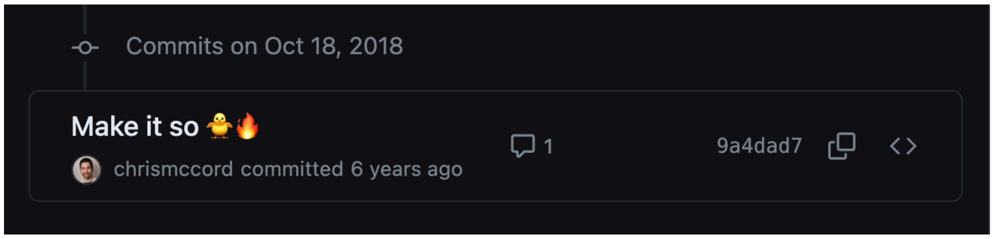
Phoenix LiveView 1.0.0 is here!

Posted on December 3rd, 2024 by Chris McCord

LiveView 1.0.0 is out!

This 1.0 milestone comes six years after the first LiveView commit.


Commits on Oct 18, 2018

Make it so 🐥🔥
chrismccord committed 6 years ago · 1 · 9a4dad7

Why LiveView

I started LiveView to scratch an itch. I wanted to create dynamic server-rendered applications without writing JavaScript. I was tired of the inevitable ballooning complexity that it brings.

Think realtime form validations, updating the quantity in a shopping cart, or real-time streaming updates. Why does it require moving mountains to solve in a traditional stack? We write the HTTP glue or GraphQL schemas and resolvers, then we figure out which validation logic needs shared or dup'd. It goes on and on from there – how do we get localization information to the client? What data serializers do we need? How do we wire up WebSockets and IPC back to our code? Is our js bundle getting too large? I guess it's time to start turning the Webpack or Parcel knobs. Wait Vite is a thing now? Or I guess Bun configuration is what we want? We've all felt this pain.

The idea was, what if we removed these problems entirely? HTTP can go away, and the server can handle all the rendering and

Search

Search for... Search

Tags

CI
ML
bumbleebee
channels
elixirconf
generators
guides
liveview
nx
phoenix
pubsub
releases
testing
uploads

Recent Posts

Phoenix LiveView 1.0.0 is here!
Phoenix LiveView 0.19 released
Phoenix 1.7.2 released

See all

dynamic update concerns. It felt like a heavy approach, but I knew Elixir and Phoenix was perfectly suited for it.

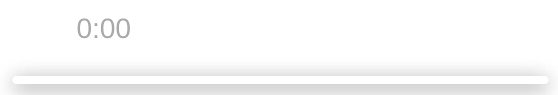
Six years later this programming model still feels like cheating. Everything is super fast. Payloads are tiny. Latency is best-in-class. Not only do you write less code, there’s simply less to think about when writing features.

Real-time foundations unlock superpowers

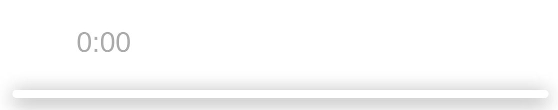
Interesting things happen when you give every user and UI a real-time, bidirectional foundation as a matter of course. You suddenly have superpowers. You almost don’t notice it. Being freed from all the mundane concerns of typical full-stack development lets you focus on just shipping features. And with Elixir, you start shipping features that other platforms can’t even conceive as possible.

Want to ship real-time server logs to the js console in development? No problem!

What about supporting production hot code upgrades where browsers can auto re-render anytime CSS stylesheets, images, or templates change – without losing state or dropping connections? Sure!



Or maybe you have an app deployed planet-wide where you do work across the cluster and aggregate the results in real-time back to the UI. Would you believe the entire LiveView, including the template markup and RPC calls, is 350 LOC?



These are the kinds of applications that LiveView enables. It feels incredible to ship these kinds of things, but it took a while to arrive here for good reasons. There was a lot to solve to make this programming model truly great.

How it started

Conceptually, what I really wanted is something like what we do in React – change some state, our template re-renders automatically, and the UI updates. But instead of a bit of UI running on the client, what if we ran it on the server? The LiveView could look like this:

```
defmodule ThermoLive do
  def render(assigns) do
    ~H"""
    <div id="thermostat">
      <p>Temperature: {@thermostat.temperature}</p>
      <p>Mode: {@thermostat.mode}</p>
      <button phx-click="inc">+</button>
      <button phx-click="dec">-</button>
    </div>
    """
  end

  def mount(%{"id" => id}, _session, socket) do
    thermostat = ThermoControl.get_thermostat!(id)
    :ok = ThermoControl.subscribe(thermostat)
```

```
      {:ok, assign(socket, thermostat: thermostat)}
    end

    def handle_info({ThermoControl, %ThermoStat{} = new_the
      {:noreply, assign(socket, thermostat: new_thermo)}
    end

    def handle_event("inc", _, socket) do
      thermostat = ThermoControl.inc(socket.assigns.thermos
      {:noreply, assign(socket, thermostat: thermostat)}
    end
  end
```

Like React, we have a render function and something that sets our initial state when the LiveView mounts. When state changes, we call render with the new state and the UI is updated.

Interactions like **`phx-click`** on the **`+`** or **`-`** button, can be sent as RPC's from client to server and the server can respond with fresh page HTML. These client/server messages use Phoenix Channels which scale to millions of connections per server.

Likewise, if the server wants to send an update to the client, such as another user changing the thermostat, the client can listen for it and replace the page HTML in the same fashion. My naive first pass on the **`phoenix_live_view.js`** client looked something like this.

```
  let main = document.querySelector("[phx-main]")
  let channel = new socket.channel("lv")
  channel.join().receive("ok", ({html}) => main.innerHTML =
  channel.on("update", ({html}) => main.innerHTML = html)

  window.addEventListener("click", e => {
    let event = e.getAttribute("phx-click")
    if(!event){ return }
    channel.push("event", {event}).receive("ok", ({html}) =
  })
```

This is how LiveView started. We went to the server for interactions, re-rendered the entire template on state change, and sent the entire page down to the client. The client then swapped out the inner HTML.

It worked, but it was not great. Partial state changes required re-executing the entire template and sending down gobs of HTML for otherwise tiny updates.

Still the basic programming model was exactly what I wanted. As HTTP fell away from my concerns, entire layers of full-stack considerations disappeared.

Next the challenge was making this something truly great. Little did we know we'd accidentally our way to outperforming many SPA use-cases along the way.

How we optimized the programming model

LiveView's diffing engine solved two problems with a single mechanism. The first problem was only executing those dynamic parts of a template that actually changed from a previous render. The second was only sending the minimal data necessary to update the client.

It solves both by splitting the template into static and dynamic parts. Considering the following LiveView template:

```
~H"""
<p class={@mode}>Temperature: {format_unit(@temperature)}
"""
```

At compile time, we convert the template into a struct like this:

```
%Phoenix.LiveView.Rendered{
  static: ["<p class=\"", \">Temperature:", "</p>"]
  dynamic: fn assigns ->
    [
      if changed?(assigns, :mode), do: assigns.mode,
      if changed?(assigns, :temperature), do: format_unit
    ]
  end
}
```

We know the static parts never change, so they are split from the dynamic Elixir expressions. Next, we compile each expression with change tracking based on the variables accessed within each expression. On render, we compare the previous template values with the new and only execute the template expression if the value has changed.

Instead of sending the entire template down on change, we can send the client all the static and dynamic parts on `mount`. After mount we only send the partial diff of dynamic values for each update.

To see how this works, we can imagine the following payload being sent on `mount` for the template above:

```
{
  s: ["<p class=\"", ">Temperature: ", "</p>"],
  0: "cooling",
  1: "68℉"
}
```

The client receives a map of static values in the `s` key, and dynamic values keyed by their index in the statics. For the client to render the full template string, it only needs to zips the static list with the dynamic values. For example:

```
["<p class=\"", "cooling", "\">Temperature: ", "68℉", "<
"<p class=\"cooling\">Temperature: 68℉</p>"
```

With the client holding a static/dynamic cache, optimizing network updates is no work at all. Any server render following `mount` simply returns the new dynamic values at their known index. Unchanged dynamic values and statics are ignored entirely.

If a LiveView runs `assign(socket, :temperature, 70)`, the `render/1` function is invoked, and the following payload gets sent down the wire:

```
{1: "70℉"}
```

Thats it! To update the UI, the client simply merges this object with its static/dynamic cache:

```
{                      {
                         s: ["<p class=\"", ">Temperature:
                         0: "cooling",
  1: "70F"     =>        1: "70℉"
}                      }
```

Then the data is zipped together on the client to produce the full HTML of the UI.

Of course `innerHTML` updates blow away UI state and are expensive to perform. So like any client-side framework, we compute minimal DOM diffs to efficiently update the DOM. In fact, we've had folks migrate from React to Phoenix LiveView because LiveView client rendering was faster what their React app could offer.

Optimizations continued from there. Including fingerprinting, for comprehensions, tree sharing, and more. You can read all about each optimization on the Dashbit blog.

We apply these optimizations *automatically and for free* thanks to our stateful client and server connection. Most other server rendered HTML solutions send the whole fragment on every update or require users to fine tune updates by hand.

Best in class latency

We've seen how LiveView payloads are smaller than the best hand-written JSON API or GraphQL query, but it's even better than that. Every LiveView holds a connection to the server so page navigation happens via live navigation. TLS handshakes, current user auth, etc happen a *single time* for the lifetime of the user's visit. This allows page navigation to happen via a single WebSocket frame, and fewer database queries for any client action. The result is fewer round trips from the client, and simply less work done by the server. This provides less latency for the end-user compared to an SPA fetching data or sending mutations up to a server.

Holding a stateful connections comes at the cost of server memory, but it's far cheaper than folks expect. At a baseline, a given channel connection consumes 40kb of memory. This gives a 1GB server a theoretical ceiling of ~25,000 concurrent LiveViews. Of course the more state you store, the more memory you consume, but you only hold onto the state you need. We also have **`` `stream` ``** primitives for handling large collections without impacting memory. Elixir and the Erlang VM were designed for this. Scaling a stateful system to millions of concurrent users isn't theoretical – we do it all the time. See WhatsApp, Discord, or our own benchmarks as examples.

With the programming model optimized on both client and server, we expanded into higher level building blocks that take advantage of our unique diffing engine.

Reusable Components with HEEx

Change tracking and minimal diffs were ground-breaking features, but our HTML templates still lacked composability. The best we could offer is "partial"-like template rendering where a function could encapsulate some partial template content. This works, but it composes poorly and is mismatched in the way we write markup. Fortunately Marlus Saraiva from the Surface project spearheaded development of an HTML-aware component system and contributed back to the LiveView project. With HEEx components, we have a declarative component system, HTML validation, and compile-time checking of component attributes and slots.

HEEx components are just annotated functions. They look like this:

```
@doc """
Renders a button.

## Examples

    <.button>Send!</.button>
    <.button phx-click="go">Send!</.button>
"""
attr :type, :string, default: nil
attr :rest, :global, include: ~w(disabled form name value

slot :inner_block, required: true

def button(assigns) do
  ~H"""
  <button
    type={@type}
    class="rounded-lg bg-zinc-900 hover:bg-zinc-700 py-2
    {@rest}
  >
    {render_slot(@inner_block)}
  </button>
  """
end
```

An invalid call to a component, such as **`<.button click="bad">`** produces a compile-time warning:

```
warning: undefined attribute "click" for component AppWeb
  lib/app_web/live/page_live.ex:123: (file)
```

Slots allows the component to accept arbitrary content from a caller. This allows components to be much more extensible by the caller without creating a bunch of bespoke partial templates to handle every scenario.

Streamlined HEEx syntax

When we introduced HEEx and function components, we added a new syntax for interpolating values within tag attributes along with `:if` and `:for` conveniences for conditionally generating templates. It looked like this:

```
<div :if={@some_condition?}>
  <ul>
    <li :for={val <- @values}>Value <%= val %></li>
  </ul>
</div>
```

Note the use of standard EEx `<%= %>` interpolation. With the release of LiveView 1.0, we are extending the HTML-aware `{}` attribute interpolation syntax to within tag bodies as well. This means you can now interpolate values directly within the tag body in a streamlined syntax:

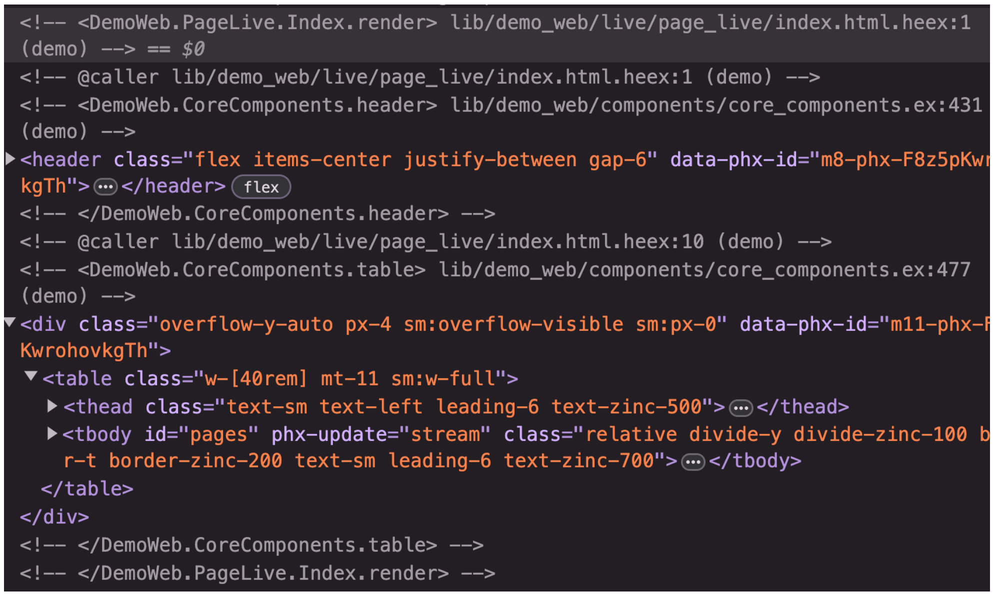
```
<div :if={@some_condition?}>
  <ul>
    <li :for={val <- @values}>Value {val}</li>
  </ul>
</div>
```

The EEx `<%= %>` remains supported and is required for generating dynamic blocks of distinct markup, as well as for interpolating values within `<script>` and `<style>` tags.

HEEx markup annotations

Gone are the days of examining your browser's HTML and then hunting for where that HTML was generated within your code. The final browser markup can be rendered within several nested layers of component calls. How do we quickly trace back who rendered what?

HEEx solves this with a `debug_heex_annotations` configuration. When set, all rendered markup will be annotated with the file:line of the function component definition, *as well as*, the file:line of the caller invocation of the component. In practice your dev HTML will look like this in the browser inspector:



```
<!-- <DemoWeb.PageLive.Index.render> lib/demo_web/live/page_live/index.html.heex:1
(demo) --> == $0
<!-- @caller lib/demo_web/live/page_live/index.html.heex:1 (demo) -->
<!-- <DemoWeb.CoreComponents.header> lib/demo_web/components/core_components.ex:431
(demo) -->
▶<header class="flex items-center justify-between gap-6" data-phx-id="m8-phx-F8z5pKw
kgTh">…</header> flex
<!-- </DemoWeb.CoreComponents.header> -->
<!-- @caller lib/demo_web/live/page_live/index.html.heex:10 (demo) -->
<!-- <DemoWeb.CoreComponents.table> lib/demo_web/components/core_components.ex:477
(demo) -->
▼<div class="overflow-y-auto px-4 sm:overflow-visible sm:px-0" data-phx-id="m11-phx-F
KwrohovkgTh">
  ▼<table class="w-[40rem] mt-11 sm:w-full">
    ▶<thead class="text-sm text-left leading-6 text-zinc-500">…</thead>
    ▶<tbody id="pages" phx-update="stream" class="relative divide-y divide-zinc-100 b
     r-t border-zinc-200 text-sm leading-6 text-zinc-700">…</tbody>
  </table>
</div>
<!-- </DemoWeb.CoreComponents.table> -->
<!-- </DemoWeb.PageLive.Index.render> -->
```

It annotates the document both at the caller site and the function component definition. If you find the above hard to navigate, you can use the new `Phoenix.LiveReloader` features that have your editor jump to an element's nearest caller or definition file:line when clicked with a special key sequence of your choosing.

Let's see it in action:

0:00

First, we can see how holding `c` while clicking jumped to the caller file:Line location for that `<.button>` invocation. Next, we see that holding `d` while clicking the button jumped to the function definition file:line.

This is such a simple quality of life improvement. It will become a key part of your workflow as soon as you try it out.

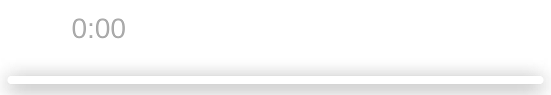
Interactive Uploads

A few years ago, LiveView tackled the file upload problem. Something that should be easy has historically been unnecessarily difficult. We wanted a single abstraction for interactive uploads for both direct to cloud, and direct to server use-cases.

With a few lines of server code you can have file uploads with drag and drop, file progress, selection pruning, file previews, and more.

More recently, we defined an **`` `UploadWriter` ``** behavior. This gives you access to the raw upload stream as it's being chunked by the client. This lets you do things like stream uploads to a different server or transcode a video as it's being uploaded.

Since the uploads happen over the existing LiveView connection, reflecting the upload progress or advanced file operations become trivial to implement:



Streams and Async

Following uploads, we shipped a streams primitive for efficiently handling large collections without needing to hold those collections in server memory. We also introduced **`` `assign_async` ``** and **`` `start_async` ``** primitives, which makes handling async operations and rendering async results a breeze.

For example, imagine you have an expensive operation that calls out to an external service. The results can be latent or spotty, or both. Your LiveView can use **`assign_async/2`** to offload this operation to a new process and **`<.async_result>`** to render the results with each loading, success, or failure state.

```
  def render(assigns) do
    ~H"""
    <.async_result :let={org} assign={@org}>
      <:loading>Loading organization <.spinner /></:loading
      <:failed :let={_failure}>there was an error loading t
      {org.name}
    </.async_result>
    """
  end

  def mount(%{"slug" => slug}, _, socket) do
    {:ok, assign_async(:org, fn -> {:ok, %{org: fetch_org(s
  end
```

Now instead of worrying about an async task crashing the UI, or carefully monitoring async ops while updating the template with a bunch of conditionals, you have a single abstraction for performing the work and rendering the results. As soon as the LiveView disconnects, the async processes are cleaned up, ensuring no wasted resources go to a UI that is no longer around.

Here we can also see slots in action with the **`<:loading>`** and **`<:failed>`** slots of the **`<.async_result>`** function component. Slots allow the caller to extend components with their own dynamic content, including their own markup and function component calls.

LiveView goes mainstream

LiveView and .NET Blazor both started about the same time. I like to think both projects helped spearhead the adoption of this programming model.

Since getting started, this model has been embraced in various ways in the Go, Rust, Java, PHP, JavaScript, Ruby, and Haskell communities. And I'm sure others I haven't yet heard of.

Most don't offer LiveView's declarative model. Instead developers are required to annotate how individual elements are updated and removed, leading to fragile applications, akin to client-side applications before the introduction of React and other declarative frameworks. Most also lack the optimizations LiveView developers get for free. Large payloads are sent on every event unless developers manually fine tune them.

React itself liked the idea of putting React on the server so much, they shipped their own React Server Components to tackle a cross section of similar goals with LiveView. In the case of RSC, pushing real-time events are left to external means.

React, like most, chose different tradeoffs because *they had no choice*. The majority skip the stateful, bidirectional communication layer because most platforms are poorly suited for it. Elixir and the Erlang VM are truly what make this programming model shine. And we have only barely discussed our built-in globally distributed clustering and PubSub. There are truly extraordinary features built into the platform that are at your fingertips.

Try it out

Now is a great time to dive in and give LiveView a try! We have launched **`` `new.phoenixframework.org` ``** which lets you get up and running in seconds with Elixir and your first Phoenix project with a single command:

For osx/linux:

```
$ curl https://new.phoenixframework.org/myapp | sh
```

For Windows PowerShell:

```
  > curl.exe -fsSO https://new.phoenixframework.org/app.bat
```

For existing applications, check the changelog for breaking changes to bring your existing apps up to speed.

What's Next

Following this release, we'll be continuing efforts around collocated JavaScript hooks, enhancing Web Component integration, supporting navigation guards, and more as outlined in our issue tracker.

Special Thanks

Arriving here wouldn't have been possible without the help of the Phoenix team, especially Steffen Deusch, who has tackled countless LiveView issues over the last year.

Happy Hacking!

–Chris