9/6/2022, written by *Marvin Hagemeister* and *Jason Miller*

Introducing Signals

Signals are a way of expressing state that ensure apps stay fast regardless of how complex they get. Signals are based on reactive principles and provide excellent developer ergonomics, with a unique implementation optimized for Virtual DOM.

At its core, a signal is an object with a `.value` property that holds some value. Accessing a signal's value property from within a component automatically updates that component when the value of that signal changes.

In addition to being straightforward and easy to write, this also ensures state updates stay fast regardless of how many components your app has. Signals are fast by default, automatically optimizing updates behind the scenes for you.

Run in REPL

```
import { signal, computed } from "@preact/signals";

const count = signal(0);
const double = computed(() => count.value * 2);

function Counter() {
  return (
    <button onClick={() => count.value++}>
      {count} x 2 = {double}
    </button>
  );
}
```

Signals can be used inside or outside of components, unlike hooks. Signals also work great alongside both hooks ***and*** class components, so you can introduce them at your own pace possible. Using signals in Preact adds just **1.6kB** to your bundle size.

If you want to jump right in, head over to our documentation to learn more in depth about signals.

Which problems are solved by signals?

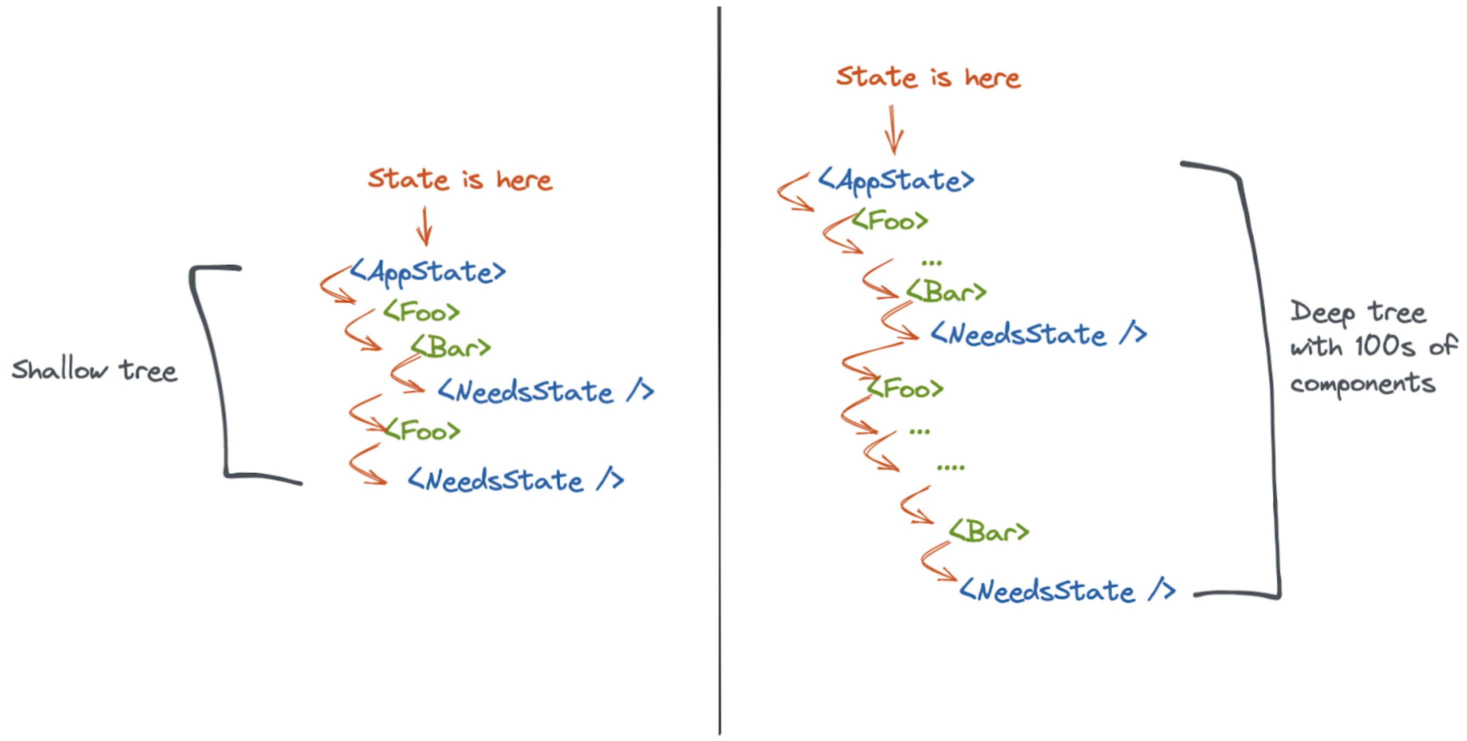
Over the past years we've worked on a wide spectrum of apps and teams, ranging from small startups to monoliths with hundreds of developers committing at the same time. During this time everyone on the core team has noticed recurring problems with the way application state is managed.

Fantastic solutions have been created that work to address these problems, but even the best solutions still require manual integration into the framework. As a result we've seen hesitance from developers in adopting these solutions, instead preferring to build using framework-provided state primitives.

We built Signals to be a compelling solution that combines optimal performance and developer ergonomics with seamless framework integration.

The global state struggle

Application state usually starts out small and simple, perhaps a few simple `useState` hooks. As an app grows and more components need to access the same piece of state, that state is eventually lifted up to a common ancestor component. This pattern repeats multiple times until the majority of state ends up living close to the root of the component tree.

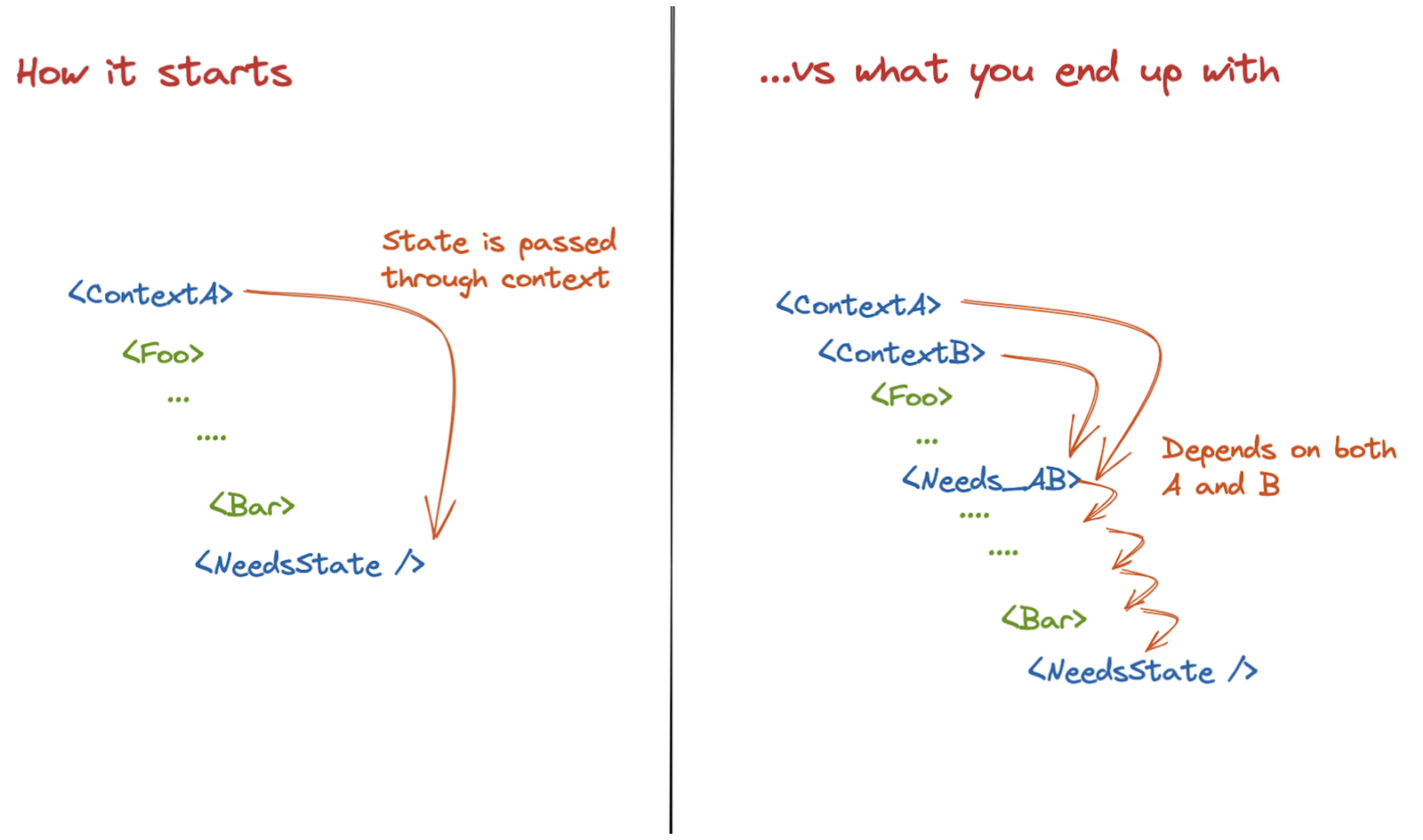


This scenario poses a challenge for traditional Virtual DOM based frameworks, which must update the entire tree affected by a state invalidation. In essence, rendering performance is a function of the number of components in that tree. We can work around this by memoizing parts of the component tree using `memo` or `useMemo` so that the framework receives the same objects. When nothing has changed, this lets the framework skip rendering some parts of the tree.

Whilst this sounds reasonable in theory, the reality is often a lot messier. In practice, as codebases grow it becomes difficult to determine where these optimizations should be placed. Frequently, even well-intentioned memoization is rendered ineffective by unstable dependency values. Since hooks have no explicit dependency tree that can be analyzed, tooling can't help developers diagnose ***why*** dependencies are unstable.

Context chaos

Another common workaround teams reach for state sharing is to place state into context. This allows short-circuiting rendering by potentially skipping render for components between the context provider and consumers. There is a catch though: only the value passed to the context provider can be updated, and only as a whole. Updating a property on an object exposed via context does not update consumers of that context - granular updates aren’t



Moving values into context seems like a worthwhile tradeoff at first, but the downsides of increasing component tree size just to share values eventually become a problem. Business logic inevitably ends up depending on multiple context values, which can force it to be implemented at a specific location in the tree. Adding a component that subscribes to context in the middle of the tree is costly, as it reduces the number of components that can be skipped when updating context. What's more, any components beneath the subscriber must now be rendered again. The only solution to this problem is heavy use of memoization, which brings us back to the problems inherent to memoization.

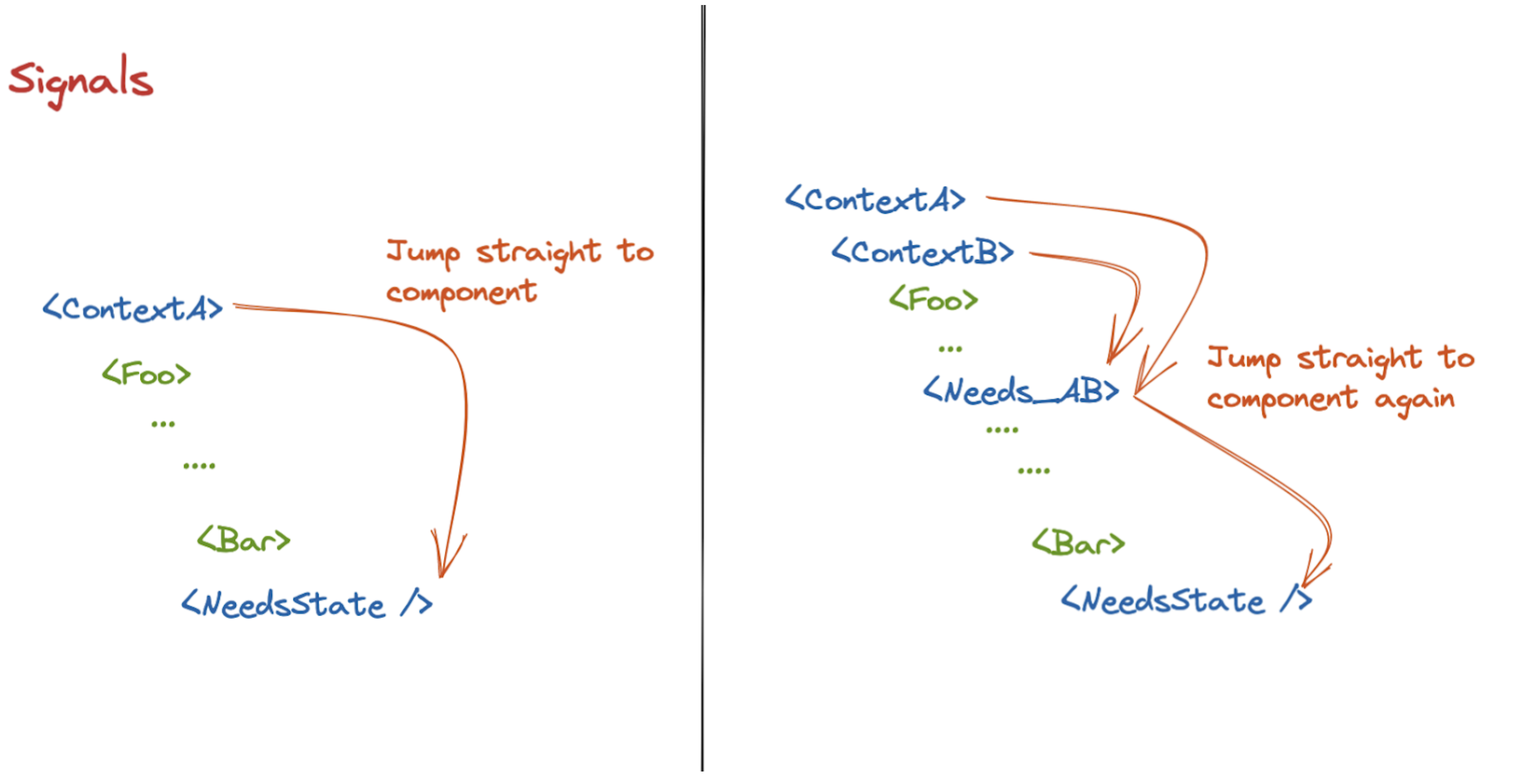
In search of a better way to manage state

We went back to the drawing board in search of a next generation state primitive. We wanted to create something that simultaneously addressed the problems in current solutions. Manual framework integration, over-reliance on memoization, suboptimal use of context, and lack of programmatic observability felt backwards.

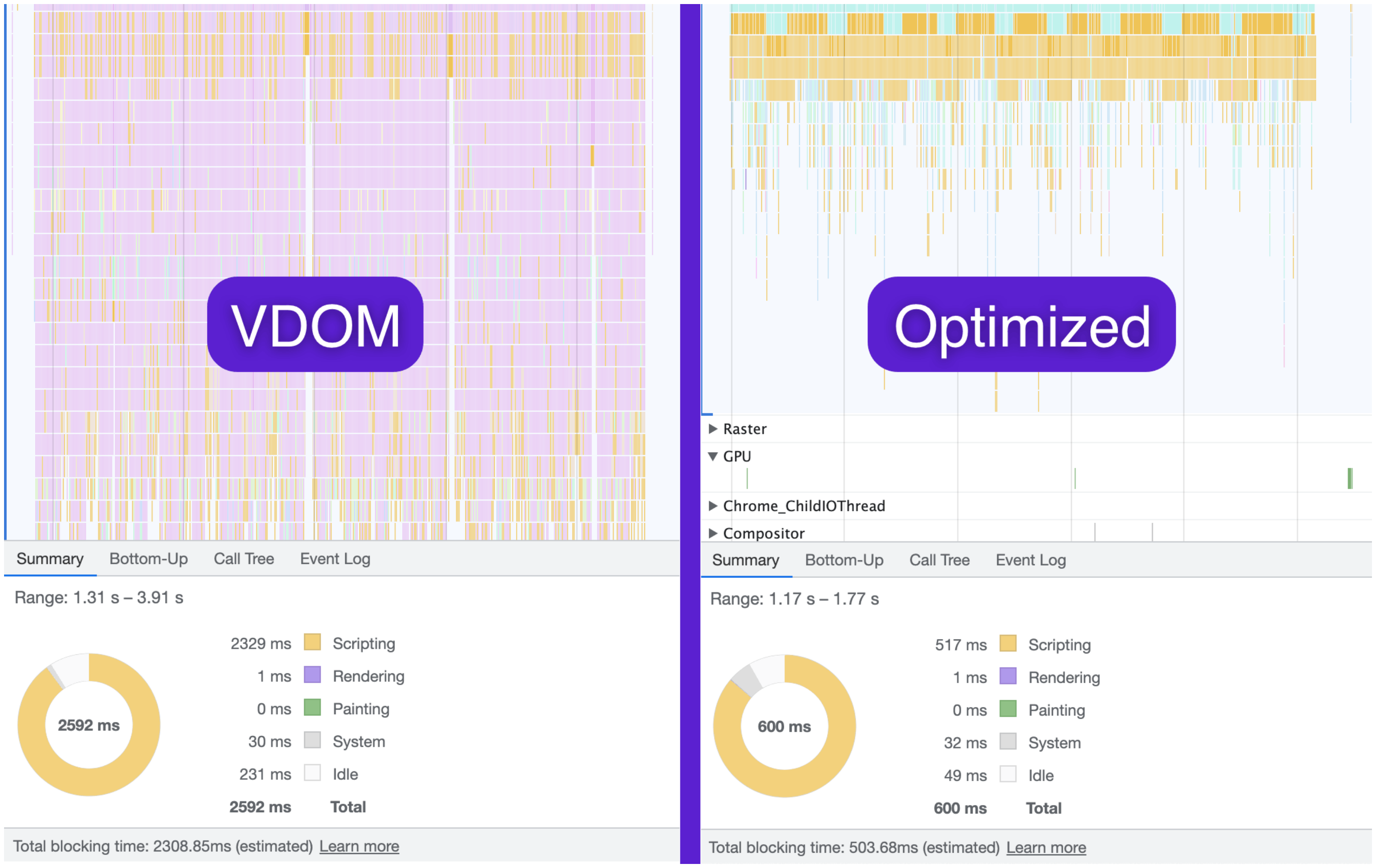
Developers need to "opt in" to performance with these strategies. What if we could reverse that and provide a system that was **fast by default**, making best-case performance updates regardless of whether that state is global, passed via props or context, or local to a component.

Signals to the future

The main idea behind signals is that instead of passing a value directly through the component tree, we pass a signal object containing the value (similar to a `ref`). When a signal's value changes, the signal itself stays the same. As a result, signals can be updated without re-rendering the components they've been passed through, since components see the signal and not its value. This lets us skip all of the expensive work of rendering components and jump immediately to the specific components in the tree that actually access the signal's value.



We’re exploiting the fact that an application's state graph is generally much shallower than its component tree. This leads to faster rendering, because far less work is required to update the state graph compared to the component tree. This difference is most apparent when measured in the browser - the screenshot below shows a DevTools Profiler trace for the same app measured twice: once using hooks as the state primitive and a second time using signals:



Range: 1.31 s – 3.91 s

2329 ms	🟨	Scripting
1 ms	🟪	Rendering
0 ms	🟩	Painting
30 ms	⬜	System
231 ms	⬜	Idle
2592 ms		**Total**

Total blocking time: 2308.85ms (estimated) Learn more

Range: 1.17 s – 1.77 s

517 ms	🟨	Scripting
1 ms	🟪	Rendering
0 ms	🟩	Painting
32 ms	⬜	System
49 ms	⬜	Idle
600 ms		**Total**

Total blocking time: 503.68ms (estimated) Learn more

The signals version vastly outperforms the update mechanism of any traditional Virtual DOM based framework. In some apps we've tested, signals are so much faster that it becomes difficult to find them in the flamegraph at all.

Signals flip the performance pitch around: instead of opting-in to performance via memoization or selectors, signals are fast by default. With signals, performance is opt-out (by not using signals).

To achieve this level of performance, signals were built on these key principles:

- **Lazy by default:** Only signals that are currently used somewhere are observed and updated - disconnected signals don't affect performance.
- **Optimal updates:** If a signal's value hasn't changed, components and effects that use that signal's value won't be updated, even if the signal's dependencies have changed.
- **Optimal dependency tracking:** The framework tracks which signals everything depends on for you - no dependency arrays like with hooks.

have nothing to do with rendering user interfaces.

Bringing signals to Preact

Having identified the right state primitive, we set about wiring it up to Preact. The thing we've always loved about hooks is that they can be used directly inside components. This is an ergonomic advantage compared to third-party state management solutions, which usually rely on "selector" functions or wrapping components in a special function to subscribe to state updates.

```
// Selector based subscription :(
function Counter() {
  const value = useSelector(state => state.count);
  // ...
}

// Wrapper function based subscription :(
const counterState = new Counter();

const Counter = observe(props => {
  const value = counterState.count;
  // ...
});
```

Neither approach felt satisfactory to us. The selector approach requires wrapping all state access in a selector, which becomes tedious for complex or nested state. The approach of wrapping components in a function requires manual effort to wrap components, which brings with it a host of issues like missing component names and static properties.

We've had the opportunity to work closely with many developers over the past few years. One common struggle, particularly for those new to (p)react, is that concepts like selectors and wrappers are additional paradigms that must be learned before feeling productive with each state management solution.

Ideally, we wouldn't need to know about selectors or wrapper functions and could simply access state directly within components:

```
 return (
   <button onClick={() => count++}>
     value: {count}
   </button>
 );
}
```

The code is clear and it's easy to understand what is going on, but unfortunately it doesn't work. The component doesn't update when clicking the button because there is no way to know that `count` has changed.

We couldn’t get this scenario out of our heads though. What could we do to make a model this clear into a reality? We began to prototype various ideas and implementations using Preact's pluggable renderer. It took time, but we eventually landed on a way to make it happen:

Run in REPL

```
// Imagine this is some global state that the whole app needs access to:
const count = signal(0);

function Counter() {
 return (
   <button onClick={() => count.value++}>
     Value: {count.value}
   </button>
 );
}
```

There are no selectors, no wrapper functions, nothing. Accessing the signal's value is enough for the component to know that it needs to update when that signal’s value changes. After testing out the prototype in a few apps, it was clear we were onto something. Writing code this way felt intuitive and didn't require any mental gymnastics to keep things working optimally.

Can we go even faster?

value changes, what if we only re-rendered the text? Better still, what if we bypassed the Virtual DOM entirely and updated the text directly in the DOM?

```
const count = signal(0);

// Instead of this:
<p>Value: {count.value}</p>

// … we can pass the signal directly into JSX:
<p>Value: {count}</p>

// … or even passing them as DOM properties:
<input value={count} onInput={...} />
```

So yeah, we did that too. You can pass a signal directly into the JSX anywhere you'd normally use a string. The signal's value will be rendered as text, and it will automatically update itself when the signal changes. This also works for props.

Next Steps

If you’re curious and want to jump right in, head over to our documentation for signals. We’d love to hear how you're going to use them.

Remember that there is no rush to switch to signals. Hooks will continue to be supported, and they work great with signals too! We recommend gradually trying out signals, starting with a few components to get used to the concepts.