goto;

GOTO
Copenhagen 2023

#GOTOcph

Driving BEAM Adoption with LiveView



HELLO!

I'm Sophie

I'm the co-author of *Programming Phoenix LiveView*,

Co-host of the BeamRad.io podcast,

Staff Engineer at GitHub,

And owner of this dog



A normal dog sleeping in a normal way



What to expect

- We'll talk about common problems in web dev and how LiveView solves them.
- We'll look at some code, so you can see how elegantly LiveView provides these solutions.

Why are you here?

So that I can tell you:

- What LiveView is
- How it works
- Why it will drive BEAM adoption and keep functional programming relevant to web dev.

Something you already know...

You 💜 Functional Programming

You might already 💜 The BEAM

But you might not know about LiveView...

What is LiveView?

LiveView brings the power of functional programming, and the BEAM, to web development

It is a simple, declarative, *functional* web development framework

How does it work?

How LiveView Works

- A live view is a process. It receives events, updates state, and renders that state on a web page.

How LiveView Works

- A live view's state is a functional and immutable Elixir data structure called a socket.

How LiveView Works

- It makes client/server communication simple because it's functional and declarative.

The LiveView life-cycle

Phase 1

The LiveView Life-Cycle

- Your app receives a regular HTTP GET request to render a page.
- A page is rendered.

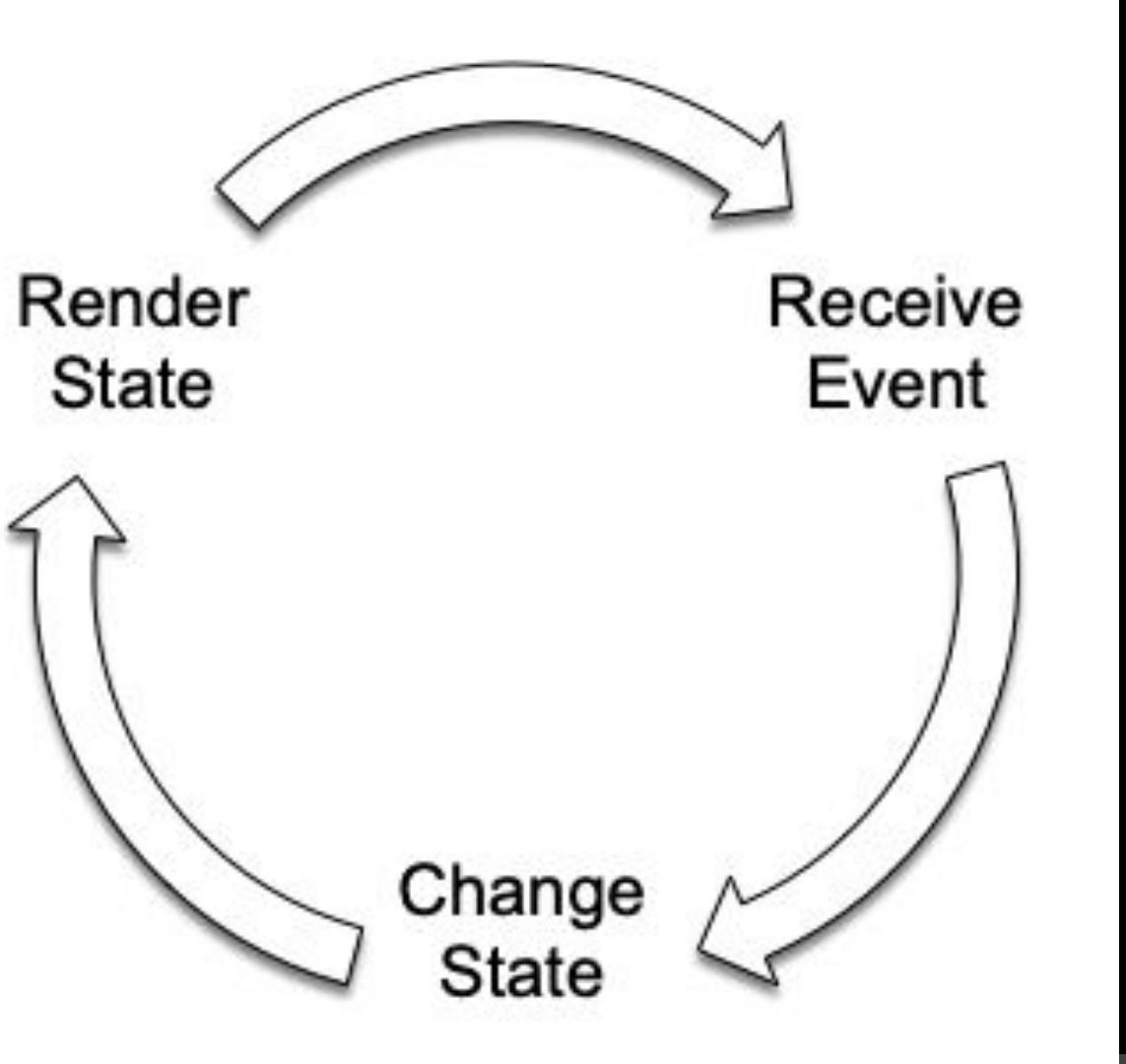
Phase 2

The LiveView Life-Cycle

- LiveView sends a WebSocket request.
- The persistent bi-directional WS connection stays open.

The LiveView Life-Cycle

- The live view receives an event, either from the client or elsewhere in the app.
- The live view updates state.
- The live view re-renders the page *efficiently* with the new state.


Render
State
Receive
Event
Change
State

What's so great about this?

LiveView is Objectively Great*

- Optimizes developer productivity
- Keeps your 🧠 on the server
- Easy to build sophisticated interactive UIs
- Supports real-time features with PubSub
- Brings concurrency + fault-tolerance to web dev

*A proven fact

LiveView is Accessible to Web Devs

- You don’t have to write code that manages concurrency or builds in resiliency.
- You don’t have to worry about mutating state.

LiveView is Accessible to Web Devs

- You don’t have to manage the details of client/server communication.
- You don’t have to write **tedious** JavaScript.

LiveView adoption has been encouraging

LiveView Adoption

- Companies already using Elixir and Phoenix are now using LiveView.
- Consultancies are reaching for LiveView for fast prototyping for their clients.
- LiveView is often used for quick development of responsive internal tooling.

But!

We're just starting to see LiveView drive *Elixir* adoption, and BEAM adoption beyond that

But adoption outside the Elixir community is slow

So, what's standing in the way?

We'll Cover

1. Common LiveView adoption blockers
2. How LiveView knocks them down
3. What's next

🛑 Common Adoption Blockers

(Really just common challenges in web development more generally)

Common blockers

- Security
- State recovery for single page apps
- File uploads
- Code organization
- Integrating JavaScript
- Efficient data transfer

🔒Securing Your Live Views

The Problem

The Problem

LiveView optimizes page load times with live navigation.

The Problem

It does so by re-using the current WS connection when navigating between live views.

The Problem

This means no new HTTP request—we skip the router entirely!

This isn’t just a LiveView problem

The General Problem

How do we secure our app *and* take advantage of the efficiency of WebSockets?

The Solution

The Solution

The LiveView framework provides an API called “Live Sessions”.

The Solution

Live sessions let you share auth logic between a group of live views.

Some code

In your Phoenix router:

```elixir
# the router

scope "/", ArcadeWeb do
  pipe_through [:browser, :require_admin_user]
  live_session :admin, on_mount: {UserAuthLive, :admin} do
    live "/admin-dashboard", Admin.DashboardLive
    # more admin routes
  end
end
```

In the `UserAuthLive` module

```elixir
def on_mount(:admin, params, %{"token" => tok}, socket) do
  socket =
    socket
    |> assign(
        :current_user,
        Accounts.get_user_by_session_token(tok))
  if socket.assigns.current_user.admin do
    {:cont, socket}
  else
    {:halt, redirect(socket, to: "/")}
  end
end
```

```elixir
def on_mount(:admin, params, %{"token" => tok}, socket) do
  socket =
    socket
    |> assign(
         :current_user,
         Accounts.get_user_by_session_token(tok))
  if socket.assigns.current_user.admin do
    {:cont, socket}
  else
    {:halt, redirect(socket, to: "/")}
  end
end
```

```elixir
def on_mount(:admin, params, %{"token" => tok}, socket) do
  socket =
    socket
    |> assign(
         :current_user,
         Accounts.get_user_by_session_token(tok))
  if socket.assigns.current_user.admin do
    {:cont, socket}
  else
    {:halt, redirect(socket, to: "/")}
  end
end
```

The Win

You don’t have to teach LiveView how to share auth logic across live redirects.

The Win

You just have to tell it *what* auth logic to share!



State Recovery

The Problem

The Problem

What happens during a deploy? Or when a live view terminates for any other unexpected reason?

The Problem

Until recently, the state of that live view was lost!

The Problem

This means that the user will see the state of their page clear out—form fields emptied, and other UI state lost.

This isn’t just a LiveView problem

The General Problem

How do we manage persistent state when connection is lost?

The Solution

The Solution

LiveView auto-recovery guarantees form state is restored if the live view disconnects or crashes.

The Solution

When the live view connects, the LiveView JavaScript framework executes any `phx-change` events.

The Solution

You get form recovery for free.

The Solution

If you need more...

The Solution

Use the `phx-auto-recover` binding to trigger a custom event when the live view mounts.

The Solution

Or...

The Solution

Restore live view state from the client

Restore state from the client

Store State in The Client

In local or session storage.

Restore state from the client

Store State in The Client

Pass State as Client Params

In local or session storage.

Initialize the LiveView JS socket with state from local storage.

```
let params = () => {
  return {
    csrf_token: csrfToken,
    restore: sessionStorage.getItem(...)
  }
}

let liveSocket = new LiveSocket("/live", Socket, {params: params})
liveSocket.connect()
```

Restore state from the client

Store State in The Client

In local or session storage.

Pass State as Client Params

Initialize the LiveView socket with state from local storage.

Restore Server State from Params

`mount/3` is called with those params. Use it to establish live view state.

```elixir
def mount(_params, _session, socket) do
  {:ok, socket
        |> assign(get_connect_params(socket)["restore"]}
end
```

The Solution

With **auto-recovery**, the LiveView framework handles 90% of your state recovery needs.

The Solution

With **Custom recovery events** and the **client state restoration** pattern, LiveView exposes the tools you need to cover the last 10%.



File Uploads

The Problem

The Problem

File uploads are an essential part of your average web app.

The Problem

But how can we implement them in LiveView in a sane way without lots of custom code?

This isn’t just a LiveView problem

The General Problem

File uploads are complicated and can often require you to reach for complex custom JavaScript or third-party JS dependencies.

The Solution

The Solution

The LiveView Upload API supports interactive direct-to-server and direct-to-cloud file uploads.

Live Uploads

Add Upload Specification

Specify upload details on mount with `allow_upload/3`

```elixir
def mount(_params, _session, socket) do
  {:ok,
   socket
   |> allow_upload(:cover_image, accept: ~w(.jpg .jpeg))}
end
```

Add Upload Specification

- LiveView tracks uploads as the `@uploads` key in the socket struct.
- LiveView validates uploaded files against your spec.
- You can render errors/progress stored in `@uploads`.

Live Uploads

Add Upload Specification

Specify upload type, size, etc. when the live view mounts.

Add Upload Markup

Give your form an upload field with `live_file_input/2`

```
<form phx-submit="save" phx-change="validate">
  <.live_file_input upload={@uploads.cover_image} />
  <button type="submit">Upload</button>
</form>
```

```
<section phx-drop-target={@uploads.cover_image.ref}>
</section>
```

Add Upload Markup

You can display an image preview.

```
<%= for entry <- @uploads.cover_image.entries do %>
  <figure>
    <.live_img_preview entry={entry} />
    <figcaption><%= entry.client_name %></figcaption>
  </figure>
<% end %>
```

Add Upload Markup

You can display upload progress.

```
<%= for entry <- @uploads.cover_image.entries do %>
  <progress value={entry.progress} max="100">
    <%=entry.progress %>% >
  </progress>
<% end %>
```

Add Upload Markup

You can display upload errors.

```
<%= for entry <- @uploads.cover_image.entries do %>
  <%= for err <- upload_errors(@uploads.cover_image, entry) do %>
    <p class="alert alert-danger"><%= error_to_string(err) %></p>
  <% end %>
<% end %>
```

Live Uploads

Add Upload Specification

Specify upload type, size, etc. when the live view mounts.

Add Upload Markup

Give your form an upload field or drag-and-drop target.

Handle Uploaded Files

Teach your live view how to save uploaded files when the form saves.

```elixir
def handle_event("save", _params, socket) do
  do_consume_uploaded_entries(socket)
  {:noreply, socket}
end
```

```elixir
def do_consume_uploaded_entries(socket) do
  consume_uploaded_entries(socket, :cover_image,fn %{path: path},entry ->
    upload_location = save_upload(path)
    {:ok, upload_location}
  end)
end
```

The Solution

Once again, the LiveView framework takes care of the tedious and complex details and exposes an API to you.

The Solution

You only need to write the code to configure you own upload needs.

Code Organization

The Problem

The Problem

LiveView modules can grow to handle too many events.

LiveView templates encapsulate too much markup and too many responsibilities.

This isn’t just a LiveView problem

The General Problem

How can we compose clean and maintainable code for complex, interactive UIs?

The Solution

The Solution

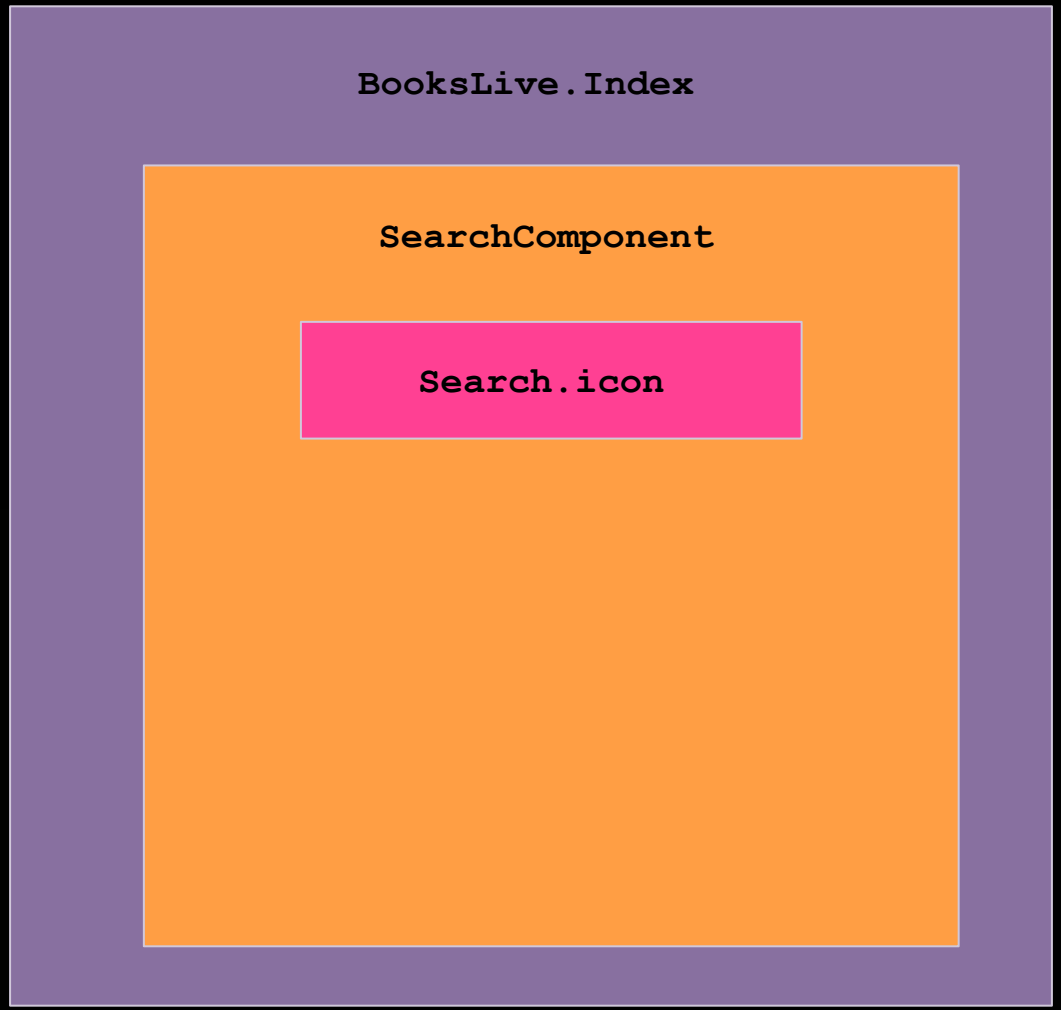
- Wrap up distinct behavior in live components.
- Encapsulate common markup in function components.

The Solution

Use the ergonomic component rendering syntax to compose elegant, layered UIs.

An Example




BooksLive.Index
SearchComponent
Search.icon

Render the `SearchComponent` **live** component from the `Index` template

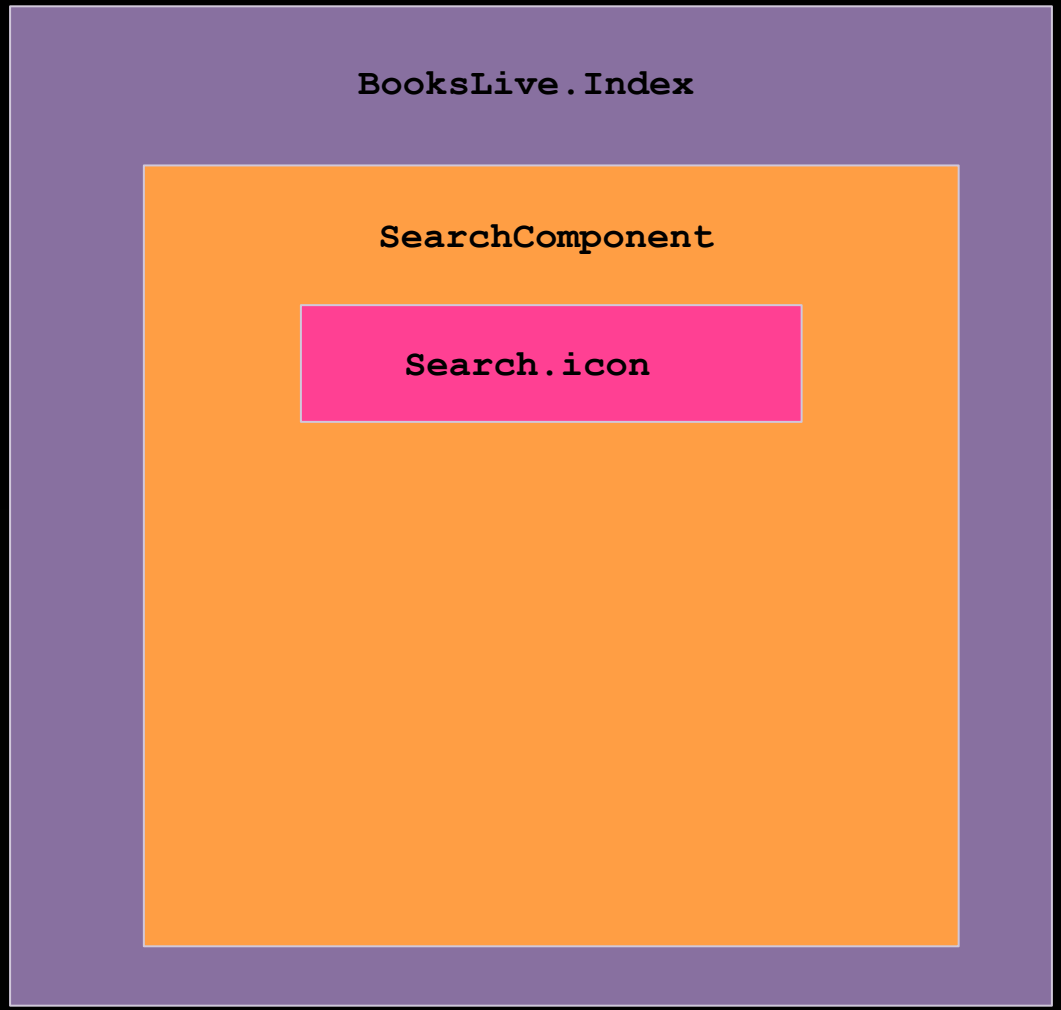
```
<!-- book index template -->

<.live_component module={SearchComponent}
                 id="search"
                 search_class={@search_class}
                 search_changeset={@search_changeset}
                 category_ids={@category_ids} />
```

That live component handles all of the search form events.

- Render the `Search.icon` **function** component from the `SearchComponent` template.

```
<Search.icon />
```


BooksLive.Index
SearchComponent
Search.icon

The Solution

LiveView components give us the tools we need to organize our live views and compose layered UIs that make maintenance a breeze.

The Solution

Function components also let us wrap up common, reusable markup.

The Solution

Function components can render content dynamically using a feature called “component slots”

An Example

A function component for rendering a bootstrap-style card

1. Define a function component that renders some provided content within the card slots.

```elixir
defmodule LiveLibraryWeb.CardComponent do
  use Phoenix.Component
  def card(assigns) do
    ~H"""
     Coming soon…
     """
  end
end
```

```elixir
defmodule LiveLibraryWeb.CardComponent do
  use Phoenix.Component
  def card(assigns) do
    ~H"""
    <div class="card review-card">
      <div class="card-header">
        <%= render_slot(@header, @item) %>
      </div>
      <div class="card-body">
        <h5 class="card-title">
          <strong><%= render_slot(@title, @item) %></strong>
        </h5>
        <p class="card-text">
          <%= render_slot(@body, @item) %>
        </p>
      </div>
    </div>
    """
  end
end
```

2. Call on the component.

```
<CardComponent.card item={@review}>

</CardComponent.cards>
```

```
<CardComponent.card item={@review}>
  <:header let={review}>
    <%= for _ <- 0..review.stars do %>
      <i class="bi bi-star-fill"></i>
    <% end %>
  </:header>

</CardComponent.cards>
```

```
<CardComponent.card item={@review}>
  <:header let={review}>
    <%= for _ <- 0..review.stars do %>
      <i class="bi bi-star-fill"></i>
    <% end %>
  </:header>

  <:title let={review}>
    by: <%= review.user.email %>
  </:title>

</CardComponent.cards>
```

```
<CardComponent.card item={@review}>
  <:header let={review}>
    <%= for _ <- 0..review.stars do %>
      <i class="bi bi-star-fill"></i>
    <% end %>
  </:header>

  <:title let={review}>
    by: <%= review.user.email %>
  </:title>

  <:body let={review}>
    <%= review.notes %>
  </:body>

</CardComponent.cards>
```

Reviews



by: tracy@email.com

Atque voluptas ullam dolorem et ipsam incidunt laborum. Accusantium est minima iure exercitationem ut. Ducimus corrupti nihil doloremque voluptatem accusamus similique eum quasi enim? Nesciunt maxime commodi voluptatem laudantium sed ipsum pariatur! Et quia rem iure officia qui.



by: jenna@email.com

Neque totam debitis consequatur ullam sed nobis voluptates. Qui quo iusto quia numquam exercitationem! Error voluptate quod dolor.

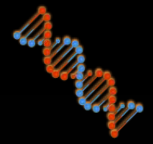


by: liz@email.com

Maiores nihil quod id odio libero ipsum quia! Qui totam at qui beatae possimus!

Component slots allow us to dynamically render different pieces of content into the function component's markup.

As a result, we get an easy-to-use, highly reusable card component that renders beautifully.



Integrating JavaScript

The Problem

The Problem

Integrating custom JavaScript can be challenging and messy.

The Problem

The need for more complex JS can force you away from LiveView.

The Problem

Simple UI interactions like toggling have to round-trip to the server.

This isn’t just a LiveView problem

The General Problem

How can we get the sophisticated, interactive UIs that users need without the complexity of many modern JS frameworks?

The Solution

The Solution

JS Bindings execute utility operations fully on the client.

The Solution

JS bindings provide a JavaScript command interface for LiveView.

The Solution

JS Bindings integrate with JS Hooks to invoke complex JavaScript in response to user actions.

An Example



- The `JS.toggle/1` function toggles the display attribute of the HTML element.

```elixir
def icon(assigns) do
  ~H"""
  <i phx-click={JS.toggle(to: "#book-search-form")} class="bi bi-search"></i>
  """
end
```

- Lots more JS commands available that wrap up common interactions like adding/remove a class, showing/hiding an element, adding temporary transition elements for animation, and more.

- These basic JS bindings let you handle common UI interactions purely on the client, without writing your own JS.

- Again, LiveView wraps up the nitty-gritty details in the framework, and exposes the tools you need to build custom functionality.

- JS commands can do even more.

You can push events to the server in response to user interactions and you can dispatch events to the client for your custom JS hooks to respond to.

- You can use even newer LiveView JS bindings to easily implement infinite scrolling, and much more.

- This scrapes the surface of what you can do with JS Bindings.

  Check out Chris McCord's LiveBeats project for much much more.

📦 Transferring Data Efficiently

The Problem

The Problem

- Managing large datasets in your LiveView app meant storing that data on the server, or else working with the `phx-update="append"` feature.

The Problem

- Storing large datasets server-side has performance implications, and the `phx-update="append"` feature can be cumbersome.

This isn’t just a LiveView problem

The General Problem

How can we efficiently manage large datasets for our users in an interactive web application?

The Solution

The Solution

LiveView streams efficiently manages large datasets in your live views by detaching that data from the socket and letting the client store it instead of the server.

An Example

Establish a large dataset in a stream when the live view initializes

```elixir
def mount(%{“room_id” => id}, _session, socket) do
  {:ok, stream(socket, :messages, Chat.last_ten_messages_for(id))}
end
```

Render the content from the stream

```
<div id="messages" phx-update="stream">
  <div :for={{dom_id, message} <- @streams.messages} id={dom_id}>
    <p>{message.content}</p>
  </div>
</div>
```

```
<div id="messages" phx-update="stream">
  <div :for={{dom_id, message} <- @streams.messages} id={dom_id}>
    <p>{message.content}</p>
  </div>
</div>
```

Now the data is no longer stored server-side

It is embedded in the HTML, and LiveView uses the DOM id to manage it

```
<div id="messages" phx-update="stream">
  <div :for={{dom_id, message} <- @streams.messages} id={dom_id}>
    <p>{message.content}</p>
  </div>
</div>
```

LiveView provides an API for inserts, updates, and deletes on stream data

You call stream functions server-side

LiveView state is updated with some streams instructions

LiveView executes those instructions on the client

The “delete” feature

```
<button phx-change="delete" phx-value={message.id}>
  Delete
</button>
```

```elixir
def handle_event("delete", %{"message_id" => id}, socket) do
  message = Chat.get_message!(message_id)
  Chat.delete_message(message)
  {:noreply, stream_delete(socket, :messages, message)}
end
```

That's it!

Wrap Up



LiveView solves for some of the most common problems in web development, especially when it comes to building interactive, real-time apps.

Things like security, file uploads, state recovery, code organization, JS requirements, and working with large datasets have elegant solutions.

LiveView wraps up the details of the solutions to these common problems, and exposes the levers for developers to implement their own custom functionality.

The declarative LiveView framework handles the hard and tedious parts of interactive web dev. You write the application code you need to meet your users' needs.

LiveView is a compelling solution for building interactive, real-time web apps. Not only for Elixirists, but for anyone working in web development today.

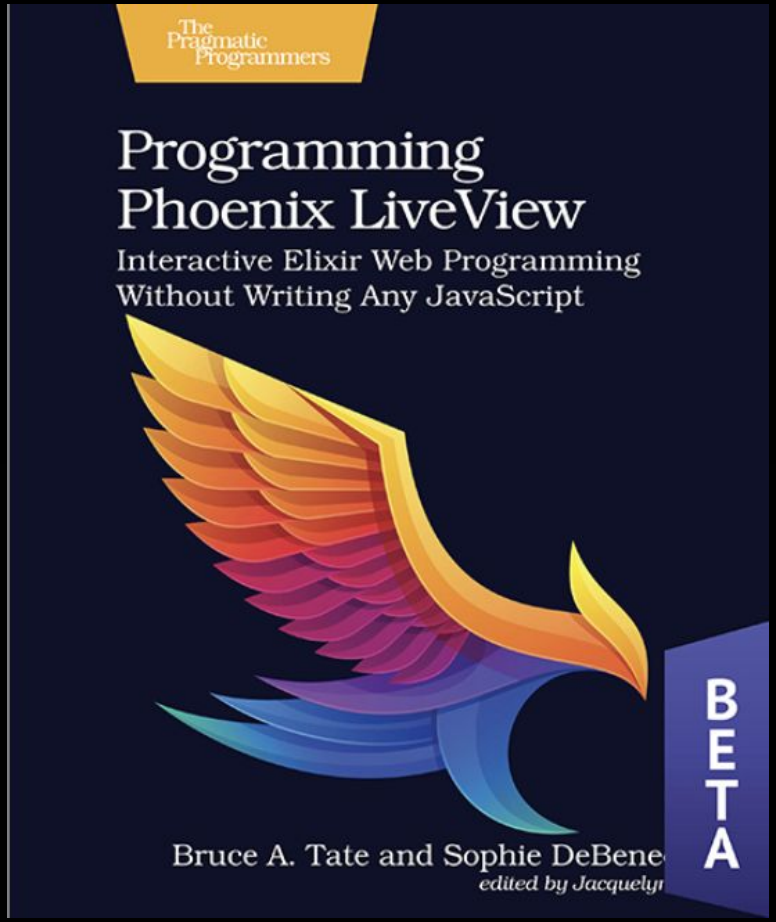
- As LiveView fast approaches 1.0, we can expect accelerated adoption, making The Beam, and functional programming in general, more relevant to web development than ever before.

Thank you!

Want more?
Buy my book!

Use code GOTO_Copenhagen for 50% off

https://bit.ly/programming-liveview



goto;

Don't forget to
rate this session
in the **GOTO Guide app**