by

The Two Reacts

January 4, 2024

Suppose I want to display something on your screen. Whether I want to display a web page like this blog post, an interactive web app, or even a native app that you might download from some app store, at least *two* devices must be involved.

Your device and mine.

It starts with some code and data on *my* device. For example, I am editing this blog post as a file on my laptop. If you see it on your screen, it must have already traveled from my device to yours. At some point, somewhere, my code and data turned into the HTML and JavaScript instructing *your* device to display this.

So how does that relate to React? React is a UI programming paradigm that lets me break down *what* to display (a blog post, a signup form, or even a whole app) into independent pieces called *components*, and compose them like LEGO blocks. I'll assume you already know and like components; check react.dev for an intro.

Components are code, and that code has to run somewhere. But wait—*whose* computer should they run on? Should they run on your computer? Or on mine?

Let's make a case for each side.

First, I'll argue that components should run on *your* computer.

Here's a little counter button to demonstrate interactivity. Click it a few times!

```
<Counter />
```

You clicked me 0 times

Assuming the JavaScript code for this component has already loaded, the number will increase. Notice that it increases *instantly on press*. There is no delay. No need to wait for the server. No need to download any additional data.

This is possible because this component's code is running on *your* computer:

```
import { useState } from "react";

export function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button
      className="dark:color-white rounded-lg bg-purple-700 px-2 py-1 font-san
      onClick={() => setCount(count + 1)}
    >
      You clicked me {count} times
    </button>
  );
}
```

Here, `count` is a piece of *client state*—a bit of information in your computer's memory that updates every time you press that button. **I don't know how many times you're going to press the button** so I can't predict and prepare all of its possible outputs on *my* computer. The most I'll dare to prepare on my computer is the *initial* rendering output ("You clicked me 0 times") and send it as HTML. But from that point and on, *your computer had to take over* running this code.

You could argue that it's *still* not necessary to run this code on your computer. Maybe I could have it running on my server instead? Whenever you press the button, your computer could ask my server for the next rendering output. Isn't that how websites worked before all of those client-side JavaScript frameworks?

Asking the server for a fresh UI works well when the user *expects* a little delay—for example, when clicking a link. When the user knows they're navigating to *some different place* in your app, they'll wait. However, any direct manipulation (such as dragging a slider, switching a tab, typing into a post composer, clicking a like button, swiping a card, hovering a menu, dragging a chart, and so on) would feel broken if it didn't reliably provide at least *some* instant feedback.

This principle isn't strictly technical—it's an intuition from the everyday life. For example, you wouldn't expect an elevator button to take you to the next floor in an instant. But when you're pushing a door handle, you *do* expect it to follow your hand's movement directly, or it will feel stuck. In fact, even with an elevator button you'd expect at least *some* instant feedback: it should yield to the pressure of your hand. Then it should light up to acknowledge your press.

When you build a user interface, you need to be able to respond to at least some interactions with *guaranteed* low latency and with *zero* network roundtrips.

You might have seen the React mental model being described as a sort of an equation: *UI is a function of state*, or `UI = f(state)`. This doesn't mean that your UI code has to literally be a single function taking state as an argument; it only means that the current state determines the UI. When the state changes, the UI needs to be recomputed. Since the state "lives" on your computer, the code to compute the UI (your components) must also run on your computer.

Or so this argument goes.

Next, I'll argue the opposite—that components should run on *my* computer.

Here's a preview card for a different post from this blog:

```
<PostPreview slug="a-chain-reaction" />
```

A Chain Reaction

2452 words

How does a component from *this* page know the number of words on *that* page?

If you check the Network tab, you'll see no extra requests. I'm not downloading that entire blog post from GitHub just to count the number of words in it. I'm not embedding the contents of that blog post on this page either. I'm not calling any APIs to count the words. And I sure did not count all those words by myself.

So how does this component work?

```
import { readFile } from "fs/promises";
import matter from "gray-matter";

export async function PostPreview({ slug }) {
  const fileContent = await readFile("./public/" + slug + "/index.md", "utf8"
  const { data, content } = matter(fileContent);
  const wordCount = content.split(" ").filter(Boolean).length;

  return (
    <section className="rounded-md bg-black/5 p-2">
      <h5 className="font-bold">
        <a href={"/" + slug} target="_blank">
          {data.title}
        </a>
      </h5>
      <i>{wordCount} words</i>
    </section>
  );
}
```

This component runs on *my* computer. When I want to read a file, I read a file with `fs.readFile`. When I want to parse its Markdown header, I parse it with `gray-matter`. When I want to count the words, I split its text and count them.

There is nothing extra I need to do because my code runs *right where the data is*.

Suppose I wanted to list *all* the posts on my blog along with their word counts.

Easy:

```
<PostList />
```

A Chain Reaction

2452 words

A Complete Guide to useEffect

9913 words

Algebraic Effects for the Rest of Us

3062 words

Before You memo()

856 words

All I needed to do was to render a `<PostPreview />` for every post folder:

```
import { readdir } from "fs/promises";
import { PostPreview } from "./post-preview";

export async function PostList() {
  const entries = await readdir("./public/", { withFileTypes: true });
  const dirs = entries.filter(entry => entry.isDirectory());
  return (
    <div className="mb-4 flex h-72 flex-col gap-2 overflow-scroll font-sans">
      {dirs.map(dir => (
        <PostPreview key={dir.name} slug={dir.name} />
      ))}
    </div>
  );
}
```

None of this code needed to run on your computer—and indeed *it couldn't* because your computer doesn't have my files. Let's check *when* this code ran:

```
<p className="text-purple-500 font-bold">
  {new Date().toString()}
</p>
```

Fri Jan 05 2024 00:50:25 GMT+0000 (Coordinated Universal Time)

Aha—that's exactly when I last deployed my blog to my static web hosting! My components ran during the build process so they had full access to my posts.

Running my components close to their data source lets them read their own data and preprocess it *before* sending any of that information to your device.

By the time you loaded this page, there was no more `<PostList>` and no more `<PostPreview>`, no `fileContent` and no `dirs`, no `fs` and no `gray-matter`. Instead, there was only a `<div>` with a few `<section>`s with `<a>`s and `<i>`s inside each of them. Your device only received *the UI it actually needs to display* (the rendered post titles, link URLs, and post word counts) rather than *the full raw data* that your components used to compute that UI from (the actual posts).

With this mental model, *the UI is a function of server data*, or `UI = f(data)`. That data only exists *my* device, so that's where the components should run.

Or so the argument goes.

UI is made of components, but we argued for two very different visions:

- `UI = f(state)` where `state` is client-side, and `f` runs on the client. This approach allows writing instantly interactive components like `<Counter />`. (Here, `f` may *also* run on the server with the initial state to generate HTML.)

- `UI = f(data)` where `data` is server-side, and `f` runs on the server only. This approach allows writing data-processing components like `<PostPreview />`. (Here, `f` runs categorically on the server only. Build-time counts as "server".)

  If we set aside the familiarity bias, both of these approaches are compelling at what they do best. Unfortunately, these visions *seem* mutually incompatible.

  If we want to allow instant interactivity like needed by `<Counter />`, we *have to* run components on the client. But components like `<PostPreview />` can't run on the client *in principle* because they use server-only APIs like `readFile`. (That's their whole point! Otherwise we might as well run them on the client.)

  Okay, what if we run all components on the server instead? But on the server, components like `<Counter />` can only render their *initial* state. The server doesn't know their *current* state, and passing that state between the server and the client is too slow (unless it's tiny like a URL) and not even always possible (e.g. my blog's server code only runs on deploy so you can't "pass" stuff to it).

  Again, it seems like we have to choose between two different Reacts:

- The "client" `UI = f(state)` paradigm that lets us write `<Counter />`.
- The "server" `UI = f(data)` paradigm that lets us write `<PostPreview />`.

  But in practice, the real "formula" is closer to `UI = f(data, state)`. If you had no `data` or no `state`, it would generalize to those cases. But ideally, I'd prefer my programming paradigm to be able to *handle both cases* without having to pick another abstraction, and I know at least a few of you would like that too.

  The problem to solve, then, is how to split our "`f`" across two very different programming environments. Is that even possible? Recall we're not talking about some actual function called `f`—here, `f` represents all our components.

  Is there some way we could split components between your computer and mine in a way that preserves what's great about React? Could we combine and nest components from two different environments? How would that work?

How *should* that work?

Give it some thought, and next time we’ll compare our notes.
