Streaming

Remix supports the web streaming API as a first-class citizen. Additionally, JavaScript server runtimes have support for streaming responses to the client.

> NOTE: Deferred UX goals rely on streaming responses. Some popular hosts do not support streaming responses. In general, any host built around AWS Lambda does not support streaming and any bare metal / VM provider will. Make sure your hosting platform supports before using this API.

The problem

Imagine a scenario where one of your routes' loaders needs to retrieve some data that for one reason or another is quite slow. For example, let's say you're showing the user the location of a package that's being delivered to their home:

```
import type { LoaderArgs } from "@remix-run/node"; // or cloudflare/deno
import { json } from "@remix-run/node"; // or cloudflare/deno
import { useLoaderData } from "@remix-run/react";

import { getPackageLocation } from "~/models/packages";

export async function loader({ params }: LoaderArgs) {
  const packageLocation = await getPackageLocation(
    params.packageId
  );

  return json({ packageLocation });
}

export default function PackageRoute() {
  const { packageLocation } =
    useLoaderData<typeof loader>();

  return (
    <main>
```

```
        <h1>Let's locate your package</h1>
        <p>
          Your package is at {packageLocation.latitude} lat
          and {packageLocation.longitude} long.
        </p>
      </main>
    );
  }
```

We'll assume that `getPackageLocation` is slow. This will lead to initial page load times and transitions to that route to take as long as the slowest bit of data. Before reaching for rendering a fallback we recommend exploring ways to speed up that slow data, though not always possible here are a few things to explore first:

- Speed up the slow thing (😄).
  - Optimize DB queries.
  - Add caching (LRU, Redis, etc).
  - Use a different data source.
- Load data concurrently loading with `Promise.all` (we have nothing to make concurrent in our example, but it might help a bit in other situations).

If initial page load is not a critical metric for your application, you can also explore the following options that can improve the perceived performance of your application client side only:

- Use the `prefetch` prop on `<Link />`.
- Add a global transition spinner.
- Add a localized skeleton UI.

If these approaches don't work well, then you may feel forced to move the slow data out of the Remix loader into a client-side fetch (and show a skeleton fallback UI while loading). In this case you'd render the fallback UI on the server render and fire off the fetch for the data on the client. This is actually not so terrible from a DX standpoint thanks to `useFetcher`. And from a UX standpoint this improves the loading experience for both client-side transitions as well as initial page load. So it does seem to solve the problem.

But it's still sub-optimal for two reasons:

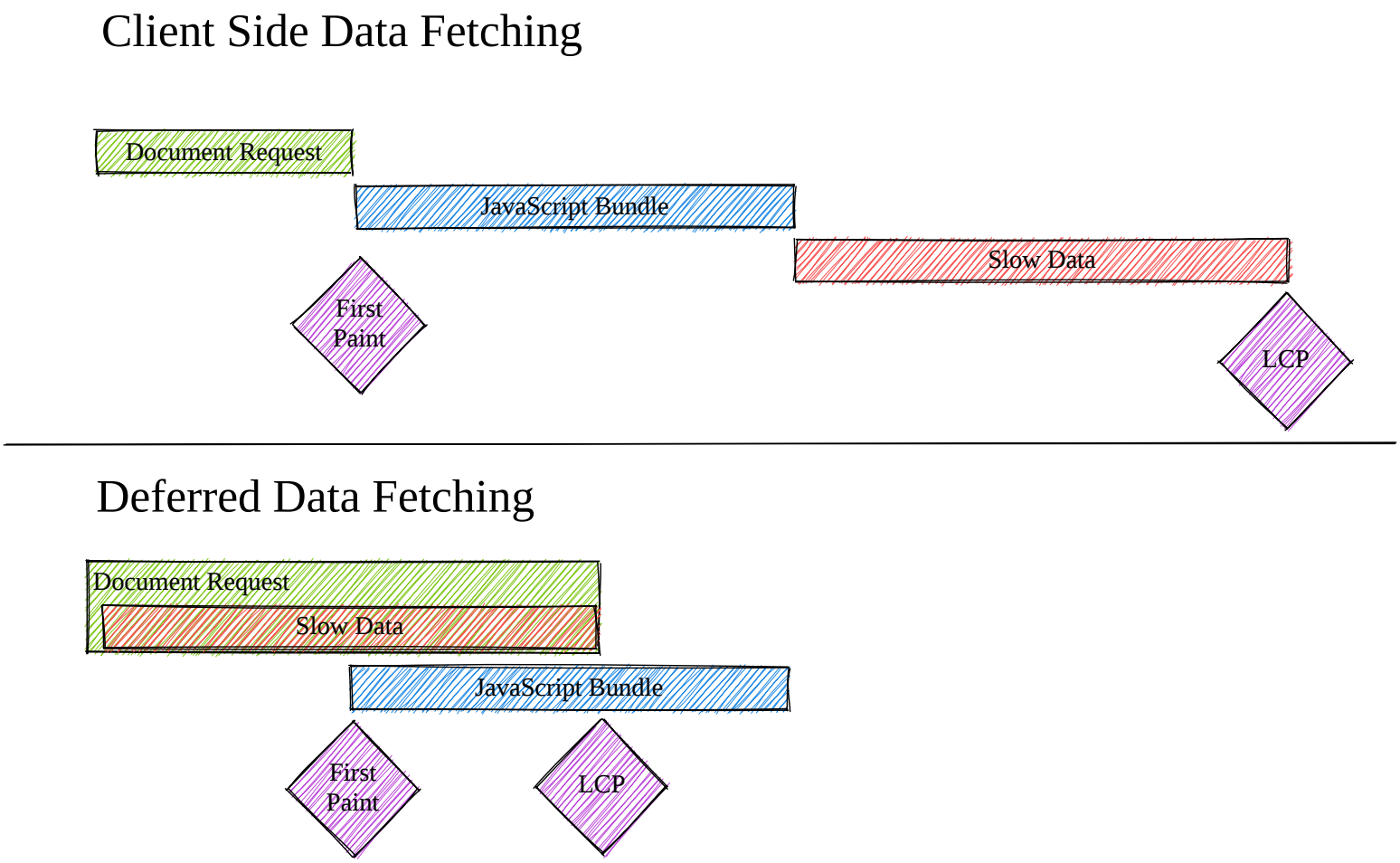
1. Client-side fetching puts your data request on a waterfall: document → JavaScript → data fetch
2. Your code can't easily switch between client-side fetching and server-side rendering (more on this later).

The solution

Remix takes advantage of React 18's streaming and server-side support for `<Suspense />` boundaries using the `defer` Response utility and `<Await />` component / `useAsyncValue` hook. By using these APIs,

you can solve both of these problems:

1. Your data is no longer on a waterfall: document & data (in parallel) → JavaScript
2. Your can easily switch between streaming and waiting for the data



Let's take a dive into how to accomplish this.

Enable React 18 Streaming

First, to enable streaming with React 18, you'll update your `entry.server.tsx` file to use `renderToPipeableStream`. Here's a simple (and incomplete) version of that:

app/entry.server.tsx

```tsx
import { PassThrough } from "stream";
import { renderToPipeableStream } from "react-dom/server";
import { RemixServer } from "@remix-run/react";
import { Response } from "@remix-run/node"; // or cloudflare/deno
import type {
  EntryContext,
  Headers,
} from "@remix-run/node"; // or cloudflare/deno

export default function handleRequest(
```

```
  request: Request,
  responseStatusCode: number,
  responseHeaders: Headers,
  remixContext: EntryContext
) {
  return new Promise((resolve) => {
    const { pipe } = renderToPipeableStream(
      <RemixServer
        context={remixContext}
        url={request.url}
      />,
      {
        onShellReady() {
          const body = new PassThrough();

          responseHeaders.set("Content-Type", "text/html");

          resolve(
            new Response(body, {
              status: responseStatusCode,
              headers: responseHeaders,
            })
          );
          pipe(body);
        },
      }
    );
  });
}
```

▶ For a more complete example, expand this

Then on the client you need to make sure you're hydrating properly with the React 18 `hydrateRoot` API:

app/entry.client.tsx

```
import { RemixBrowser } from "@remix-run/react";
import { hydrateRoot } from "react-dom/client";

hydrateRoot(document, <RemixBrowser />);
```

With just that in place, you're unlikely to see any significant performance improvement. But with that alone you can now use `React.lazy` to SSR components but delay hydration on the client. This can open up network bandwidth for more critical things like styles, images, and fonts leading to a better LCP and TTI.

Using `defer`

With React streaming set up, now you can start adding `Await` usage for your slow data requests where you'd rather render a fallback UI. Let's do that for our example above:

```tsx
import { Suspense } from "react";
import type { LoaderArgs } from "@remix-run/node"; // or cloudflare/deno
import { defer } from "@remix-run/node"; // or cloudflare/deno
import { Await, useLoaderData } from "@remix-run/react";

import { getPackageLocation } from "~/models/packages";

export function loader({ params }: LoaderArgs) {
  const packageLocationPromise = getPackageLocation(
    params.packageId
  );

  return defer({
    packageLocation: packageLocationPromise,
  });
}

export default function PackageRoute() {
  const data = useLoaderData<typeof loader>();

  return (
    <main>
      <h1>Let's locate your package</h1>
      <Suspense
        fallback={<p>Loading package location...</p>}
      >
        <Await
          resolve={data.packageLocation}
          errorElement={
            <p>Error loading package location!</p>
          }
        >
          {(packageLocation) => (
            <p>
              Your package is at {packageLocation.latitude}{" "}
              lat and {packageLocation.longitude} long.
            </p>
          )}
        </Await>
      </Suspense>
    </main>
  );
}
```

▶ Alternatively, you can use the `useAsyncValue` hook:

Evaluating the solution

So rather than waiting for the whole `document -> JavaScript -> hydrate -> request` cycle, with streaming we start the request for the slow data as soon as the document request comes in. This can significantly speed up the user experience.

Remix treats any `Promise` values as deferred data. These will be streamed to the client as each `Promise` resolves. If you'd like to prevent streaming for critical data, just use `await` and it will be included in the initial presentation of the document to the user.

```
return defer({
  // critical data (not deferred):
  packageLocation: await packageLocationPromise,
  // non-critical data (deferred):
  packageLocation: packageLocationPromise,
});
```

Because of this, you can A/B test deferring, or even determine whether to defer based on the user or data being requested:

```
export async function loader({
  request,
  params,
}: LoaderArgs) {
  const packageLocationPromise = getPackageLocation(
    params.packageId
  );
  const shouldDefer = await shouldDeferPackageLocation(
    request,
    params.packageId
  );

  return defer({
    packageLocation: shouldDefer
      ? packageLocationPromise
      : await packageLocationPromise,
  });
}
```

That `shouldDeferPackageLocation` could be implemented to check the user making the request, whether the package location data is in a cache, the status of an A/B test, or whatever else you want. This is

pretty sweet 🍭

Also, because this happens at request time (even on client transitions), makes use of the URL via nested routing (rather than requiring you to render before you know what data to fetch), and it's all just regular HTTP, we can prefetch and cache the response! Meaning client-side transitions can be *much* faster (in fact, there are plenty of situations when the user may never be presented with the fallback at all).

Another thing that's not immediately recognizable is if your server can finish loading deferred data before the client can load the JavaScript and hydrate, the server will stream down the HTML and add it to the document *before React hydrates*, thereby increasing performance for those on slow networks. This works even if you never add `<Scripts />` to the page thanks to React 18's support for out-of-order streaming.

FAQ

Why not defer everything by default?

The Remix defer API is another lever Remix offers to give you a nice way to choose between trade-offs. Do you want a better TTFB (Time to first byte)? Defer stuff. Do you want a low CLS (Content Layout Shift)? Don't defer stuff. You want a better TTFB, but also want a lower CLS? Defer just the slow and unimportant stuff.

It's all trade-offs, and what's neat about the API design is that it's well suited for you to do easy experimentation to see which trade-offs lead to better results for your real-world key indicators.

When does the fallback render?

The `<Await />` component will only throw the promise up the `<Suspense>` boundary on the initial render of the `<Await />` component with an unsettled promise. It will not re-render the fallback if props change. Effectively, this means that you *will not* get a fallback rendered when a user submits a form and loader data is revalidated. You *will* get a fallback rendered when the user navigates to the same route with different params (in the context of our above example, if the user selects from a list of packages on the left to find their location on the right).

This may feel counter-intuitive at first, but stay with us, we really thought this through and it's important that it works this way. Let's imagine a world without the deferred API. For those scenarios you're probably going to want to implement Optimistic UI for form submissions/revalidation.

When you decide you'd like to try the trade-offs of `defer`, we don't want you to have to change or remove those optimizations because we want you to be able to easily switch between deferring some data and not deferring it. So we ensure that your existing optimistic states work the same way. If we didn't do this, then you could experience what we call "Popcorn UI" where submissions of data trigger the fallback loading state instead of the optimistic UI you'd worked hard on.

So just keep this in mind: **Deferred is 100% only about the initial load of a route and it's params.**