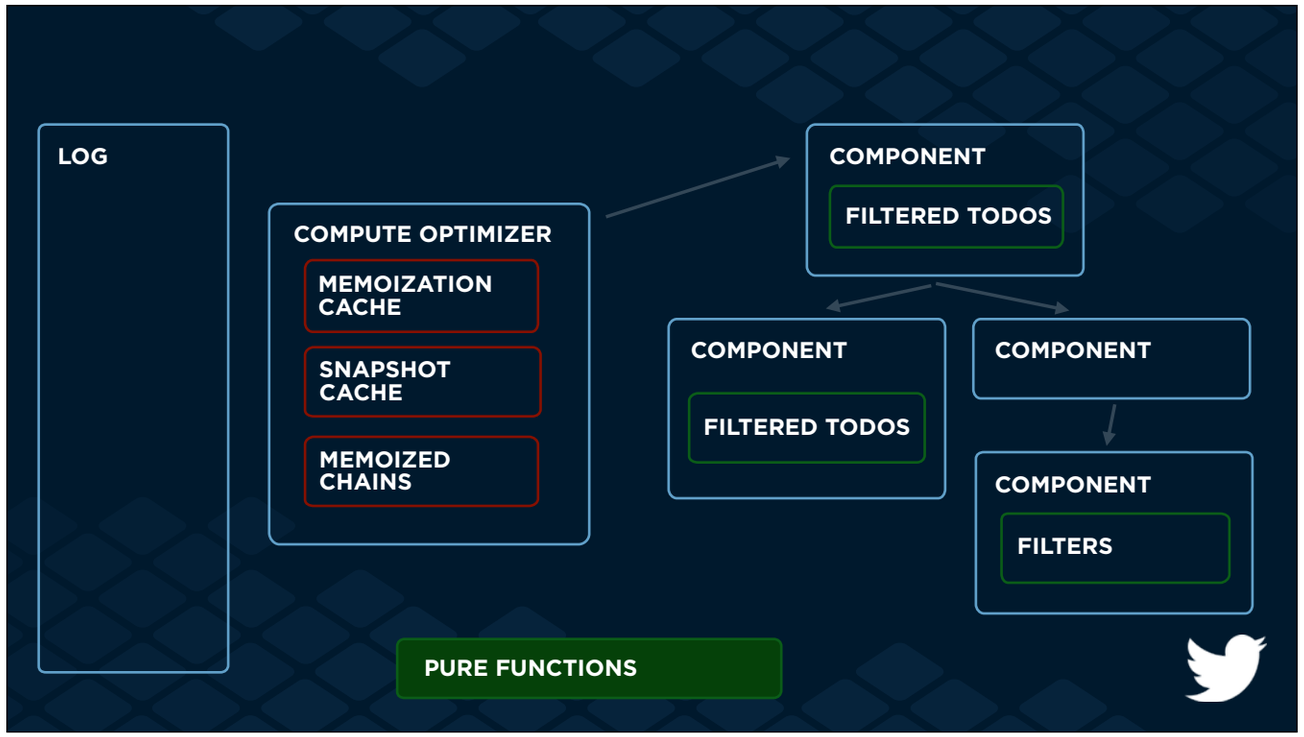

How might a first cut at an optimizer work?  A first step might be memoizing calls to apply a reduce over the log.  An optimizer can take a list of facts and a reducer, and keep a bounded cache of the values for hot calls.



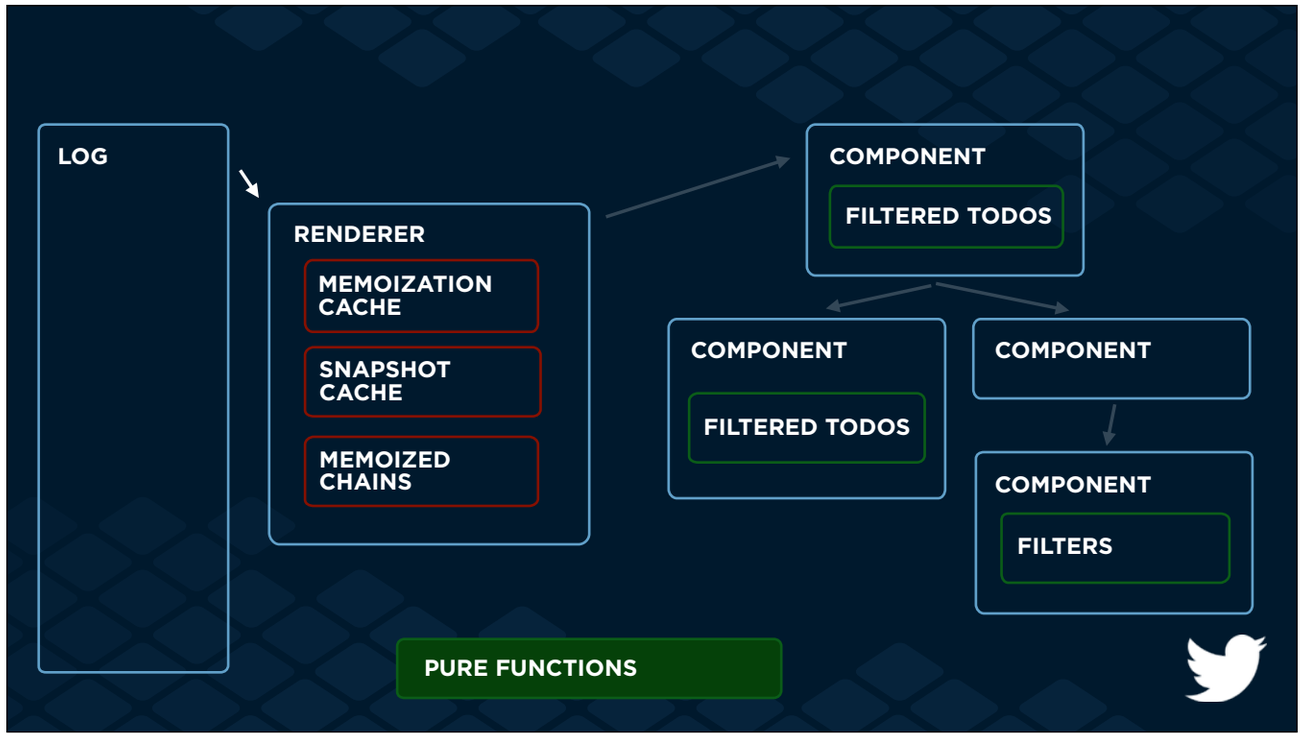

Next, we might want to keep “snapshots” of computation we’ve performed over a previous list of facts.  Keeping a bounded cache of computed values for hot computations can speed up computing them as new facts come in.



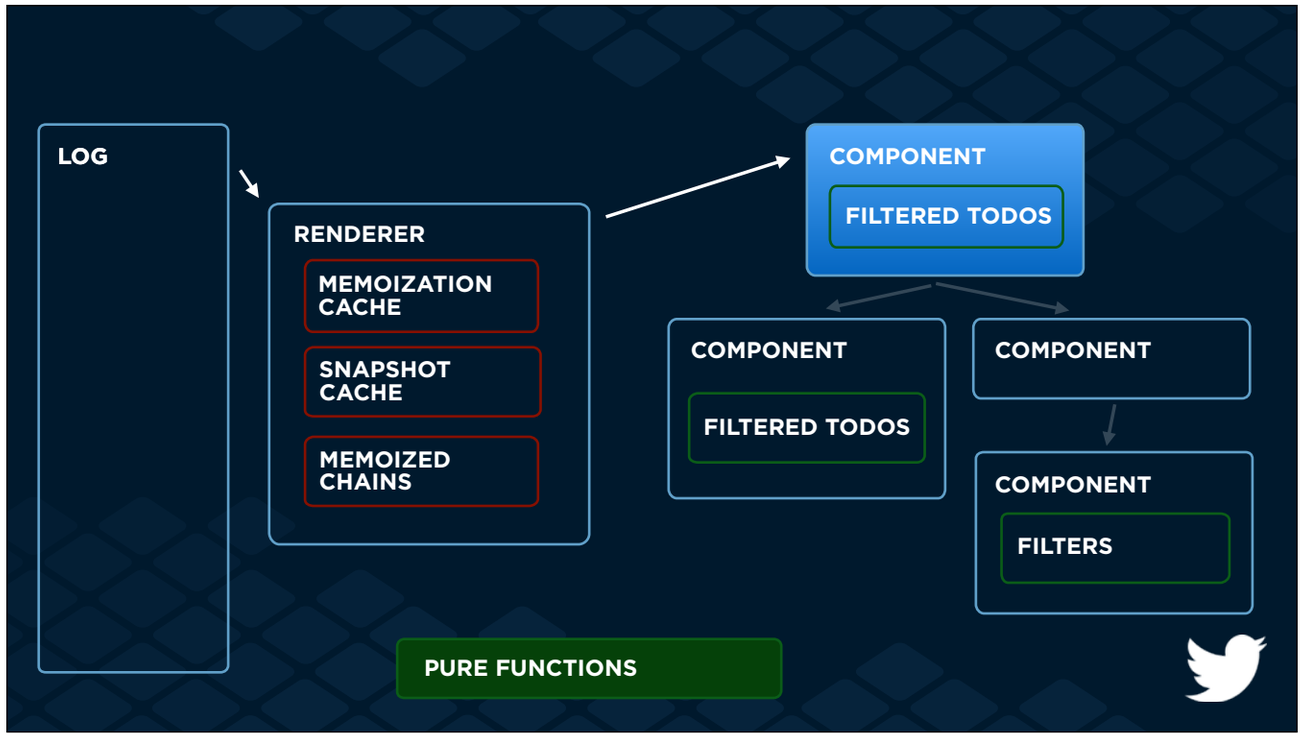

After that, we might want to support memoizing across chains of computation.  We'd need to define and limit the semantics of how this works, and this is a place where API experimentation will be really interesting, and where making it feel like plain programming is important.  One example is nuclear-js, which feels like a natural solution for folks coming from Clojure.



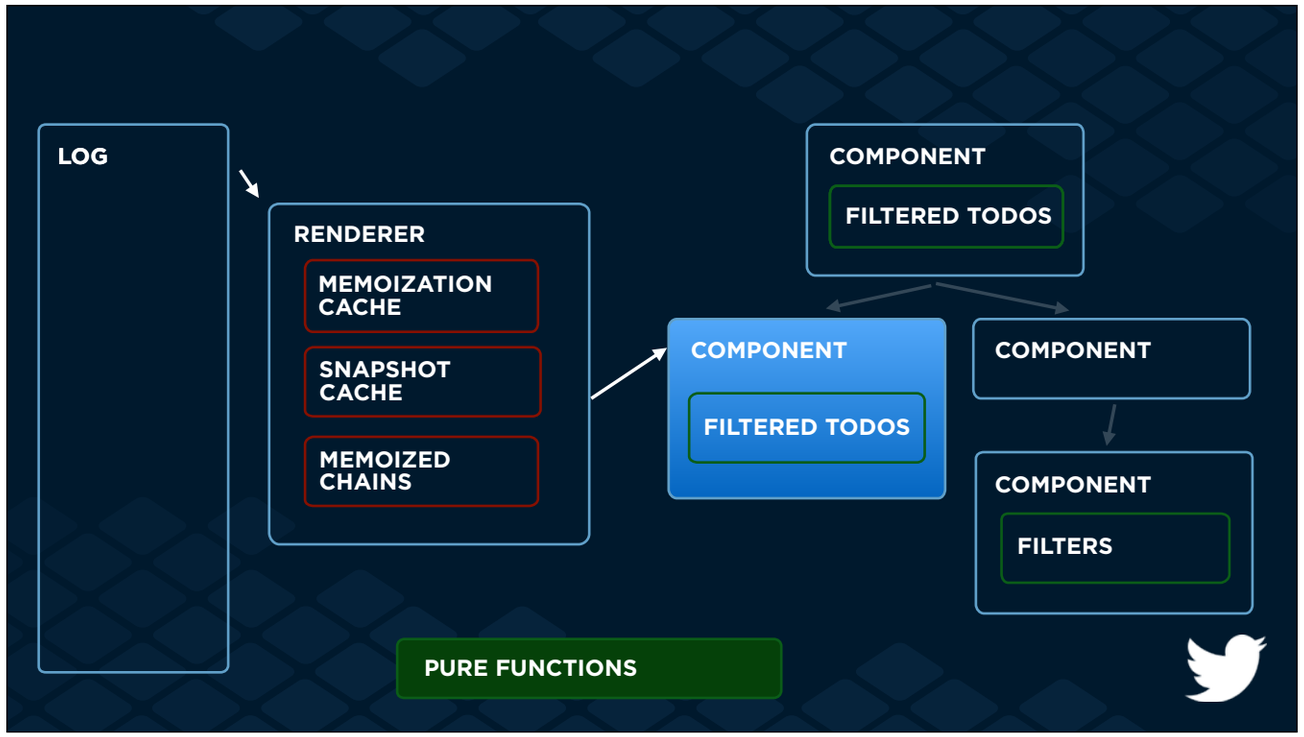

Okay, so these are some ways we can optimize computation, let's talk about how we might approach optimizing rendering.  This might diverge a little bit from the standard React approach, but to me it's a reflection of a core strength of React's declarative component system that this is even possible.



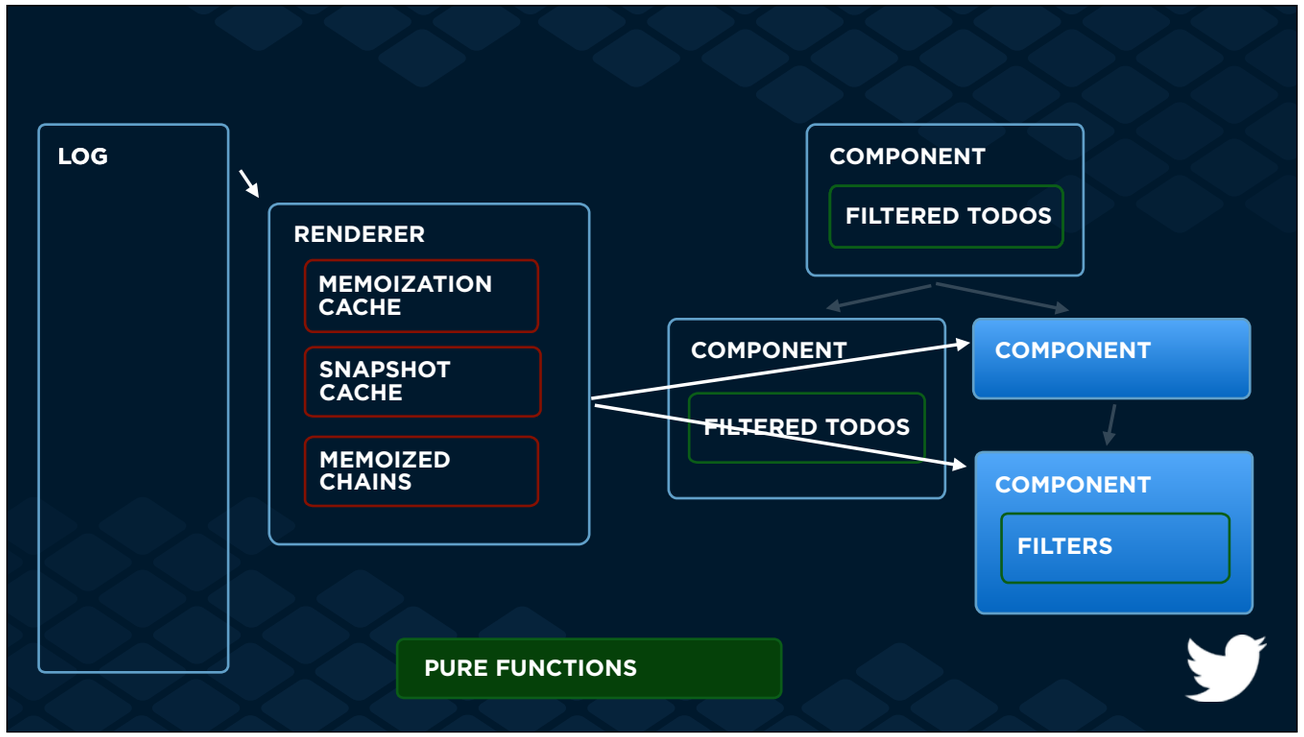

Instead of thinking of the optimizer as just optimizing compute, it can also optimize rendering.  After a new fact is recorded, we know we may need to update some parts of the component tree.



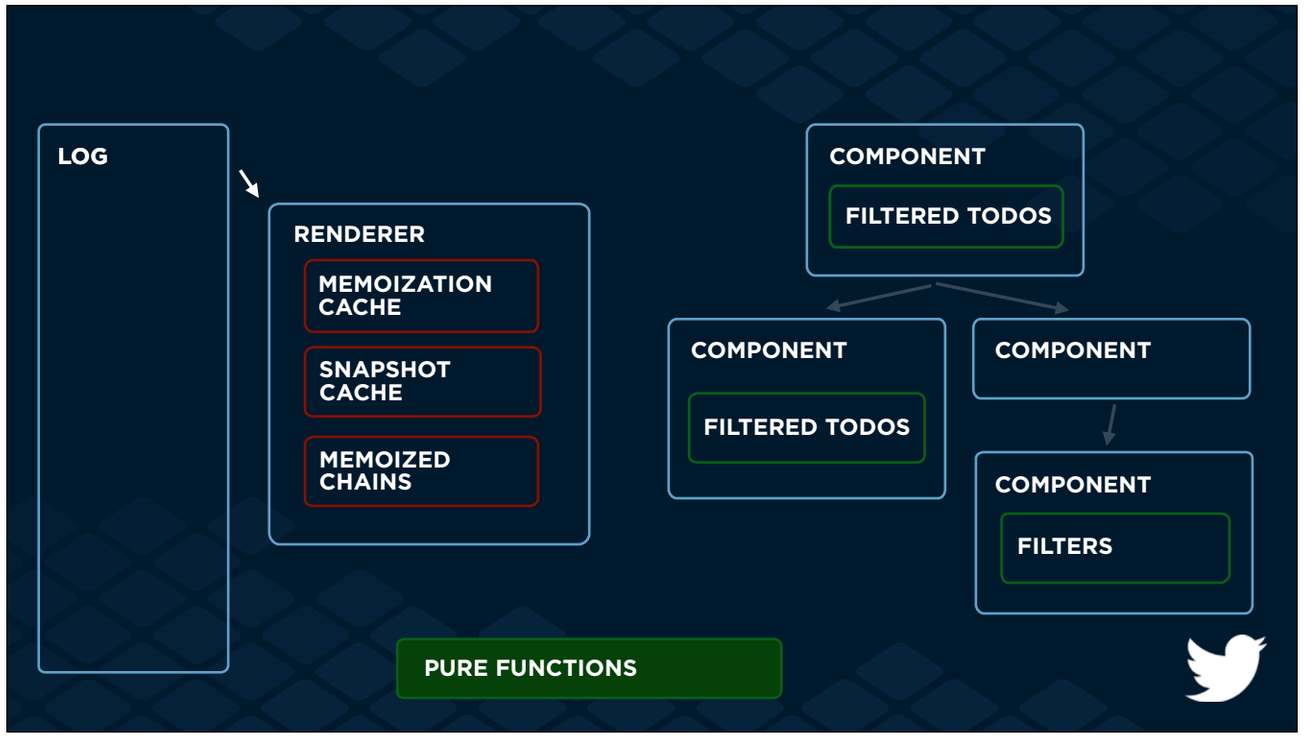

Starting at the top, we can compute what this component needs and diff.  If there's no changes, we don't need to re-render it.




Next we look at its children.  We perform the left component's computation, diff to see if it is different, and if it is, we re-render that component.




On the right side, there's no computation on the middle component so we can skip right over it.  Underneath it we run that component's computation, diff and discover we need to re-render.  When we re-render, this is the same performance as calling setState.  The real difference is we essentially add a short-circuit like 'shouldComponentUpdate' to the renderer, if the computation for that component didn't change at all.




So that's a sense of this idea.  I'm sure there are lots of problems to solve here, and also that other people much smarter than me have probably tackled most of them elsewhere. :)  I'd love to hear any thoughts folks have on this, or other work that's related that they could point me to where I can learn more.


If you're curious, there is related code and explanation here: https://github.com/kevinrobinson/redux/pull/1.  I'd love your help!

Thanks!
Kevin
@krob