RSC From Scratch. Part 1: Server Components

In this technical deep dive, we'll implement a very simplified version of React Server Components (RSC) from the scratch.
This deep dive will be published in several parts:

- **Part 1: Server Components**
- Part 2: Client Components *(not written yet)*
- Part 3: TBD *(not written yet)*

Seriously, this is a deep dive!

This deep dive doesn't explain the benefits of React Server Components, how to implement an app using RSC, or how to implement a framework using them. Instead, it walks you through the process of "inventing" them on your own from scratch.

🔬 **This is a deep dive for people who like to learn new technologies by implementing them from scratch.**
It assumes some background in web programming and some familiarity with React.

🚧 **This deep dive is not intended as an introduction to how to *use* Server Components.** We are working to document Server Components on the React website. In the meantime, if your framework supports Server Components, please refer to its docs.

😳 **For pedagogical reasons, our implementation will be significantly less efficient than the real one used by React.**
We will note future optimization opportunities in the text, but we will strongly prioritize conceptual clarity over efficiency.

Let's jump back in time...

Suppose that you woke up one morning and found out it's 2003 again. Web development is still in its infancy. Let's say you want to create a personal blog website that shows content from text files on your server. In PHP, it could look like this:

```
<?php
  $author = "Jae Doe";
  $post_content = @file_get_contents("./posts/hello-world.txt");
?>
<html>
  <head>
    <title>My blog</title>
  </head>
  <body>
```

```
    <nav>
      <a href="/">Home</a>
      <hr>
    </nav>
    <article>
      <?php echo htmlspecialchars($post_content); ?>
    </article>
    <footer>
      <hr>
      <p><i>(c) <?php echo htmlspecialchars($author); ?>, <?php echo date("Y"); ?></i></p
    </footer>
  </body>
</html>
```

(We're going to pretend that tags like `<nav>`, `<article>`, and `<footer>` existed back then to keep the HTML easy to read.)

When you open `http://locahost:3000/hello-world` in your browser, this PHP script returns an HTML page with the blog post from `./posts/hello-world.txt`. An equivalent Node.js script written using the today's Node.js APIs might look like this:

```
import { createServer } from 'http';
import { readFile } from 'fs/promises';
import escapeHtml from  'escape-html';

createServer(async (req, res) => {
  const author = "Jae Doe";
  const postContent = await readFile("./posts/hello-world.txt", "utf8");
  sendHTML(
    res,
    `<html>
      <head>
        <title>My blog</title>
      </head>
      <body>
        <nav>
          <a href="/">Home</a>
          <hr />
        </nav>
        <article>
          ${escapeHtml(postContent)}
        </article>
        <footer>
          <hr>
          <p><i>(c) ${escapeHtml(author)}, ${new Date().getFullYear()}</i></p>
        </footer>
      </body>
    </html>`
  );
}).listen(8080);

function sendHTML(res, html) {
  res.writeHead(200, { "Content-Type": "text/html" });
  res.end(html);
}
```

Open this example in a sandbox.

Imagine that you could take a CD-ROM with a working Node.js engine back to 2003, and you could run this code on the server. If you wanted to bring a React-flavored paradigm to that world, what features would you add, and in what order?

Step 1: Let's invent JSX

The first thing that's not ideal about the code above is direct string manipulation. Notice you've had to call `escapeHtml(postContent)` to ensure that you don't accidentally treat content from a text file as HTML.

One way you could solve this is by splitting your logic from your "template", and then introducing a separate templating language that provides a way to inject dynamic values for text and attributes, escapes text content safely, and provides domain-specific syntax for conditions and loops. That's the approach taken by some of the most popular server-centric frameworks in 2000s.

However, your existing knowledge of React might inspire you to do this instead:

```
createServer(async (req, res) => {
  const author = "Jae Doe";
  const postContent = await readFile("./posts/hello-world.txt", "utf8");
  sendHTML(
    res,
    <html>
      <head>
        <title>My blog</title>
      </head>
      <body>
        <nav>
          <a href="/">Home</a>
          <hr />
        </nav>
        <article>
          {postContent}
        </article>
        <footer>
          <hr />
          <p><i>(c) {author}, {new Date().getFullYear()}</i></p>
        </footer>
      </body>
    </html>
  );
}).listen(8080);
```

This looks similar, but our "template" is not a string anymore. Instead of writing string interpolation code, we're putting a subset of XML into JavaScript. In other words, we've just "invented" JSX. JSX lets you keep markup close to the related rendering logic, but unlike string interpolation, it prevents mistakes like mismatching open/close HTML tags or forgetting to escape text content.

Under the hood, JSX produces a tree of objects that look like this:

```
// Slightly simplified
{
  $$typeof: Symbol.for("react.element"), // Tells React it's a JSX element (e.g. <html>)
  type: 'html',
  props: {
    children: [
      {
        $$typeof: Symbol.for("react.element"),
        type: 'head',
        props: {
          children: {
            $$typeof: Symbol.for("react.element"),
            type: 'title',
            props: { children: 'My blog' }
          }
        }
      },
      {
        $$typeof: Symbol.for("react.element"),
        type: 'body',
        props: {
          children: [
            {
              $$typeof: Symbol.for("react.element"),
              type: 'nav',
              props: {
                children: [{
                  $$typeof: Symbol.for("react.element"),
                  type: 'a',
                  props: { href: '/', children: 'Home' }
                }, {
                  $$typeof: Symbol.for("react.element"),
                  type: 'hr',
                  props: null
                }]
              }
            },
            {
              $$typeof: Symbol.for("react.element"),
              type: 'article',
              props: {
                children: postContent
              }
            },
            {
              $$typeof: Symbol.for("react.element"),
              type: 'footer',
              props: {
                /* ...And so on... */
              }
            }
          ]
        }
      }
    ]
  }
}
```

However, in the end what you need to send to the browser is HTML — not a JSON tree. (At least, for now!)

Let's write a function that turns your JSX to an HTML string. To do this, we'll need to specify how different types of nodes (a string, a number, an array, or a JSX node with children) should turn into pieces of HTML:

```
function renderJSXToHTML(jsx) {
  if (typeof jsx === "string" || typeof jsx === "number") {
    // This is a string. Escape it and put it into HTML directly.
    return escapeHtml(jsx);
  } else if (jsx == null || typeof jsx === "boolean") {
    // This is an empty node. Don't emit anything in HTML for it.
    return "";
  } else if (Array.isArray(jsx)) {
    // This is an array of nodes. Render each into HTML and concatenate.
    return jsx.map((child) => renderJSXToHTML(child)).join("");
  } else if (typeof jsx === "object") {
    // Check if this object is a React JSX element (e.g. <div />).
    if (jsx.$$typeof === Symbol.for("react.element")) {
      // Turn it into an an HTML tag.
      let html = "<" + jsx.type;
      for (const propName in jsx.props) {
        if (jsx.props.hasOwnProperty(propName) && propName !== "children") {
          html += " ";
          html += propName;
          html += "=";
          html += escapeHtml(jsx.props[propName]);
        }
      }
      html += ">";
      html += renderJSXToHTML(jsx.props.children);
      html += "</" + jsx.type + ">";
      return html;
    } else throw new Error("Cannot render an object.");
  } else throw new Error("Not implemented.");
}
```

Open this example in a sandbox.

Give this a try and see the HTML being rendered and served!

Turning JSX into an HTML string is usually known as "Server-Side Rendering" (SSR). **It is important note that RSC and SSR are two very different things (that tend to be used together).** In this guide, we're *starting* from SSR because it's a natural first thing you might try to do in a server environment. However, this is only the first step, and you will see significant differences later on.

Step 2: Let's invent components

After JSX, the next feature you'll probably want is components. Regardless of whether your code runs on the client or on the server, it makes sense to split the UI apart into different pieces, give them names, and pass information to them by props.

Let's break the previous example apart into two components called `BlogPostPage` and `Footer`:

```
function BlogPostPage({ postContent, author }) {
  return (
    <html>
      <head>
        <title>My blog</title>
      </head>
      <body>
        <nav>
          <a href="/">Home</a>
          <hr />
        </nav>
        <article>
          {postContent}
        </article>
        <Footer author={author} />
      </body>
    </html>
  );
}

function Footer({ author }) {
  return (
    <footer>
      <hr />
      <p>
        <i>
          (c) {author} {new Date().getFullYear()}
        </i>
      </p>
    </footer>
  );
}
```

Then, let's replace inline JSX tree we had with `<BlogPostPage postContent={postContent} author={author} />`:

```
createServer(async (req, res) => {
  const author = "Jae Doe";
  const postContent = await readFile("./posts/hello-world.txt", "utf8");
  sendHTML(
    res,
    <BlogPostPage
      postContent={postContent}
      author={author}
    />
  );
}).listen(8080);
```

If you try to run this code without any changes to your `renderJSXToHTML` implementation, the resulting HTML will look broken:

```
<!-- This doesn't look like valid at HTML at all... -->
<function BlogPostPage({postContent,author}) {...}>
</function BlogPostPage({postContent,author}) {...}>
```

The problem is that our `renderJSXToHTML` function (which turns JSX into HTML) assumes that `jsx.type` is always a string with the HTML tag name (such as `"html"`, `"footer"`, or `"p"`):

```
if (jsx.$$typeof === Symbol.for("react.element")) {
  // Existing code that handles HTML tags (like <p>).
  let html = "<" + jsx.type;
  // ...
  html += "</" + jsx.type + ">";
  return html;
}
```

But here, `BlogPostPage` is a function, so doing `"<" + jsx.type + ">"` prints its source code. You don't want to send that function's code in an HTML tag name. Instead, let's *call* this function — and serialize the JSX it *returns* to HTML:

```
if (jsx.$$typeof === Symbol.for("react.element")) {
  if (typeof jsx.type === "string") { // Is this a tag like <div>?
    // Existing code that handles HTML tags (like <p>).
    let html = "<" + jsx.type;
    // ...
    html += "</" + jsx.type + ">";
    return html;
  } else if (typeof jsx.type === "function") { // Is it a component like <BlogPostPage>?
    // Call the component with its props, and turn its returned JSX into HTML.
    const Component = jsx.type;
    const props = jsx.props;
    const returnedJsx = Component(props);
    return renderJSXToHTML(returnedJsx);
  } else throw new Error("Not implemented.");
}
```

Now, if you encounter a JSX element like `<BlogPostPage author="Jae Doe" />` while generating HTML, you will *call* `BlogPostPage` as a function, passing `{ author: "Jae Doe" }` to that function. That function will return some more JSX. And you already know how to deal with JSX — you pass it back to `renderJSXToHTML` which continues generating HTML from it.

This change alone is enough to add support for components and passing props. Check it out:

Open this example in a sandbox.

Step 3: Let's add some routing

Now that we've got basic support for components working, it would be nice to add a few more pages to the blog.

Let's say a URL like `/hello-world` needs to show an individual blog post page with the content from `./posts/hello-world.txt`, while requesting the root `/` URL needs to show an a long index page with the content from every blog post. This means we'll want to add a new `BlogIndexPage` that shares the layout with `BlogPostPage` but has different content inside.

Currently, the `BlogPostPage` component represents the entire page, from the very `<html>` root. Let's extract the shared UI parts between pages (header and footer) out of the `BlogPostPage` into a reusable `BlogLayout` component:

```
function BlogLayout({ children }) {
  const author = "Jae Doe";
  return (
    <html>
      <head>
        <title>My blog</title>
      </head>
      <body>
        <nav>
          <a href="/">Home</a>
          <hr />
        </nav>
        <main>
          {children}
        </main>
        <Footer author={author} />
      </body>
    </html>
  );
}
```

We'll change the `BlogPostPage` component to only include the content we want to slot *inside* that layout:

```
function BlogPostPage({ postSlug, postContent }) {
  return (
    <section>
      <h2>
        <a href={"/" + postSlug}>{postSlug}</a>
      </h2>
      <article>{postContent}</article>
    </section>
  );
}
```

Here is how `<BlogPostPage>` will look when nested inside `<BlogLayout>`:

Home

---

hello-world

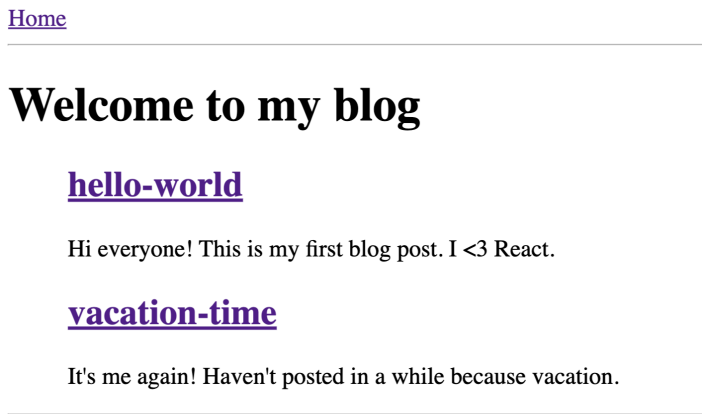
Hi everyone! This is my first blog post. I <3 React.

(c) Jae Doe 2023

Let's also add a *new* `BlogIndexPage` component that shows every post in `./posts/*.txt` one after another:

```
function BlogIndexPage({ postSlugs, postContents }) {
  return (
    <section>
      <h1>Welcome to my blog</h1>
      <div>
        {postSlugs.map((postSlug, index) => (
          <section key={postSlug}>
            <h2>
              <a href={"/" + postSlug}>{postSlug}</a>
            </h2>
            <article>{postContents[index]}</article>
          </section>
        ))}
      </div>
    </section>
  );
}
```

Then you can nest it inside `BlogLayout` too so that it has the same header and footer:



Finally, let's change the server handler to pick the page based on the URL, load the data for it, and render that page inside the layout:

```
createServer(async (req, res) => {
  const url = new URL(req.url, `http://${req.headers.host}`);
  let page;
  if (url.pathname === "/") {
    // We're on the index route which shows every blog post one by one.
    // Read all the files in the posts folder, and load their contents.
    const postFiles = await readdir("./posts");
    const postSlugs = postFiles.map((file) => file.slice(0, file.lastIndexOf(".")));
    const postContents = await Promise.all(
      postSlugs.map((postSlug) =>
        readFile("./posts/" + postSlug + ".txt", "utf8")
      )
    );
    page = <BlogIndexPage postSlugs={postSlugs} postContents={postContents} />;
  } else if (!url.pathname.includes(".")) { // Don't match static files (e.g. favicon.ico
    // We're showing an individual blog post.
    // Read the corresponding file from the posts folder.
    const postSlug = sanitizeFilename(url.pathname.slice(1));
```

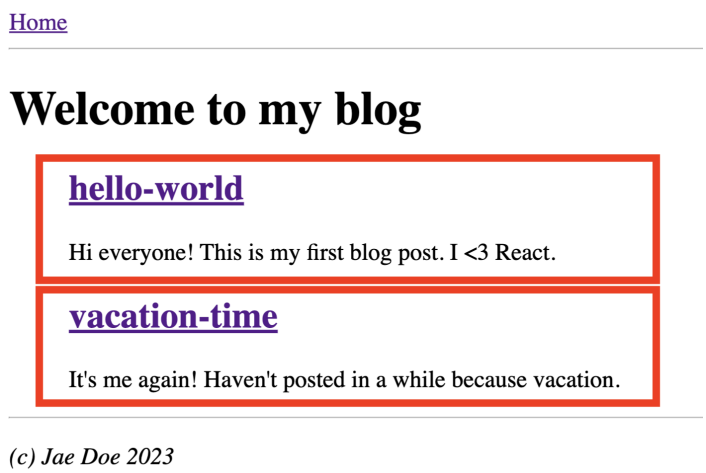
```
      const postContent = await readFile("./posts/" + postSlug + ".txt", "utf8");
      page = <BlogPostPage postSlug={postSlug} postContent={postContent} />;
    }
    if (page) {
      // Wrap the matched page into the shared layout.
      sendHTML(res, <BlogLayout>{page}</BlogLayout>);
    } else {
      res.writeHead(404);
      res.end();
    }
  }).listen(8080);
```

Now you can navigate around the blog. However, the code is getting a bit verbose and clunky. We'll solve that next.

Open this example in a sandbox.

Step 4: Let's invent async components

You might have noticed that this part of the `BlogIndexPage` and `BlogPostPage` components looks exactly the same:



It would be nice if we could somehow make this a reusable component. However, even if you extracted its rendering logic into a separate `Post` component, you would still have to somehow "plumb down" the `content` for each individual post:

```
  function Post({ slug, content }) { // Someone needs to pass down the `content` prop from
    return (
      <section>
```

```
      <h2>
        <a href={"/" + slug}>{slug}</a>
      </h2>
      <article>{content}</article>
    </section>
  )
}
```

Currently, the logic for loading `content` for posts is duplicated between here and here. We load it outside of the component hierarchy because the `readFile` API is asynchronous — so we can't use it directly in the component tree. *(Let's ignore that `fs` APIs have synchronous versions—this could've been a read from a database, or a call to some async third-party library.)*

Or can we?...

If you are used to client-side React, you might be used to the idea that you can't call an API like `fs.readFile` from a component. Even with traditional React SSR (server rendering), your existing intuition might tell you that each of your components needs to *also* be able to run in the browser — and so a server-only API like `fs.readFile` would not work.

But if you tried to explain this to someone in 2003, they would find this limitation rather odd. You can't `fs.readFile`, really?

Recall that we're approaching everything from the first principles. For now, we are *only* targeting the server environment, so we don't need to limit our components to code that runs in the browser. It is also perfectly fine for a component to be asynchronous, since the server can just wait with emitting HTML for it until its data has loaded and is ready to display.

Let's remove the `content` prop, and instead make `Post` an `async` function loads file content via an `await readFile()` call:

```
async function Post({ slug }) {
  const content = await readFile("./posts/" + slug + ".txt", "utf8");
  return (
    <section>
      <h2>
        <a href={"/" + slug}>{slug}</a>
      </h2>
      <article>{content}</article>
    </section>
  )
}
```

Similarly, let's make `BlogIndexPage` an `async` function that takes care of enumerating posts using `await readdir()`:

```
async function BlogIndexPage() {
  const postFiles = await readdir("./posts");
  const postSlugs = postFiles.map((file) =>
    file.slice(0, file.lastIndexOf("."))
  );
  return (
    <section>
```

```
        <h1>Welcome to my blog</h1>
        <div>
          {postSlugs.map((slug) => (
            <Post key={slug} slug={slug} />
          ))}
        </div>
      </section>
    );
  }
```

Now that `Post` and `BlogIndexPage` load data for themselves, let's create a `<Router>` component that matches the route:

```
function Router({ url }) {
  let page;
  if (url.pathname === "/") {
    page = <BlogIndexPage />;
  } else if (!url.pathname.includes(".")) {
    const postSlug = sanitizeFilename(url.pathname.slice(1));
    page = <BlogPostPage postSlug={postSlug} />;
  } else {
    const notFound = new Error("Not found.");
    notFound.statusCode = 404;
    throw notFound;
  }
  return <BlogLayout>{page}</BlogLayout>;
}
```

Finally, the top-level server handler can delegate all the rendering to the `<Router>`:

```
createServer(async (req, res) => {
  const url = new URL(req.url, `http://${req.headers.host}`);
  try {
    await sendResponse(res, <Router url={url} />);
  } catch (err) {
    console.error(err);
    res.writeHead(err.statusCode ?? 500);
    res.end();
  }
}).listen(8080);
```

But wait, we need to *actually* make `async` / `await` work inside components first. How do we do this?

Let's find the place in our `renderJSXToHTML` implementation where we call the component function:

```
  } else if (typeof jsx.type === "function") {
    const Component = jsx.type;
    const props = jsx.props;
    const returnedJsx = Component(props); // <--- This is where we're calling components
    return renderJSXToHTML(returnedJsx);
  } else throw new Error("Not implemented.");
```

Since component functions can now be asynchronous, let's add an `await` in there:

```
    // ...
    const returnedJsx = await Component(props);
    // ...
```

This means `renderJSXToHTML` itself would now have to be an `async` function now, and calls to it will need to be `await` ed.

```
async function renderJSXToHTML(jsx)  {
  // ...
}
```

With this change, any component in the tree can be `async`, and the resulting HTML "waits" for them to resolve.

Look at how much clearer the logic has become as a result:

Open this example in a sandbox.

> Note that this implementation is not ideal because each `await` is "blocking". For example, we can't even *start* sending the HTML until *all* of it has been generated. Ideally, we'd want to *stream* the server payload as it's being generated. This is more complex, and we won't do it in this part of the walkthrough — for now we'll just focus on the data flow. However, it's important to note that we can add streaming later without any changes to the components themselves. Each component only uses `await` to wait for its own *data* (which is unavoidable), but parent components don't need to `await` their children — even when children are `async`. This is why React can stream parent components' output before their children finish rendering.

Step 5: Let's preserve state on navigation

So far, our server can only render a route to an HTML string:

```
async function sendHTML(res, jsx) {
  const html = await renderJSXToHTML(jsx);
  res.writeHead(200, { "Content-Type": "text/html" });
  res.end(html);
}
```

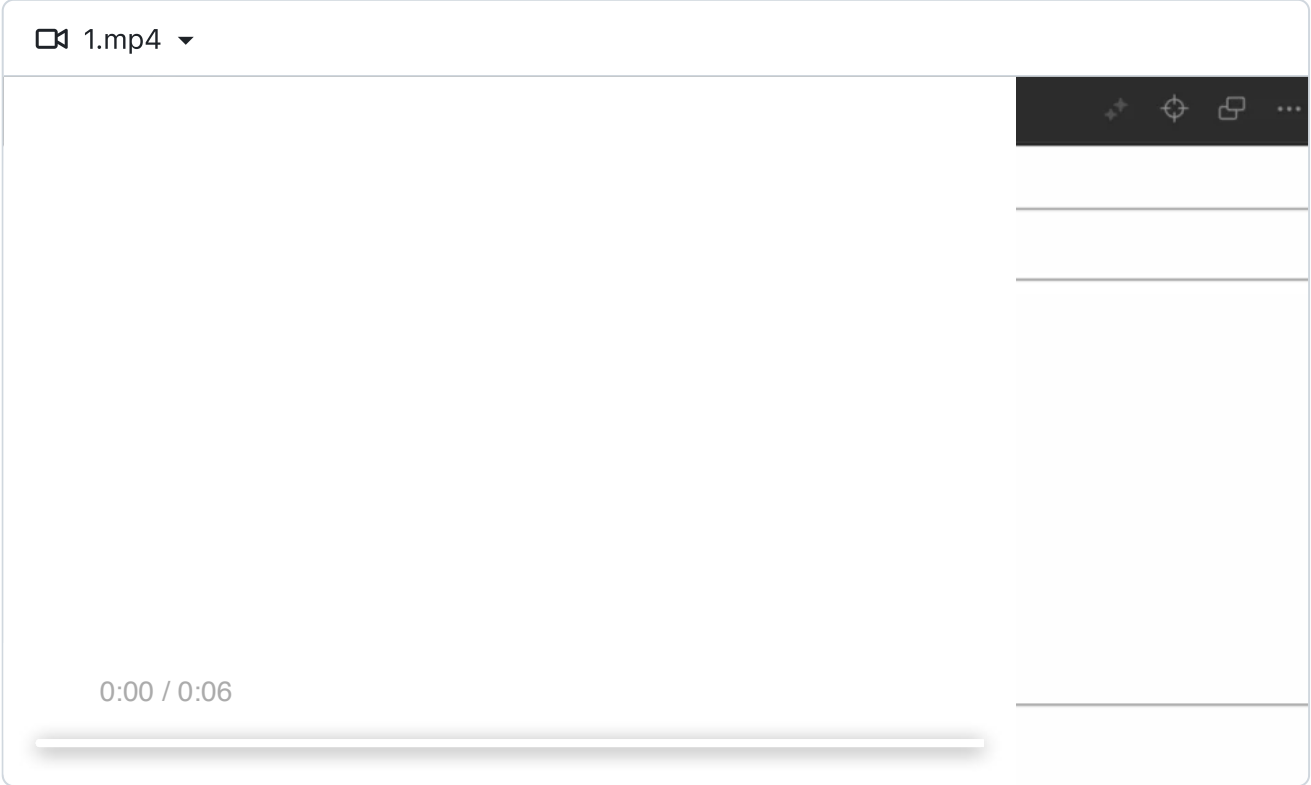
This is great for the first load — the browser is optimized to show HTML as quickly as possible — but it's not ideal for navigations. **We'd like to be able to update "just the parts that changed" *in-place*, preserving the client-side state both inside and around them (e.g. an input, a video, a popup, etc).** This will also let mutations (e.g. adding a comment to a blog post) feel fluid.

To illustrate the problem, let's add an `<input />` to the `<nav>` inside the `BlogLayout` component JSX:

```
<nav>
  <a href="/">Home</a>
  <hr />
```

```
    <input />
    <hr />
  </nav>
```

Notice how the state of the input gets "blown away" every time you navigate around the blog:



This might be OK for a simple blog, but if you want to be able to build more interactive apps, at some point this behavior becomes a dealbreaker. You want to let the user navigate around the app without constantly losing local state.

We're going to fix this in three steps:

1. Add some client-side JS logic to intercept navigations (so we can refetch content manually without reloading the page).
2. Teach our server to serve JSX over the wire instead of HTML for subsequent navigations.
3. Teach the client to apply JSX updates without destroying the DOM (hint: we'll use React for that part).

Step 5.1: Let's intercept navigations

We're gonna need some client-side logic, so we'll add a `<script>` tag for a new file called `client.js`. In this file, we'll override the default behavior for navigations within the site so that they call our own function called `navigate`:

```
async function navigate(pathname) {
  // TODO
}

window.addEventListener("click", (e) => {
  // Only listen to link clicks.
  if (e.target.tagName !== "A") {
```

```
    return;
  }
  // Ignore "open in a new tab".
  if (e.metaKey || e.ctrlKey || e.shiftKey || e.altKey) {
    return;
  }
  // Ignore external URLs.
  const href = e.target.getAttribute("href");
  if (!href.startsWith("/")) {
    return;
  }
  // Prevent the browser from reloading the page but update the URL.
  e.preventDefault();
  window.history.pushState(null, null, href);
  // Call our custom logic.
  navigate(href);
}, true);

window.addEventListener("popstate", () => {
  // When the user presses Back/Forward, call our custom logic too.
  navigate(window.location.pathname);
});
```

In the `navigate` function, we're going to `fetch` the HTML response for the next route, and update the DOM to it:

```
let currentPathname = window.location.pathname;

async function navigate(pathname) {
  currentPathname = pathname;
  // Fetch HTML for the route we're navigating to.
  const response = await fetch(pathname);
  const html = await response.text();

  if (pathname === currentPathname) {
    // Get the part of HTML inside the <body> tag.
    const bodyStartIndex = html.indexOf("<body>") + "<body>".length;
    const bodyEndIndex = html.lastIndexOf("</body>");
    const bodyHTML = html.slice(bodyStartIndex, bodyEndIndex);

    // Replace the content on the page.
    document.body.innerHTML = bodyHTML;
  }
}
```

Open this example in a sandbox.

This code isn't quite production-ready (for example, it doesn't change `document.title` or announce route changes), but it shows that we can successfully override the browser navigation behavior. Currently, we're fetching the HTML for the next route, so the `<input>` state still gets lost. In the next step, we're going to teach our server to serve JSX instead of HTML for navigations. 👀

Step 5.2: Let's send JSX over the wire

Remember our earlier peek at the object tree that JSX produces:

```
{
  $$typeof: Symbol.for("react.element"),
  type: 'html',
  props: {
    children: [
      {
        $$typeof: Symbol.for("react.element"),
        type: 'head',
        props: {
          // ... And so on ...
```

We're going to add a new mode to our server. When the request ends with `?jsx`, we'll send a tree like this instead of HTML. This will make it easy for the client to determine what parts have changed, and only update the DOM where necessary. This will solve our immediate problem of the `<input>` state getting lost on every navigation, but that's not the only reason we are doing this. In the next part (not now!) you will see how this also lets us pass new information (not just HTML) from the server to the client.

To start off, let's change our server code to call a new `sendJSX` function when there's a `?jsx` search param:

```
createServer(async (req, res) => {
  // ...
  try {
    if (url.searchParams.has("jsx")) {
      url.searchParams.delete("jsx"); // Keep the URL passed to the Router clean.
      await sendJSX(res, <Router url={url} />);
    } else {
      await sendHTML(res, <Router url={url} />);
    }
    // ...
```

In `sendJSX`, we'll use `JSON.stringify(jsx)` to turn the object tree above into a JSON string that we can pass down the network:

```
async function sendJSX(res, jsx) {
  const jsxString = JSON.stringify(jsx, null, 2); // Indent with two spaces.
  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(jsxString);
}
```

We'll keep referring to this as "sending JSX", but we're not sending the JSX syntax itself (like `"<Foo />"`) over the wire. We're only taking the object tree produced by JSX, and turning it into a JSON-formatted string. However, the exact transport format will be changing over time (for example, the real RSC implementation uses a different format that we will explore later in this series).

Let's change the client code to see what passes through the network:

```
async function navigate(pathname) {
  currentPathname = pathname;
  const response = await fetch(pathname + "?jsx");
```

```
    const jsonString = await response.text();
    if (pathname === currentPathname) {
      alert(jsonString);
    }
  }
```

Give this a try. If you load the index `/` page now, and then press a link, you'll see an alert with an object like this:

```
{
  "key": null,
  "ref": null,
  "props": {
    "url": "http://localhost:3000/hello-world"
  },
  // ...
}
```

That's not very useful — we were hoping to get a JSX tree like `<html>...</html>`. What went wrong?

Initially, our JSX looks like this:

```
<Router url="http://localhost:3000/hello-world" />
// {
//   $$typeof: Symbol.for('react.element'),
//   type: Router,
//   props: { url: "http://localhost:3000/hello-world" } },
//    ...
// }
```

It is "too early" to turn this JSX into JSON for the client because we don't know what `Router` wants to render, and `Router` only exists on the server. We need to *call* the `Router` component to find out what JSX we need to send to the client.

If we call the `Router` function with `{ url: "http://localhost:3000/hello-world" } }` as props, we get this piece of JSX:

```
<BlogLayout>
  <BlogIndexPage />
</BlogLayout>
```

Again, it is "too early" to turn this JSX into JSON for the client because we don't know what `BlogLayout` wants to render — and it only exists on the server. We have to call `BlogLayout` too, and find out what JSX it want to pass to the client, and so on.

(An experienced React user might object: can't we send their code to the client so that it can execute them? Hold that thought until the next part of this series! But even that would only work for `BlogLayout` because `BlogIndexPage` calls `fs.readdir`.)

At the end of this process, we end up with a JSX tree that does not reference any server-only code. For example:

```
<html>
  <head>...</head>
  <body>
    <nav>
      <a href="/">Home</a>
      <hr />
    </nav>
    <main>
    <section>
      <h1>Welcome to my blog</h1>
      <div>
        ...
      </div>
    </main>
    <footer>
      <hr />
      <p>
        <i>
          (c) Jae Doe 2003
        </i>
      </p>
    </footer>
  </body>
</html>
```

Now, *that* is the kind of tree that we can pass to `JSON.stringify` and send to the client.

Let's write a function called `renderJSXToClientJSX`. It will take a piece of JSX as an argument, and it will attempt to "resolve" its server-only parts (by calling the corresponding components) until we're only left with JSX that the client can understand.

Structurally, this function is similar to `renderJSXToHTML`, but instead of HTML, it traverses and returns objects:

```
async function renderJSXToClientJSX(jsx) {
  if (
    typeof jsx === "string" ||
    typeof jsx === "number" ||
    typeof jsx === "boolean" ||
    jsx == null
  ) {
    // Don't need to do anything special with these types.
    return jsx;
  } else if (Array.isArray(jsx)) {
    // Process each item in an array.
    return Promise.all(jsx.map((child) => renderJSXToClientJSX(child)));
  } else if (jsx != null && typeof jsx === "object") {
    if (jsx.$$typeof === Symbol.for("react.element")) {
      if (typeof jsx.type === "string") {
        // This is a component like <div />.
        // Go over its props to make sure they can be turned into JSON.
        return {
          ...jsx,
          props: await renderJSXToClientJSX(jsx.props),
        };
      } else if (typeof jsx.type === "function") {
```

```
        // This is a custom React component (like <Footer />).
        // Call its function, and repeat the procedure for the JSX it returns.
        const Component = jsx.type;
        const props = jsx.props;
        const returnedJsx = await Component(props);
        return renderJSXToClientJSX(returnedJsx);
      } else throw new Error("Not implemented.");
    } else {
      // This is an arbitrary object (for example, props, or something inside of them).
      // Go over every value inside, and process it too in case there's some JSX in it.
      return Object.fromEntries(
        await Promise.all(
          Object.entries(jsx).map(async ([propName, value]) => [
            propName,
            await renderJSXToClientJSX(value),
          ])
        )
      );
    }
  } else throw new Error("Not implemented");
}
```

Next, let's edit `sendJSX` to turn JSX like `<Router />` into "client JSX" first before stringifying it:

```
async function sendJSX(res, jsx) {
  const clientJSX = await renderJSXToClientJSX(jsx);
  const clientJSXString = JSON.stringify(clientJSX, null, 2);
  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(clientJSXString);
}
```

Open this example in a sandbox.

Now clicking on a link shows an alert with a tree that looks similar to HTML — which means we're ready to try diffing it!

> Note: For now, our goal is to get something working, but there's a lot left to be desired in the implementation. The format itself is very verbose and repetitive, so the real RSC uses a more compact format. As with HTML generation earlier, it's bad that the entire response is being `await`ed at once. Ideally, we want to be able to stream JSX in chunks as they become available, and piece them together on the client. It's also unfortunate that we're resending parts of the shared layout (like `<html>` and `<nav>`) when we know for a fact that they have not changed. While it's important to have the *ability* to refresh the entire screen in-place, navigations within a single layout should not ideally refetch that layout by default. **A production-ready RSC implementation doesn't suffer from these flaws, but we will embrace them for now to keep the code easier to digest.**

Step 5.3: Let's apply JSX updates on the client

Strictly saying, we don't have to use React to diff JSX. So far, our JSX nodes *only* contain built-in browser components like `<nav>`, `<footer>`. You could start with a library that doesn't have a concept of client-side components at all, and use it to diff and apply the JSX updates. However, we'll want to allow rich interactivity later on, so we will be using React from the start.

Our app is server-rendered to HTML, so the first thing we'll want to do is to hydrate that HTML.

If you're not familiar with the concept of hydration, it goes like this. We want to manage the contents of our DOM node with React. When we want to render different JSX, React needs to find the correct DOM nodes to delete or modify. So when the page loads with an existing DOM, React needs to "match up" the initial HTML and the initial JSX first. Then it will have a good idea of which DOM node corresponds to which place in the JSX tree. This might sound superfluous, but as we add more features (for example, asynchronous streaming of parts of the page), it becomes pretty difficult to locate each node reliably without knowing what the initial JSX was. (However, note that with traditional hydration, the code of your components is always available at the client, so all you need to pass is a top-level component like `<App />`. From that point, React will call the components recursively to "learn" what the JSX is supposed to be.) This process is called "hydration" because it's like water flowing into a tree and waking it up from sleep. It can update now.

If you're familiar with the concept of hydration, you might be a little confused — don't you need to pass a root component like `<App />` to `hydrateRoot(document, <App />)`? Indeed, right now we don't have a root component like `<App />` on the client at all! From the client's perspective, currently our entire app is one big chunk of JSX with exactly *zero* React components in it.

What we *can* do is to hydrate it with the client JSX tree directly. Instead of `<App />` you'd have a tree like `<html>...</html>`:

```
import { hydrateRoot } from 'react-dom/client';

const root = hydrateRoot(document, getInitialClientJSX());

function getInitialClientJSX() {
  // TODO: return the <html>...</html> client JSX tree mathching the initial HTML
}
```

This would be extremely fast because right now, there are no components in the client JSX tree at all. React would walk the DOM tree and JSX tree in a near-instant, and build its internal data structure that's necessary to update that tree later on.

Then, whenever the user navigates, we'd fetch the JSX for the next page and update the DOM with `root.render`:

```
async function navigate(pathname) {
  currentPathname = pathname;
  const clientJSX = await fetchClientJSX(pathname);
  if (pathname === currentPathname) {
    root.render(clientJSX);
  }
}

async function fetchClientJSX(pathname) {
  // TODO: fetch and return the <html>...</html> client JSX tree for the next route
}
```

This will achieve what we wanted — it will update the DOM in the same way React normally does, without destroying the state.

Now let's figure out how to implement these two functions.

Step 5.3.1: Let's fetch JSX from the server

We'll start with `fetchClientJSX` because it is easier to implement.

First, let's recall how our `?jsx` server endpoint works:

```
async function sendJSX(res, jsx) {
  const clientJSX = await renderJSXToClientJSX(jsx);
  const clientJSXString = JSON.stringify(clientJSX, null, 2);
  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(clientJSXString);
}
```

On the client, we're going to call this endpoint, and then feed the response to `JSON.parse` to turn it back into JSX:

```
async function fetchClientJSX(pathname) {
  const response = await fetch(pathname + "?jsx");
  const clientJSXString = await response.text();
  const clientJSX = JSON.parse(clientJSXString);
  return clientJSX;
}
```

If you try this implementation, you'll see an error whenever you click a link and attempt to render the fetched JSX:

```
Objects are not valid as a React child (found: object with keys {type, key, ref,
props, _owner, _store}).
```

Here's why. The object we're passing to `JSON.stringify` looks like this:

```
{
  $$typeof: Symbol.for("react.element"),
  type: 'html',
  props: {
    // ...
```

However, if you look at the `JSON.parse` result on the client, the `$$typeof` property seems to be lost in transit:

```
{
  type: 'html',
  props: {
    // ...
```

Without `$$typeof: Symbol.for("react.element")`, React on the client will refuse to recognize it as a valid JSX node.

This is an intentional security mechanism. By default, React refuses to treat arbitrary JSON objects fetched from the network as JSX tags. The trick is that a Symbol value like `Symbol.for('react.element')` doesn't "survive" JSON serialization, and gets stripped out by `JSON.stringify`. That protects your app from rendering JSX that wasn't directly created by your app's code.

However, we *did* actually create these JSX nodes (on the server) and *do* want to render them on the client. So we need to adjust our logic to "carry over" the `$$typeof: Symbol.for("react.element")` property despite it not being JSON-serializable.

Luckily, this is not too difficult to fix. `JSON.stringify` accepts a replacer function which lets us customize how the JSON is generated. On the server, we're going to substutute `Symbol.for('react.element')` with a special string like `"$RE"`:

```
async function sendJSX(res, jsx) {
  // ...
  const clientJSXString = JSON.stringify(clientJSX, stringifyJSX); // Notice the second a
  // ...
}

function stringifyJSX(key, value) {
  if (value === Symbol.for("react.element")) {
    // We can't pass a symbol, so pass our magic string instead.
    return "$RE"; // Could be arbitrary. I picked RE for React Element.
  } else if (typeof value === "string" && value.startsWith("$")) {
    // To avoid clashes, prepend an extra $ to any string already starting with $.
    return "$" + value;
  } else {
    return value;
  }
}
```

On the client, we'll pass a reviver function to `JSON.parse` to replace `"$RE"` back with `Symbol.for('react.element')`:

```
async function fetchClientJSX(pathname) {
  // ...
  const clientJSX = JSON.parse(clientJSXString, parseJSX); // Notice the second argument
  // ...
}

function parseJSX(key, value) {
  if (value === "$RE") {
    // This is our special marker we added on the server.
    // Restore the Symbol to tell React that this is valid JSX.
    return Symbol.for("react.element");
  } else if (typeof value === "string" && value.startsWith("$$")) {
    // This is a string starting with $. Remove the extra $ added by the server.
    return value.slice(1);
  } else {
    return value;
  }
}
```

Open this example in a sandbox.

Now you can navigate between the pages again — but the updates are fetched as JSX and applied on the client!

If you type into the input and then click a link, you'll notice the `<input>` state is preserved on all navigations except the very first one. This is because we haven't told React what the initial JSX for the page is, and so it can't attach to the server HTML properly.

Step 5.3.2: Let's inline the initial JSX into the HTML

We still have this bit of code:

```
const root = hydrateRoot(document, getInitialClientJSX());

function getInitialClientJSX() {
  return null; // TODO
}
```

We need to hydrate the root with the initial client JSX, but where do we get that JSX on the client?

Our page is server-rendered to HTML; however, for further navigations we need to tell React what the initial JSX for the page was. In some cases, it might be possible to partially reconstruct from the HTML, but not always—especially when we start adding interactive features in the next part of this series. We also don't want to *fetch* it since it would create an unnecessary waterfall.

In traditional SSR with React, you also encounter a similar problem, but for data. You need to have the data for the page so that components can hydrate and return their initial JSX. In our case, there are no components on the page so far (at least, none that run in the browser), so nothing needs to run — but there is also no code on the client that knows how to generate that initial JSX.

To solve this, we're going to assume that the string with the initial JSX is available as a global variable on the client:

```
const root = hydrateRoot(document, getInitialClientJSX());

function getInitialClientJSX() {
  const clientJSX = JSON.parse(window.__INITIAL_CLIENT_JSX_STRING__, reviveJSX);
  return clientJSX;
}
```

On the server, we will modify the `sendHTML` function to *also* render our app to client JSX, and inline it at the end of HTML:

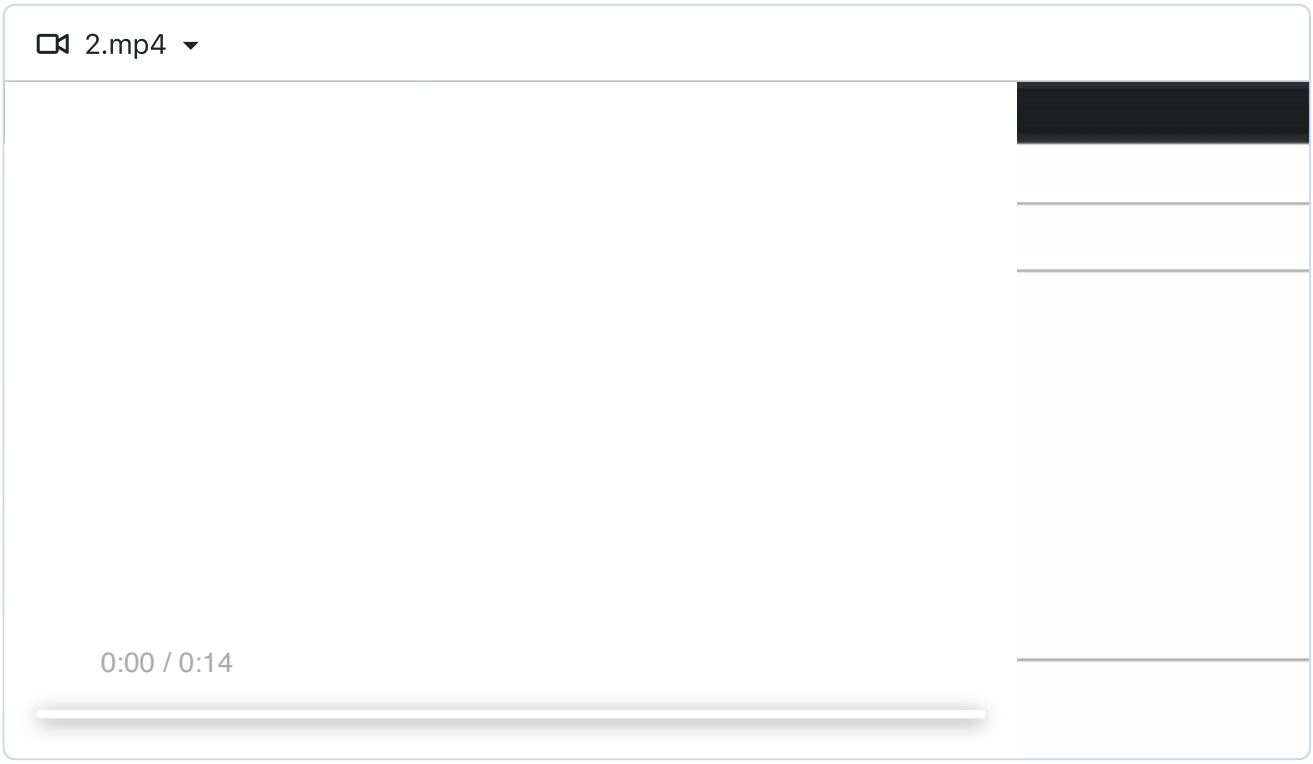
```
async function sendHTML(res, jsx) {
  let html = await renderJSXToHTML(jsx);

  // Serialize the JSX payload after the HTML to avoid blocking paint:
  const clientJSX = await renderJSXToClientJSX(jsx);
  const clientJSXString = JSON.stringify(clientJSX, stringifyJSX, 2);
  html += `<script>window.__INITIAL_CLIENT_JSX_STRING__ = `;
  html += JSON.stringify(clientJSXString).replace(/</g, "\\u003c"); // Escape the string
```

```
    html += `</script>`;
    // ...
```

Finally, we need a few small adjustments to how we generate HTML for text nodes so that React can hydrate them.

Open this example in a sandbox.

Now you can type into an input, and its state is no longer lost between navigations:



That's the goal we originally set out to accomplish! Of course, preserving the state of this particular input isn't the point—the important part is that our app can now refresh and navigate "in-place" on any page, and not worry about destroying any state.

> Note: Although a real RSC implementation *does* encode the JSX in the HTML payload, there are a few important differences. A production-ready RSC setup sends JSX chunks as they're being produced instead of a single large blob at the end. When React loads, hydration can start immediately—React starts traversing the tree using the JSX chunks that are already available instead of waiting for all of them to arrive. RSC also lets you mark some components as *Client* components, which means they *still* get SSR'd into HTML, but their code *is* included in the bundle. For Client components, only JSON of their props gets serialized. In the future, React may add extra mechanisms to deduplicate content between HTML and the embedded payload.

Step 6: Let's clean things up

Now that our code actually *works*, we're going to move the architecture a tiny bit closer to the real RSC. We're still not going to implement complex mechanisms like streaming yet, but we'll fix a few flaws and prepare for the next wave of features.

Step 6.1: Let's avoid duplicating work

Have another look at how we're producing the initial HTML:

```
async function sendHTML(res, jsx) {
  // We need to turn <Router /> into "<html>...</html>" (a string):
  let html = await renderJSXToHTML(jsx);

  // We *also* need to turn <Router /> into <html>...</html> (an object):
  const clientJSX = await renderJSXToClientJSX(jsx);
```

Suppose `jsx` here is `<Router url="https://localhost:3000" />`.

First, we call `renderJSXToHTML`, which will call `Router` and other components recursively as it creates an HTML string. But we also need to send the initial client JSX—so call `renderJSXToClientJSX` right after, which *again* calls the `Router` and all other components. We're calling every component twice! Not only is this slow, it's also potentially incorrect — for example, if we were rendering a `Feed` component, we could get different outputs from these functions. We need to rethink how the data flows.

What if we generated the client JSX tree *first*?

```
async function sendHTML(res, jsx) {
  // Let's turn <Router /> into <html>...</html> (an object) first:
  const clientJSX = await renderJSXToClientJSX(jsx);

  // ...
```

By this point, all our components have executed. Then, let's generate HTML from *that* tree:

```
async function sendHTML(res, jsx) {
  const clientJSX = await renderJSXToClientJSX(jsx);

  // Turn that <html>...</html> into "<html>...</html>" (a string):
  let html = await renderJSXToHTML(clientJSX);
```

Now components are only called once per request, as they should be.

Open this example in a sandbox.

Step 6.2: Let's use React to render HTML

Initially, we needed a custom `renderJSXToHTML` implementation so that we could control how it executes our components. For example, we've need to add support for `async` functions to it. But now that we pass a precomputed client JSX tree to it, there is no point to maintaining a custom implementation. Let's delete it, and use React's built-in `renderToString` instead:

```
import { renderToString } from 'react-dom/server';

// ...

async function sendHTML(res, jsx) {
```

```
const clientJSX = await renderJSXToClientJSX(jsx);
let html = renderToString(clientJSX);
// ...
```

Open this example in a sandbox.

Notice a parallel with the client code. Even though we've implemented new features (like `async` components), we're still able to use existing React APIs like `renderToString` or `hydrateRoot`. It's just that the way we use them is different.

In a traditional server-rendered React app, you'd call `renderToString` and `hydrateRoot` with your root `<App />` component. But in our approach, we first evaluate the "server" JSX tree using `renderJSXToClientJSX`, and pass its *output* to the React APIs.

In a traditional server-rendered React app, components execute in the same way *both* on the server and the client. But in our approach, components like `Router`, `BlogIndexPage` and `Footer` are effectively server-*only* (at least, for now).

As far as `renderToString` and `hydrateRoot` are concerned, it's pretty much as if `Router`, `BlogIndexPage` and `Footer` have never existed in the first place. By then, they have already "melted away" from the tree, leaving behind only their output.

Step 6.3: Let's split the server in two

In the previous step, we've decoupled running components from generating HTML:

- First, `renderJSXToClientJSX` runs our components to produce client JSX.
- Then, React's `renderToString` turns that client JSX into HTML.

Since these steps are independent, they don't have to be done in the same process or even on the same machine.
To demonstrate this, we're going split `server.js` into two files:

- `server/rsc.js`: This server will run our components. It always outputs JSX — no HTML. If our components were accessing a database, it would make sense to run this server close to the data center so that the latency is low.
- `server/ssr.js`: This server will generate HTML. It can live on the "edge", generating HTML and serving static assets.

We'll run them both in parallel in our `package.json`:

```
"scripts": {
  "start": "concurrently \"npm run start:ssr\" \"npm run start:rsc\"",
  "start:rsc": "nodemon -- --experimental-loader ./node-jsx-loader.js ./server/rsc.js",
  "start:ssr": "nodemon -- --experimental-loader ./node-jsx-loader.js ./server/ssr.js"
},
```

In this example, they'll be on the same machine, but you could host them separately.

The RSC server is the one that renders our components. It's only capable of serving their JSX output:

```
// server/rsc.js

createServer(async (req, res) => {
  const url = new URL(req.url, `http://${req.headers.host}`);
  try {
    await sendJSX(res, <Router url={url} />);
  } catch (err) {
    console.error(err);
    res.writeHead(err.statusCode ?? 500);
    res.end();
  }
}).listen(8081);

function Router({ url }) {
  // ...
}

// ...
// ... All other components we have so far ...
// ...

async function sendJSX(res, jsx) {
  // ...
}

function stringifyJSX(key, value) {
  // ...
}

async function renderJSXToClientJSX(jsx) {
  // ...
}
```

The other server is the SSR server. The SSR server is the server that our users will hit. It asks the RSC server for JSX, and then either serves that JSX as a string (for navigations between pages), or turns it into HTML (for the initial load):

```
// server/ssr.js

createServer(async (req, res) => {
  const url = new URL(req.url, `http://${req.headers.host}`);
  if (url.pathname === "/client.js") {
    // ...
    return;
  }
  // Get the serialized JSX response from the RSC server
  const response = await fetch("http://127.0.0.1:8081" + url.pathname);
  const clientJSXString = await response.text();
  if (url.searchParams.has("jsx")) {
    // If the user is navigating between pages, send that serialized JSX as is
    res.writeHead(response.status, { "Content-Type": "application/json" });
    res.end(clientJSXString);
  } else {
    // If this is an initial page load, revive the tree and turn it into HTML
    const clientJSX = JSON.parse(clientJSXString, parseJSX);
    let html = renderToString(clientJSX);
    html += `<script>window.__INITIAL_CLIENT_JSX_STRING__ = `;
```

```
      html += JSON.stringify(clientJSXString).replace(/</g, "\\u003c");
      html += `</script>`;
      // ...
      res.writeHead(response.status, { "Content-Type": "text/html" });
      res.end(html);
    }
  }).listen(8080);
```

Open this example in a sandbox.

We're going to keep this separation between RSC and "the rest of the world" (SSR and user machine) throughout this series. Its importance will become clearer in the next parts when we start adding features to both of these worlds, and tying them together.

(Strictly speaking, it is technically possible to run RSC and SSR within the same process, but their module environments would have to be isolated from each other. This is an advanced topic, and is out of scope of this post.)

Recap

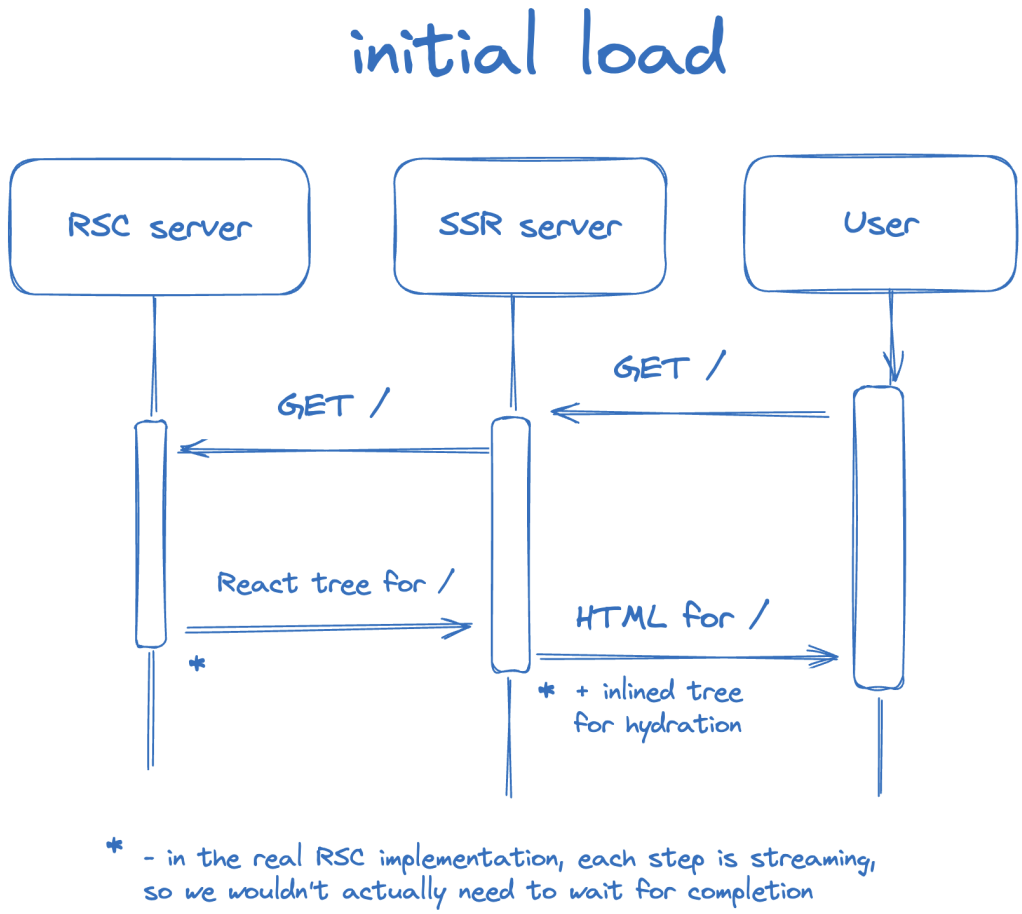
And we're done for today!

It might seem like we've written a lot of code, but we really haven't:

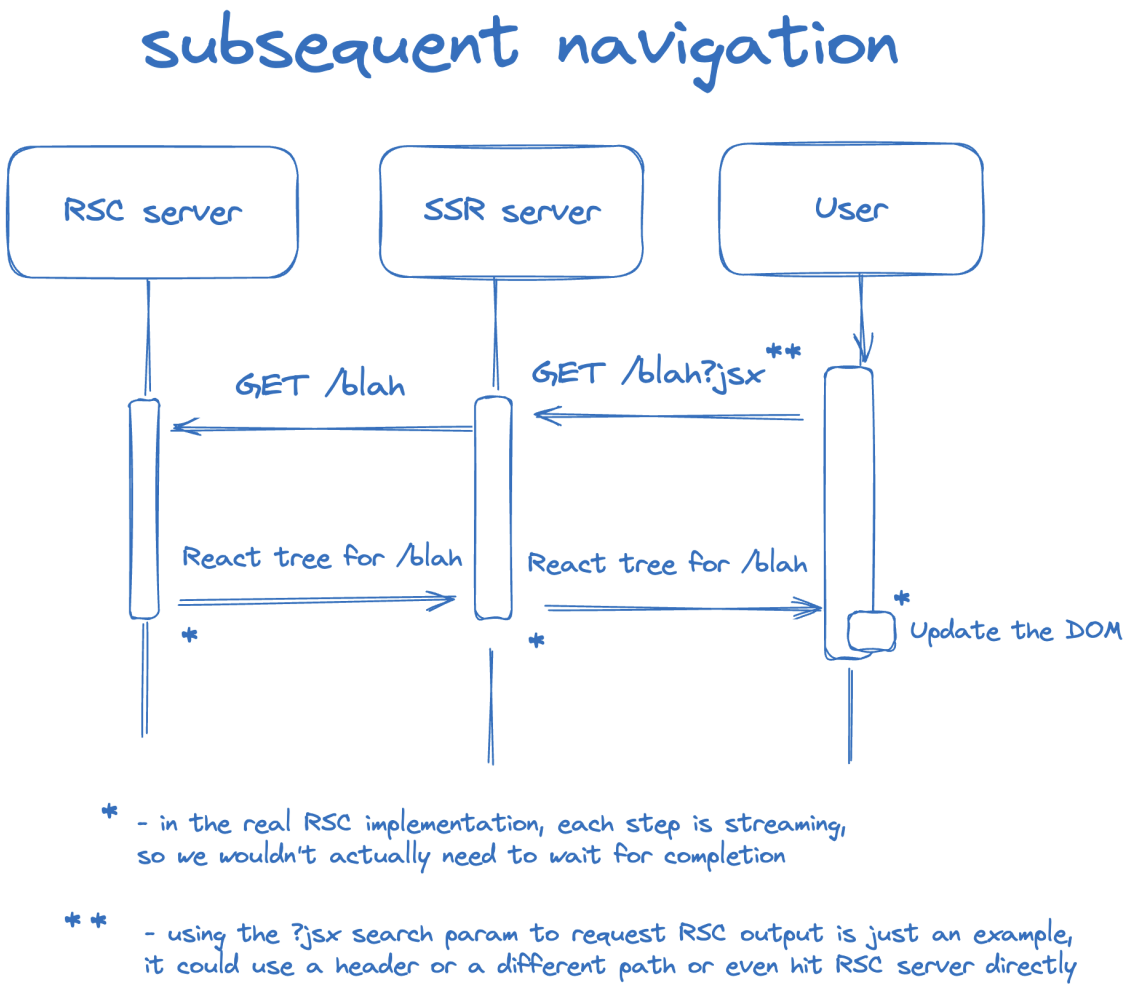
- `server/rsc.js` is 150 lines of code, out of which 80 are our own components.
- `server/ssr.js` is 50 lines of code.
- `client.js` is 60 lines of code.

Have a read through them. To help the data flow "settle" in our minds, let's draw a few diagrams.

Here is what happens during the first page load:



And here is what happens when you navigate between pages:



Finally, let's establish some terminology:

- We will say **React Server** (or just capitalized Server) to mean *only* the RSC server environment. Components that exist only on the RSC server (in this example, that's all our components so far) are called **Server Components**.
- We will say **React Client** (or just capitalized Client) to mean any environment that consumes the React Server output. As you've just seen, SSR is a React Client — and so is the browser. We don't support components on the Client *yet* — we'll build that next! — but it shouldn't be a huge spoiler to say that we will call them **Client Components**.

Challenges

If reading through this post wasn't enough to satisfy your curiosity, why not play with the final code?

Here's a few ideas for things you can try:

- Add a random background color to the `<body>` of the page, and add a transition on the background color. When you navigate between the pages, you should see the background color animating.
- Implement support for fragments (<>) in the RSC renderer. This should only take a couple of lines of code, but you need to figure out where to place them and what they should do.
- Once you do that, change the blog to format the blog posts as Markdown using the `<Markdown>` component from `react-markdown`. Yes, our existing code should be able to handle that!

- The `react-markdown` component supports specifying custom implementations for different tags. For example, you can make your own `Image` component and pass it as `<Markdown components={{ img: Image }}>`. Write an `Image` component that measures the image dimensions (you can use some npm package for that) and automatically emits `width` and `height`.
- Add a comment section to each blog post. Keep comments stored in a JSON file on the disk. You will need to use `<form>` to submit the comments. As an extra challenge, extend the logic in `client.js` to intercept form submissions and prevent reloading the page. Instead, after the form submits, refetch the page JSX so that the comment list updates in-place.
- When you navigate between two different blog posts, their *entire* JSX gets diffed. But this doesn't always make sense — conceptually, these are two *different* posts. For example, if you start typing a comment on one of them, but then press a link, you don't want that comment to be preserved just because the input is in the same location. Can you think of a way to solve this? (Hint: You might want to teach the `Router` component to treat different pages with different URLs as different components by wrapping the `{page}` with something. Then you'd need to ensure this "something" doesn't get lost over the wire.)
- The format to which we serialize JSX is currently very repetitive. Do you have any ideas on how to make it more compact? You can check a production-ready RSC framework like Next.js App Router, or our official non-framework RSC demo for inspiration. Even without implementing streaming, it would be nice to at least represent the JSX elements in a more compact way.
- Imagine you wanted to add support for Client Components to this code. How would you do it? Where would you start?

Have fun!