READING TIME • 13 MIN

A Love Letter to React

Annie Ruygt

We're Fly.io. React, Phoenix, Rails, whatever you use: we put your code into lightweight microVMs on our own hardware in 26 cities and counting. Check us out—your app can be running close to your users within minutes.

It's hard to overstate the impact React has had since its release in 2013. For me, React came with a few revelations. First was a reactive HTML-aware component model for building UIs. Second was colocated markup *directly in the app code*. Third, it focused on efficiency in a world where SPAs were increasingly heavy-handed.

It was also something that I could grok in a weekend.

My previous attempts at drive-by learning other reactive frameworks of the day were not so successful. Phoenix borrowed a lot from React when we first shipped LiveView in 2018, but only recently have we gone all-in with an HTML-aware component system in Phoenix 1.7.

It's hard to imagine building applications any other way. What follows is a heartfelt homage to React's impact on the frontend and backend world, and how Phoenix landed where it did thanks to the revelations React brought almost ten years ago.

Reactive Component System

With LiveView, I was inspired by React components and their beautifully simple programming model. A component is an object that defines a render function, and returns some HTML (or in later versions, a function that renders HTML). That function makes use of component state, and whenever a state change occurs the render function is called again.

```
class Example extends React.Component {
  render(){
    return (
      <div>
        <p>You clicked {this.state.count}
times</p>
        <button onClick={() =>
this.setState({count: this.state.count +
1})}>
          Click me
        </button>
      </div>
    )
  }

  constructor(props){
    super(props)
    this.state = {count: 0}
  }
}
```

This was such a simple model to understand when coming into React for the first time. Here we have a React component with some state, and a `render` function. The function returns some HTML, and calls `setState` when a button is clicked. Any time `setState` is called, React will call `render` again, and the UI updates. Easy peasy.

We borrowed from this with LiveView by taking that model and slapping it on the server in a stateful process:

```
defmodule DemoWeb.CounterLive do
  use Phoenix.LiveView

  def render(assigns) do
    ~H"""
    <div>
      <p>You clicked <%= @count %>
times</p>
      <button phx-click="inc">Click
me</button>
    </div>
    """
  end

  def mount(_, _, socket), do: {:ok,
assign(socket, count: 0)}

  def handle_event("inc"), do: {:noreply,
update(socket, :count, &(&1 + 1))}
end
```

I've talked previously about what this kind of programming model on the server enables, like the fact we don't write bespoke routes, controllers, and serializers, or JSON APIs or GraphQL endpoints. But here we're just appreciating how easy this model is to understand. In a world of ever-increasing framework complexity, React's take on interactive applications was a breath of fresh air that we were quick to borrow.

Another choice React made was also extremely contentious at the time: putting HTML right in with your app code. People hated it. But React was right.

JSX: A Colocation Revelation

Like many folks ten years ago, you might still be thinking "HTML in your app code?! Are we back to the 2000's PHP days of mixing code and markup together in a file? And we call this progress?"

These kind of takes were common. They also missed the point. Unlike the PHP days of yore, React applications weren't a string concatenation of app code, HTML, and business logic masquerading as a web application.

React's JSX templates place the most coupled pieces of UI together: the markup and stateful code supporting that markup. Think about it: you have a bunch of variables (state) in your app code that are also needed in your template code for UI rendering or behavior. You also have a bunch of UI interactions in your templates that make it back into app code—like button clicks. These two things are necessarily tightly coupled. Change either side of the contract and the other side breaks. So React made the wise step to put those tightly coupled things together.

This brings us to a lesson React taught me that I later carried over to Phoenix: if two pieces of code can only exist together, they should live together. Or to think about it another way, if two pieces of code must *change* together, they must live together.

There's no guesswork on what happens if I change some LiveView state or LiveView template variables because they live in the same file. I also don't have to search throughout the codebase to find which coupled-but-distant template file needs to be added or changed to accommodate the code I'm writing.

Now, there are times where it's not practical to write app code and markup in a single file. Sometimes template reuse or a large document means it makes more sense to have a separate template. In these cases, you want the next best thing: colocated files. In general, the tightly coupled parts of your application should be as close as practically possible. If not the same file, then the same directory, and so on.

HTML-aware Components as Extensible Building Blocks

React also popularized HTML-aware components with their JSX template system. On top of writing HTML in your component's app code, you *call* components from markup in an HTML tag-like way.

This is more than a cute way to make function calls. It's also not something I appreciated right away. The advantage of this approach is a natural composition of static HTML tags alongside dynamic components and logic. Large HTML structures quickly lose their shape when mixing dynamic code and reusable UI with tags—an issue with Ruby or Elixir-like templating engines.

For example, imagine you need to render a collection of items, then within that collection, conditionally call some other template code. With Rails or older Phoenix style `<%=`

`%>` templates, the markup structure almost entirely gets lost in the mess of branches:

```
<h1><%= @title %></h1>
<table class="border border-gray-100
rounded-lg">
  <thead>
    <%= for {_field, label} <- @fields do
%>
      <th><%= label %></th>
    <% end %>
  </thead>
  <tbody>
    <%= for row <- @rows do %>
      <tr>
        <%= for {{field, label}, i} <-
Enum.with_index(@fields) do %>
          <td>
            <%= if i == 0 do %>
              <div class="text-bold">
                <%= row[field] %>
              </div>
            <% else %>
              <div class="text-center p-
4">
                <%= row[field] %>
              </div>
            <% end %>
          </td>
        <% end %>
      </tr>
    <% end %>
  </tbody>
</table>
```

This has a few problems. First, the markup structure is completely lost when mixing code branches and comprehensions.

```
                <\/div>
              <% end %>
            </td>
          <% end %>
        </tr>
      <% end %>
    </tbody>
```

This makes template editing a brittle and frustrating experience. If our goal is to dynamically build markup, why does the markup structure get lost in the mix? It gets worse when we try to encapsulate this table into a reusable piece of UI. The best we could do prior to adopting React's approach is bespoke functions or templates that hide the entire table from the caller:

```
def table(assigns) do
  ~H"""
  <h1><%= @title %></h1>
  <table class="border border-gray-100
rounded-lg">
    ...
  </table>
  """
end
```

Then the caller can render the component:

```
<%= table(title: "Users", rows: @users,
fields: [name: "Name", bio: "Bio"]) %>
```

This works, but extensible UI components are all but impossible. The moment we want to customize one aspect of the table, we need to write another template like `user_table` which slightly alters the cells or adds more actionable links to another cell, and so on. If we tried to make it extensible without an HTML-aware component primitive, we'd end up with something like:

```
<%= table(title: "Users", rows: @users,
fields: [name: "Name, bio: "Bio"]) do %>
  <%= for row <- @rows do %>
    <tr>
      <%= for {field, label} <- @fields
do %>
        <%= user_cell(user: row, field:
field, label: label) %>
      <% end %>
      <td class="actions">
        <a href="..." data-
method="post">Confirm User</a>
        <a href="..." data-
method="post">Ban User</a>
      </td>
    </tr>
  <% end %>
<% end %>
```

Our bespoke functions now mask the HTML structure, which makes it difficult to figure out what's happening. We also can't easily encapsulate table row and cell styling.

Worse, we prevent the caller from passing their own arbitrary block content to our components.

For example, imagine instead of a string "Users" as the table title, the caller wanted to render HTML within the `<h1>`, such as a subtitle, icon, or even another component? With template engines that only do string concatenation, passing strings around prohibits all of this. A caller may try passing a string of HTML instead, but it's a nonstarter:

```
<%= table(title: """
  Listing <em>Users</em>
  #{icon(name: "avatar")}
  """,
  rows: @users, fields: [name: "Name,
bio: "Bio"])
do %>
  <%= for row <- @rows do %>
    <tr>
      <%= for {field, label} <- @fields
do %>
        <%= user_cell(user: row, field:
field, label: label) %>
      <% end %>
      <td class="actions">
        <a href="..." data-
method="post">Confirm User</a>
        <a href="..." data-
method="post">Ban User</a>
      </td>
    </tr>
  <% end %>
<% end %>
```

Passing strings around for arbitrary content quickly breaks down. It's not only terrible to write, but the user would have to forgo HTML escaping and carefully inject user-input into their dynamic strings. That's a no-go.

React's JSX showed us a better way. If we make our templating engine HTML-aware and component calls become tag-like expressions, we solve the readability issues. Next, we can allow the caller to provide their own arbitrary markup as arguments.

React allows passing markup as an inner component block, or as a regular argument ("prop" in React parlance) to the component. For example, in React, one could write:

```
<Table
  rows={users}
  title={
    <h1>Listing <em>Users</em></h1>
  }
/>
```

Later frameworks like Vue, and the Web Component spec itself standardized and expanded this concept with the "slot" terminology.

In Phoenix, HTML syntax for components along with slots turns our mess of mixed HTML tags and strings into this beautifully extensible UI component:

```
<div>
  <.table rows={@users}>
    <:title>Listing <em>Users</em>
</:title>
    <:col :let={user} label="Name"><%=
user.name %></:col>
    <:col :let={user} label="Bio"><%=
user.bio %></:col>
  </.table>
</div>
```

The Phoenix HEEx template engine supports calling components external to the current scope in a similar React style, such as `<Table.simple>...</Table.simple>`. Phoenix also allows calling imported function components directly with the `<.component_name />` notation.

In the table example above, we call the `table` function with arguments passed in a tag-like attribute syntax, just like in React props. Next, the table accepts an internal block of arbitrary markup, and we here we can make use of slots to pass `title` and `col` information.

The neat thing about slots in Phoenix is the fact that they are collection based. The caller can provide an arbitrary number of entries, such as in our `<:col>` example. To render a table, internally the `table` component can simply iterate over the `cols` we passed for each `row`, and "yield" back to us the individual user resources. You can see this in action via the `:let={user}` syntax in the col entries.

The internal table can also iterate over the `cols` to build the `<th>`s for the table head. What results is far more pleasant to write than pure HTML and can be extended by the caller without bespoke functions. The function component and slot primitives allow us to encapsulate everything about building tables in our UI in a single place.

Like React, you'll find that your Phoenix applications establish a surprisingly small set of core UI building blocks that you can use throughout your application.

Phoenix screams on Elixir

Phoenix screams on Fly.io.

Fly.io was practically born to run Phoenix. With super-clean built-in private networking for clustering and global edge deployment, LiveView apps feel like native apps anywhere in the world.

Deploy your Phoenix app in minutes. →

Efficient at Its Core

My SPA trials and tribulations began before React entered this world. I've gone from jQuery spaghetti, Backbone.js, Angular 1, Angular 2, Ember, and finally React. React provided just the right amount of structure, while being quick to pick up and get going with. It was also super fast.

React really pushed the industry forward with their virtual DOM features. Instead of replacing large parts of the browser's DOM with a freshly rendered template on any little change, React kept a "virtual" DOM as a datastructure that it was able to compute diffs against. This allowed React to compute the minimal set of concrete DOM operations required to update the browser when state changes occur.

This was groundbreaking at the time.

Other SPA frameworks quickly followed suit with their own optimizations. Server-side frameworks are a different paradigm entirely, but they can learn a lot from React's innovation. Phoenix certainly did.

For Phoenix, we borrowed these ideas, but we have this pesky layer between the app and the client, known as the network. Our problem set is quite different from React, but if you squint, you can see all the same inspirations and approaches we took in Phoenix LiveView's optimizations.

For example, on the server we only want to execute the parts of the template that changed rather than the entire template. Otherwise we're wasting CPU cycles. Likewise, we only want to send the dynamic parts of the template that changed down the wire instead of the entire thing to limit latency and bandwidth. While we don't keep a virtual DOM on the server, we do keep track of the static and dynamic parts of the HEEx templates. This allows us to do efficient diff-based rendering on the server and send down minimal diffs to the client. Meanwhile, the client uses morphdom to apply only the minimal patches necessary on the client.

The end result is this: a state change occurs in the LiveView component tree, a diff of changes is computed on the server with noops where possible, and the minimal diff of changes is sent down the wire. On the client, we take those changes and apply them via a minimal set of DOM operations to efficiently update the UI. Sound familiar?

React's Influence on the Backend

React changed the front-end game when it was released, and its ideas have trickled up to the backend world. And no, I don't mean React Server Components (but React is also trickling up to the server too!). Outside of Phoenix, you'll find other backend frameworks now ship with their own HTML-aware component system, such as Laravel's Blade templates in the PHP space.

If you're a backend framework in 2022 and not shipping an HTML-aware engine, it's time to follow React's lead. I can't imagine Phoenix not landing where we did, and my only regret is we didn't follow React sooner. Thank you React for paving the way! ❤

LAST UPDATED • NOV 2, 2022

Chris McCord
@chris_mccord

Previous post ↓

Logbook: October 21 to 28, 2022