React Labs: What We've Been Working On – March 2023

March 22, 2023 by Joseph Savona, Josh Story, Lauren Tan, Mengdi Chen, Samuel Susla, Sathya Gunasekaran, Sebastian Markbåge, and Andrew Clark

In React Labs posts, we write about projects in active research and development. We've made significant progress on them since our last update, and we'd like to share what we learned.

React Server Components

React Server Components (or RSC) is a new application architecture designed by the React team.

We've first shared our research on RSC in an introductory talk and an RFC. To recap them, we are introducing a new kind of component—Server Components—that run ahead of time and are excluded from your JavaScript bundle. Server Components can run during the build, letting you read from the filesystem or fetch static content. They can also run on the server, letting you access your data layer without having to build an API. You can pass data by props from Server Components to the interactive Client Components in the browser.

RSC combines the simple "request/response" mental model of server-centric Multi-Page Apps with the seamless interactivity of client-centric Single-Page Apps, giving you the best of both worlds.

Since our last update, we have merged the React Server Components RFC to ratify the proposal. We resolved outstanding issues with the React Server Module Conventions proposal, and reached consensus with our partners to go with the `"use client"` convention. These documents also act as specification for what an RSC-compatible implementation should support.

The biggest change is that we introduced `async / await` as the primary way to do data fetching from Server Components. We also plan to support data loading from the client by introducing a new Hook called `use` that unwraps Promises. Although we can't support `async / await` in arbitrary components in client-only apps, we plan to add support for it when you structure your client-only app similar to how RSC apps are structured.

Now that we have data fetching pretty well sorted, we're exploring the other direction: sending data from the client to the server, so that you can execute database mutations and implement forms. We're doing this by letting you pass Server Action functions across the server/client boundary, which the client can then call, providing seamless RPC. Server Actions also give you progressively enhanced forms before JavaScript loads.

React Server Components has shipped in Next.js App Router. This showcases a deep integration of a router that really buys into RSC as a primitive, but it's not the only way to build a RSC-compatible router and framework. There's a clear separation for features provided by the RSC spec and implementation. React Server Components is meant as a spec for components that work across compatible React frameworks.

We generally recommend using an existing framework, but if you need to build your own custom framework, it is possible. Building your own RSC-compatible framework is not as easy as we'd like it to be, mainly due to the deep bundler integration needed. The current generation of bundlers are great for use on the client, but they weren't designed with first-class support for splitting a single module graph between the server and the client. This is why we're now partnering directly with bundler developers to get the primitives for RSC built-in.

Asset Loading

Suspense lets you specify what to display on the screen while the data or code for your components is still being loaded. This lets your users progressively see more content while the page is loading as well as during the router navigations that load more data and code. However, from the user's perspective, data loading and rendering do not tell the whole story when considering whether new content is ready. By default, browsers load stylesheets, fonts, and images independently, which can lead to UI jumps and consecutive layout shifts.

We're working to fully integrate Suspense with the loading lifecycle of stylesheets, fonts, and images, so that React takes them into account to determine whether the content is ready to be displayed. Without any change to the way you author your React components, updates will behave in a more coherent and pleasing manner. As an optimization, we will also provide a manual way to preload assets like fonts directly from components.

We are currently implementing these features and will have more to share soon.

Document Metadata

Different pages and screens in your app may have different metadata like the `<title>` tag, description, and other `<meta>` tags specific to this screen. From the maintenance perspective, it's more scalable to keep this information close to the React component for that page or screen. However, the HTML tags for this metadata need to be in the document `<head>` which is typically rendered in a component at the very root of your app.

Today, people solve this problem with one of the two techniques.

One technique is to render a special third-party component that moves `<title>`, `<meta>`, and other tags inside it into the document `<head>`. This works for major browsers but there are many clients which do not run client-side JavaScript, such as Open Graph parsers, and so this technique is not universally suitable.

Another technique is to server-render the page in two parts. First, the main content is rendered and all such tags are collected. Then, the `<head>` is rendered with these tags. Finally, the `<head>` and the main content are sent to the browser. This approach works, but it prevents you from taking advantage of the React 18's Streaming Server Renderer because you'd have to wait for all content to render before sending the `<head>`.

This is why we're adding built-in support for rendering `<title>`, `<meta>`, and metadata `<link>` tags anywhere in your component tree out of the box. It would work the same way in all environments, including fully client-side code, SSR, and in the future, RSC. We will share more details about this soon.

React Optimizing Compiler

Since our previous update we've been actively iterating on the design of React Forget, an optimizing compiler for React. We've previously talked about it as an "auto-memoizing compiler", and that is true in some sense. But building the compiler has helped us understand React's programming model even more deeply. A better way to understand React Forget is as an automatic *reactivity* compiler.

The core idea of React is that developers define their UI as a function of the current state. You work with plain JavaScript values — numbers, strings, arrays, objects — and use standard JavaScript idioms — if/else, for, etc — to describe your component logic. The mental model is that React will re-render whenever the application state changes. We believe this simple mental model and keeping close to JavaScript semantics is an important principle in React's programming model.

The catch is that React can sometimes be *too* reactive: it can re-render too much. For example, in JavaScript we don't have cheap ways to compare if two objects or arrays are equivalent (having the same keys and values), so creating a new object or array on each render may cause React to do more work than it strictly needs to. This means developers have to explicitly memoize components so as to not over-react to changes.

Our goal with React Forget is to ensure that React apps have just the right amount of reactivity by default: that apps re-render only when state values *meaningfully* change. From an implementation perspective this means automatically memoizing, but we believe that the reactivity framing is a better way to understand React and Forget. One way to think about this is that React currently re-renders when object identity changes. With Forget, React re-renders when the semantic value changes — but without incurring the runtime cost of deep comparisons.

In terms of concrete progress, since our last update we have substantially iterated on the design of the compiler to align with this automatic reactivity approach and to incorporate feedback from using the compiler internally. After some significant refactors to the compiler starting late last year, we've now begun using the compiler in production in limited areas at Meta. We plan to open-source it once we've proved it in production.

Finally, a lot of people have expressed interest in how the compiler works. We're looking forward to sharing a lot more details when we prove the compiler and open-source it. But there are a few bits we can share now:

The core of the compiler is almost completely decoupled from Babel, and the core compiler API is (roughly) old AST in, new AST out (while retaining source location data). Under the hood we use a custom code representation and transformation pipeline in order to do low-level semantic analysis. However, the primary public interface to the compiler will be via Babel and other build system plugins. For ease of testing we currently have a Babel plugin which is a very thin wrapper that calls the compiler to generate a new version of each function and swap it in.

As we refactored the compiler over the last few months, we wanted to focus on refining the core compilation model to ensure we could handle complexities such as conditionals, loops, reassignment, and mutation. However, JavaScript has a lot of ways to express each of those features: if/else, ternaries, for, for-in, for-of, etc. Trying to support the full language up-front would have delayed the point where we could validate the core model. Instead, we started with a small but representative subset of the language: let/const, if/else, for loops, objects, arrays, primitives, function calls, and a few other features. As we gained confidence in the core model and refined our internal abstractions, we expanded the supported language subset. We're also explicit about syntax we don't yet support, logging diagnostics and skipping compilation for unsupported input. We have utilities to try the compiler on Meta's codebases and see what unsupported features are most common so we can prioritize those next. We'll continue incrementally expanding towards supporting the whole language.

Making plain JavaScript in React components reactive requires a compiler with a deep understanding of semantics so that it can understand exactly what the code is doing. By taking this approach, we're creating a system for reactivity within JavaScript that lets you write product code of any complexity with the full expressivity of the language, instead of being limited to a domain specific language.

Offscreen Rendering

Offscreen rendering is an upcoming capability in React for rendering screens in the background without additional performance overhead. You can think of it as a version of the `content-visibility` CSS property that works not only for DOM elements but React components, too. During our research, we've discovered a variety of use cases:

- A router can prerender screens in the background so that when a user navigates to them, they're instantly available.
- A tab switching component can preserve the state of hidden tabs, so the user can switch between them without losing their progress.
- A virtualized list component can prerender additional rows above and below the visible window.
- When opening a modal or popup, the rest of the app can be put into "background" mode so that events and updates are disabled for everything except the modal.

Most React developers will not interact with React's offscreen APIs directly. Instead, offscreen rendering will be integrated into things like routers and UI libraries, and then developers who use those libraries will automatically benefit without additional work.

The idea is that you should be able to render any React tree offscreen without changing the way you write your components. When a component is rendered offscreen, it does not actually *mount* until the component becomes visible — its effects are not fired. For example, if a component uses `useEffect` to log analytics when it appears for the first time, prerendering won't mess up the accuracy of those analytics. Similarly, when a component goes offscreen, its effects are unmounted, too. A key feature of offscreen rendering is that you can toggle the visibility of a component without losing its state.

Since our last update, we've tested an experimental version of prerendering internally at Meta in our React Native apps on Android and iOS, with positive performance results. We've also improved how offscreen rendering works with Suspense — suspending inside an offscreen tree will not trigger Suspense fallbacks. Our remaining work involves finalizing the primitives that are exposed to library developers. We expect to publish an RFC later this year, alongside an experimental API for testing and feedback.

Transition Tracing

The Transition Tracing API lets you detect when React Transitions become slower and investigate why they may be slow. Following our last update, we have completed the initial design of the API and published an RFC. The basic capabilities have also been implemented. The project is currently on hold. We welcome feedback on the RFC and look forward to resuming its development to provide a better performance measurement tool for React. This will be particularly useful with routers built on top of React Transitions, like the Next.js App Router.

In addition to this update, our team has made recent guest appearances on community podcasts and livestreams to speak more on our work and answer questions.

- Dan Abramov and Joe Savona were interviewed by Kent C. Dodds on his YouTube channel, where they discussed concerns around React Server Components.
- Dan Abramov and Joe Savona were guests on the JSParty podcast and shared their thoughts about the future of React.

Thanks to Andrew Clark, Dan Abramov, Dave McCabe, Luna Wei, Matt Carroll, Sean Keegan, Sebastian Silbermann, Seth Webster, and Sophie Alpert for reviewing this post.

Thanks for reading, and see you in the next update!

PREVIOUS
React Canaries: Enabling Incremental Feature Rollout Outside Meta

NEXT
Introducing react.dev
