goto;

GOTO
Copenhagen 2023

#GOTOcph

LiveView Handles the Hard Parts:

Optimizing WebSocket Communication with LiveView Streams

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

Today

I'll show you:

- How LiveView streams optimize client-server communication over WebSockets
- How to work with streams
- How streams work under the hood

Today

You'll leave with:

- An appreciation for the declarative nature of LiveView
- A solid understanding of where streams fit in LiveView
- The tools you need to dive deeper into LiveView

What is LiveView?

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.

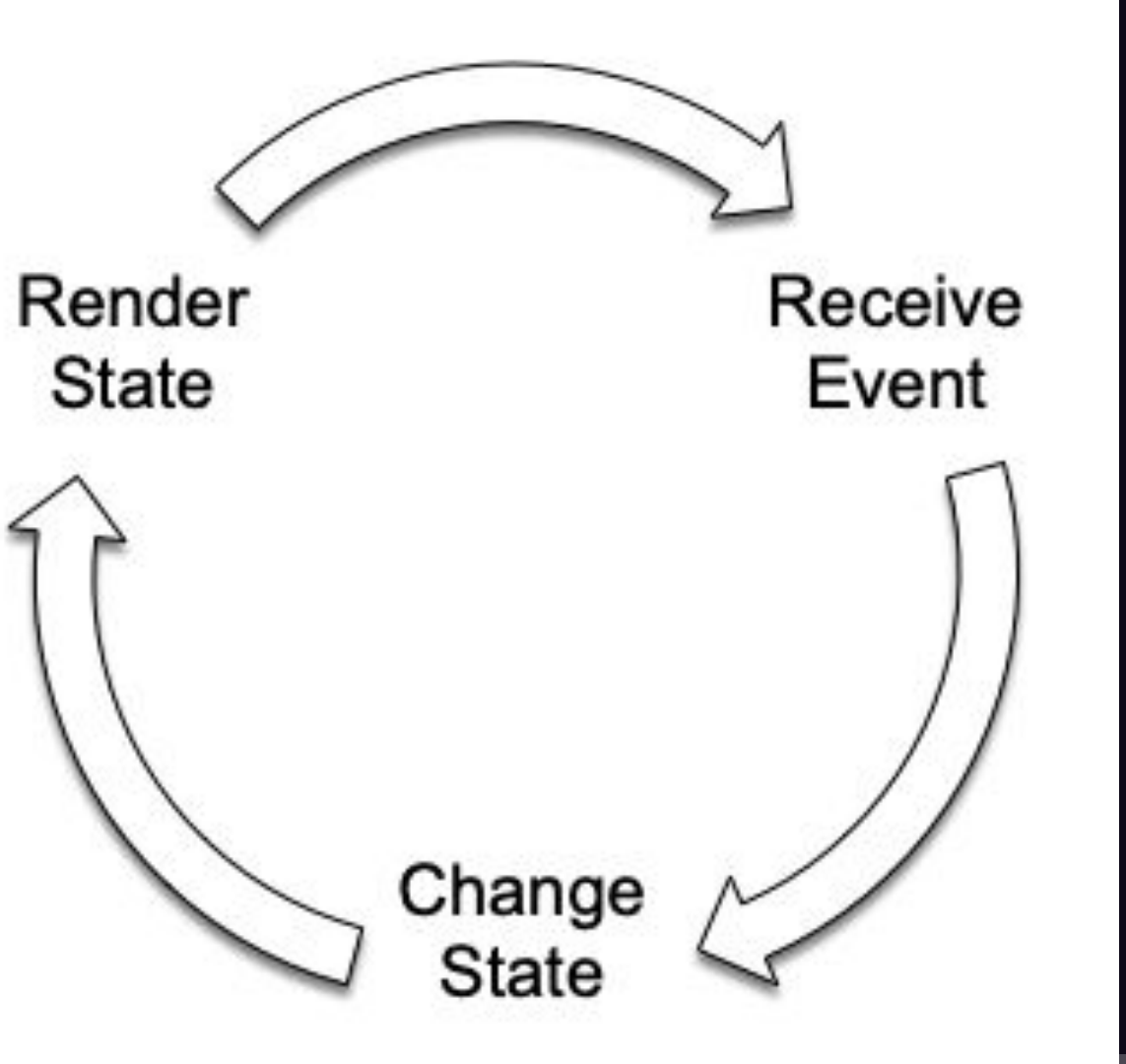
Phase 2

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.
- 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 allows developers to build lightning fast, interactive, real-time web apps with ease.

- This is because LiveView handles the hard parts of client/server communication, OTP process management, and even JavaScript integration and execution.

This leaves you to focus on building the custom logic and interactions that your users require.

You never have to tell LiveView *how* to do something. You only have to tell it *what* to do.

One of the best examples of this is the new **LiveView streams** functionality.

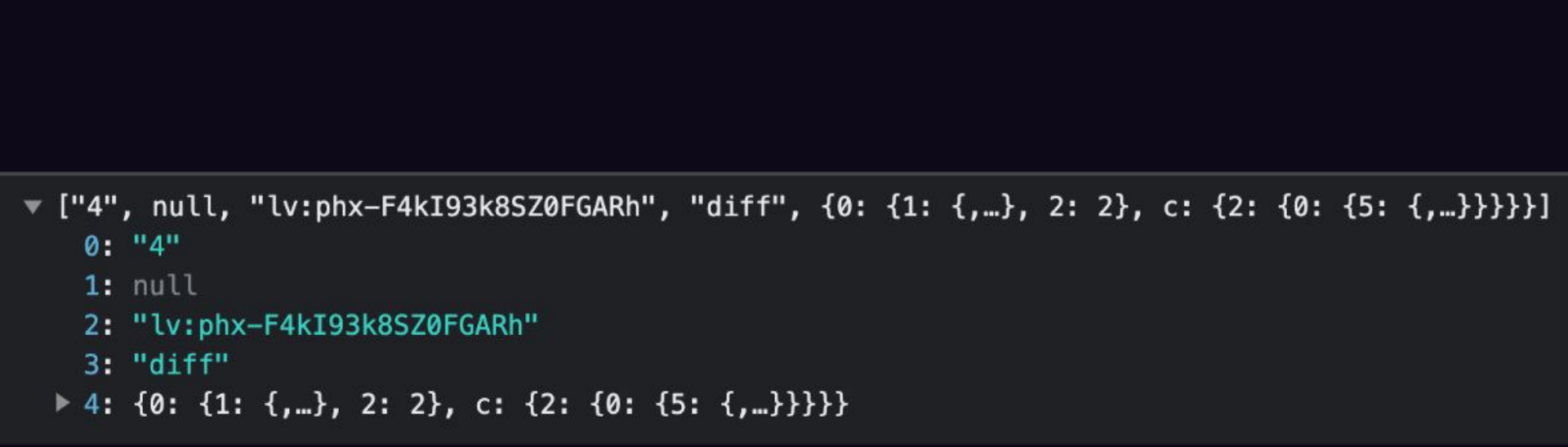
Before we dive into streams, let’s talk about the problem they solve.

The Problem

The Problem

LiveView always aimed to optimize client/server communication by minimizing the amount of data sent down to the client when the page is re-rendered.

- LiveView sends only the small amount of data that describes the page diff, and the LiveView JS code changes only that small part of the page.


["4", null, "lv:phx-F4kI93k8SZ0FGARh", "diff", {0: {1: {,…}, 2: 2}, c: {2: {0: {5: {,…}}}}}]
0: "4"
1: null
2: "lv:phx-F4kI93k8SZ0FGARh"
3: "diff"
4: {0: {1: {,…}, 2: 2}, c: {2: {0: {5: {,…}}}}}

But this isn’t always enough

What about when you're dealing with very large data-sets?

The Problem

Before streams, managing large datasets in your LiveView app meant storing that data on the server, or using the `phx-update="append"` feature.

The Problem

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

The Solution

LiveView Streams

The Solution

LiveView Streams efficiently manage large datasets by detaching that data from the socket and storing it **client-side**.

Next Up...

We'll take a look at how to use streams

And we'll peek under the hood at the streams implementation

What we're building

A chat app

weekend-plans
sre-team

antonia@streamchat.io [2023-03-02T01:27:10]

Voluptatem laborum aut alias voluptatum veritatis pariatur repellat!

virgil@streamchat.io [2023-03-02T01:27:10]

Et voluptatibus sed laudantium?

delpha@streamchat.io [2023-03-02T01:27:10]

Error nihil sequi quia omnis perferendis fugiat!

general@streamchat.io [2023-03-02T01:27:10]

Molestiae laborum aut vitae odio deserunt ad quasi dolore magni!

send

First up

Inserting into the stream

Establishing a stream

We'll use streams to store and render the most recent 10 messages in the chat room UI.

Step 1. Put the stream data in socket assigns

```elixir
def handle_params(%{"id" => id}, _uri, %{assigns: %{live_action: :show}} = socket) do
  {:noreply,
    socket
    |> assign(:room, Chat.get_room!(id))
    |> stream(:messages, Chat.last_ten_messages_for(id))
end
```

Step 2. Render the stream data in the template

```
<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>
```

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

weekend-plans
sre-team

antonia@streamchat.io [2023-03-02T01:27:10]

Voluptatem laborum aut alias voluptatum veritatis pariatur repellat!

virgil@streamchat.io [2023-03-02T01:27:10]

Et voluptatibus sed laudantium?

delpha@streamchat.io [2023-03-02T01:27:10]

Error nihil sequi quia omnis perferendis fugiat!

general@streamchat.io [2023-03-02T01:27:10]

Molestiae laborum aut vitae odio deserunt ad quasi dolore magni!

send

 Under the hood

```
streams: %{
__changed__: MapSet.new([:messages]),
messages: %Phoenix.LiveView.LiveStream{
name: :messages,
dom_id: #Function<3.113057034/1 in Phoenix.LiveView.LiveStream.new/3>,
inserts: [
{"messages-5", -1,
  %StreamChat.Chat.Message{
    __meta__: #Ecto.Schema.Metadata<:loaded, "messages">,
    id: 5,
    content: "Iste cum provident tenetur.",
    room_id: 1,
    sender_id: 8,
    inserted_at: ~N[2023-03-02 01:27:10],
    updated_at: ~N[2023-03-02 01:27:10]
  }},
  # ...
  deletes: []
  }
},
```

socket.assigns

The `stream/4` function

adds a `:streams` key to socket assigns, which in turn points to a map with the provided key.

The `stream/4` function

This contains a `Phoenix.LiveView.LiveStream` struct that holds all of the info the LiveView JS code needs to display your stream data on the page.

```
streams: %{
__changed__: MapSet.new([:messages]),
messages: %Phoenix.LiveView.LiveStream{
name: :messages,
dom_id: #Function<3.113057034/1 in Phoenix.LiveView.LiveStream.new/3>,
inserts: [
{"messages-5", -1,
  %StreamChat.Chat.Message{
    __meta__: #Ecto.Schema.Metadata<:loaded, "messages">,
    id: 5,
    content: "Iste cum provident tenetur.",
    room_id: 1,
    sender_id: 8,
    inserted_at: ~N[2023-03-02 01:27:10],
    updated_at: ~N[2023-03-02 01:27:10]
  }},
  # ...
  deletes: []
  }
},
```

```
streams: %{
__changed__: MapSet.new([:messages]),
messages: %Phoenix.LiveView.LiveStream{
name: :messages,
dom_id: #Function<3.113057034/1 in Phoenix.LiveView.LiveStream.new/3>,
inserts: [
{"messages-5", -1,
  %StreamChat.Chat.Message{
    __meta__: #Ecto.Schema.Metadata<:loaded, "messages">,
    id: 5,
    content: "Iste cum provident tenetur.",
    room_id: 1,
    sender_id: 8,
    inserted_at: ~N[2023-03-02 01:27:10],
    updated_at: ~N[2023-03-02 01:27:10]
  }},
  # ...
  deletes: []
  }
},
```

- The `:inserts` key contains the list of messages we're **inserting** into the initial stream.
- The `:deletes` key contains any messages to **remove** from the template.
- In our template, we access this data via the `@streams.messages` assignment.

```
streams: %{
__changed__: MapSet.new([:messages]),
messages: %Phoenix.LiveView.LiveStream{
name: :messages,
dom_id: #Function<3.113057034/1 in Phoenix.LiveView.LiveStream.new/3>,
inserts: [
{"messages-5", -1,
  %StreamChat.Chat.Message{
    __meta__: #Ecto.Schema.Metadata<:loaded, "messages">,
    id: 5,
    content: "Iste cum provident tenetur.",
    room_id: 1,
    sender_id: 8,
    inserted_at: ~N[2023-03-02 01:27:10],
    updated_at: ~N[2023-03-02 01:27:10]
  }},
  # ...
  deletes: []
  }
},
```

```
streams: %{
__changed__: MapSet.new([:messages]),
messages: %Phoenix.LiveView.LiveStream{
name: :messages,
dom_id: #Function<3.113057034/1 in Phoenix.LiveView.LiveStream.new/3>,
inserts: [
{"messages-5", -1,
  %StreamChat.Chat.Message{
    __meta__: #Ecto.Schema.Metadata<:loaded, "messages">,
    id: 5,
    content: "Iste cum provident tenetur.",
    room_id: 1,
    sender_id: 8,
    inserted_at: ~N[2023-03-02 01:27:10],
    updated_at: ~N[2023-03-02 01:27:10]
  }},
  # ...
  deletes: []
  }
},
```

What's so great about this?

It solves the problem of transferring large data-sets between the client and server for subsequent updates

Because after the initial render, the data is stored client-side ONLY

- After the initial render, the list of messages will no longer be present in the socket under `streams.messages.inserts`.

It will be available only to the LiveView client-side code via the HTML on the page.

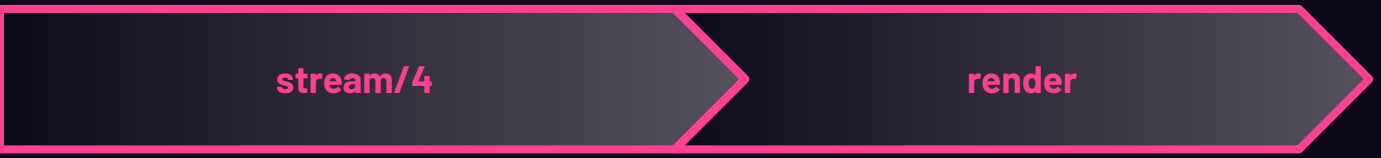
Another nice thing...

You don’t have to tell LiveView how to render the data in streams

Let’s break it down

stream/4

You call stream/4 when the live view is initialized to add data to socket assigns



LiveView renders the template

You call `@streams.messages` in your template, and LiveView renders the data in `socket.assigns.streams.inserts`

This is the declarative nature of LiveView

It will do the same thing for stream data you update or delete

Before we take a look at that...

A closer look at rendering

The data we're rendering is stored in `socket.assigns.streams.messages`.

- This points to a single LiveView Stream **struct**.

But, in the template, we're *iterating* over `@socket.streams.messages`.

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

Streams are enumerable

Let's take a look at how LiveView Streams implement the Enumerable protocol

```elixir
def reduce(%LiveStream{inserts: inserts}, acc, fun) do
  do_reduce(inserts, acc, fun)
end
```

```
[
  {"messages-5", -1,
    %StreamChat.Chat.Message{
      __meta__: #Ecto.Schema.Metadata<:loaded, "messages">,
      id: 5,
      content: "Iste cum provident tenetur.",
      room_id: 1,
      sender_id: 8,
      sender: #StreamChat.Accounts.User<
        __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
        id: 8,
        email: "keon@streamchat.io",
        ...
      >,
      inserted_at: ~N[2023-03-02 01:27:10],
      updated_at: ~N[2023-03-02 01:27:10]
    }},
  # ...
]
```

```elixir
defp do_reduce([{dom_id, _at, item} | tail], {:cont, acc}, fun) do
  do_reduce(tail, fun.({dom_id, item}, acc), fun)
end
```

Let’s inspect that list of tuples more closely

```elixir
messages = socket.assigns.streams.messages

for {dom_id, message} <- messages do
  IO.inspect {dom_id, message}
end
```

```
{"messages-5",
 %StreamChat.Chat.Message{
   __meta__: #Ecto.Schema.Metadata<:loaded, "messages">,
   id: 5,
   content: "Iste cum provident tenetur.",
   room_id: 1,
   sender_id: 8,
   sender: #StreamChat.Accounts.User<
     __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
     id: 8,
     email: "keon@streamchat.io",
     ...
   >,
   inserted_at: ~N[2023-03-02 01:27:10],
   updated_at: ~N[2023-03-02 01:27:10]
 }
}
# ...
```

Streams and DOM IDs

- LiveView uses each item's DOM id to track that item on the page and allow us to edit and delete the item.

- LiveView expects this DOM id to be attached to the HTML element that contains that stream item.

We attach the DOM id to each div produced by the iteration in our `:for` directive.

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

Remember, all stream data is stored client-side ONLY.

So, LiveView must be able to derive all the information it needs about the item and its position in the stream from the rendered HTML itself.

This is what the DOM id is for!

That’s all we need to render a list of stream items

Recap

We **stored** the initial stream in socket assigns, **iterated** over it, and **rendered** it using the required HTML structure and attributes.

Now, the page will render with this list of messages from the stream, and the live view will no longer hold this list of messages in state.

```
streams: %{
  __changed__: MapSet.new([:messages]),
  messages: %Phoenix.LiveView.LiveStream{
  name: :messages,
  dom_id: #Function<3.113057034/1 in Phoenix.LiveView.LiveStream.new/3>,
  inserts: [],
  deletes: []
}
```

Next up...

Updating with `stream_insert/4`

Assume we have:

- A form for a new message
- A PubSub integration to notify live views when a new message is created

```elixir
def handle_info(%{event: "new_message", payload: %{message: message}}, socket) do
  {:noreply,
    socket
    |> stream_insert(:messages, message)}
end
```

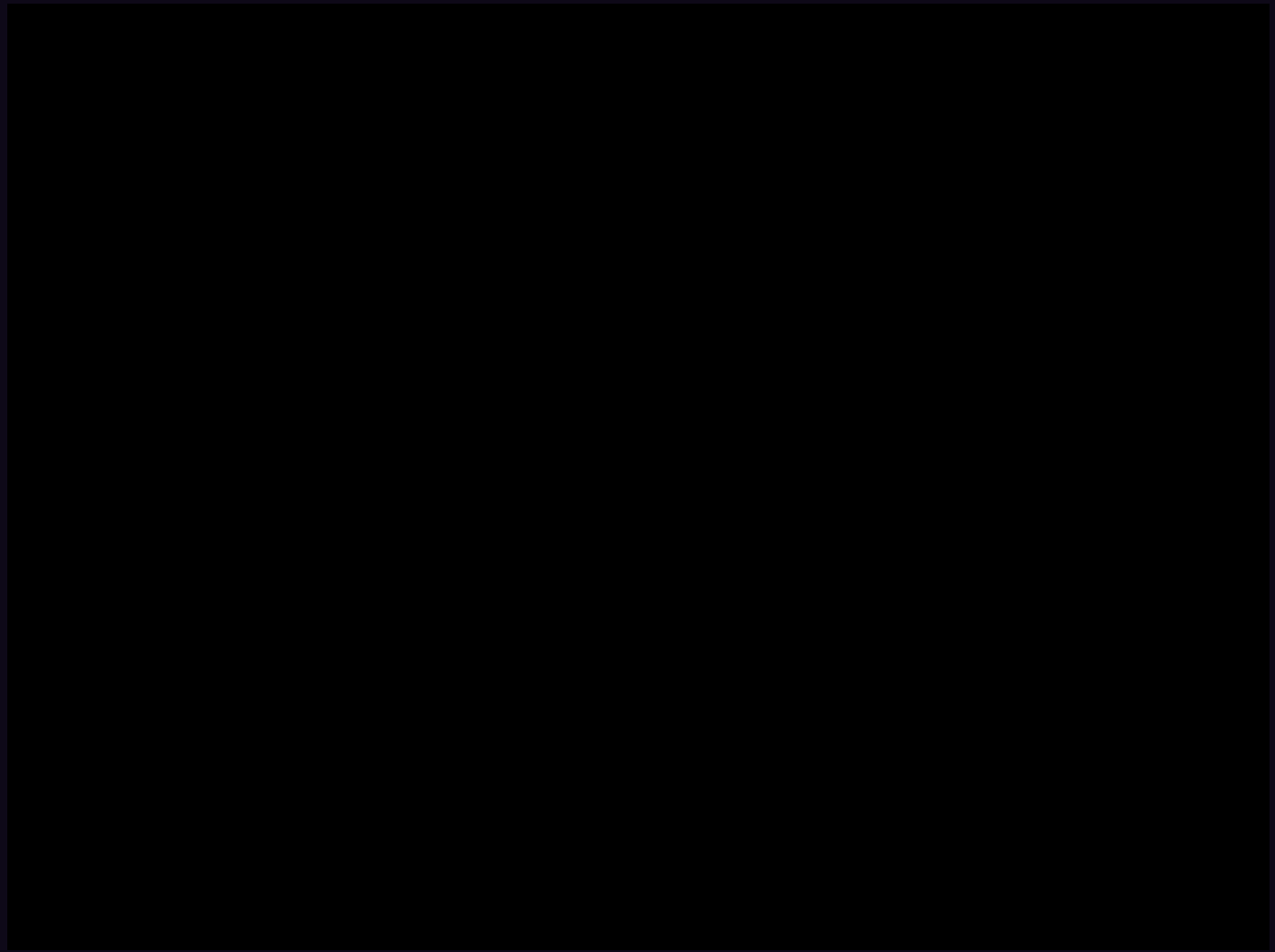
```
streams: %{
  __changed__: MapSet.new([:messages]),
  messages: %Phoenix.LiveView.LiveStream{
    name: :messages,
    dom_id: #Function<3.113057034/1 in Phoenix.LiveView.LiveStream.new/3>,
    inserts: [
      {"messages-17", -1,
        %StreamChat.Chat.Message{
          id: 17,
          content: "10",
          #...
        }
      },
    ],
    deletes: []
  }
}
# ...
```

That's all you need to tell LiveView to append a new item

Now, LiveView will re-render the template

And the LiveView client will follow the instructions in `@streams.messages` to append the new message

What about updating an existing message in place?



Let’s do that now

Assume we have:

- A form for editing an existing message.
- A PubSub integration to notify live views when a message is edited.

```elixir
def handle_info(%{event: "updated_message", payload: %{message: message}}, socket) do
  {:noreply,
    socket
    |> stream_insert(:messages, message, at: -1)}
end
```

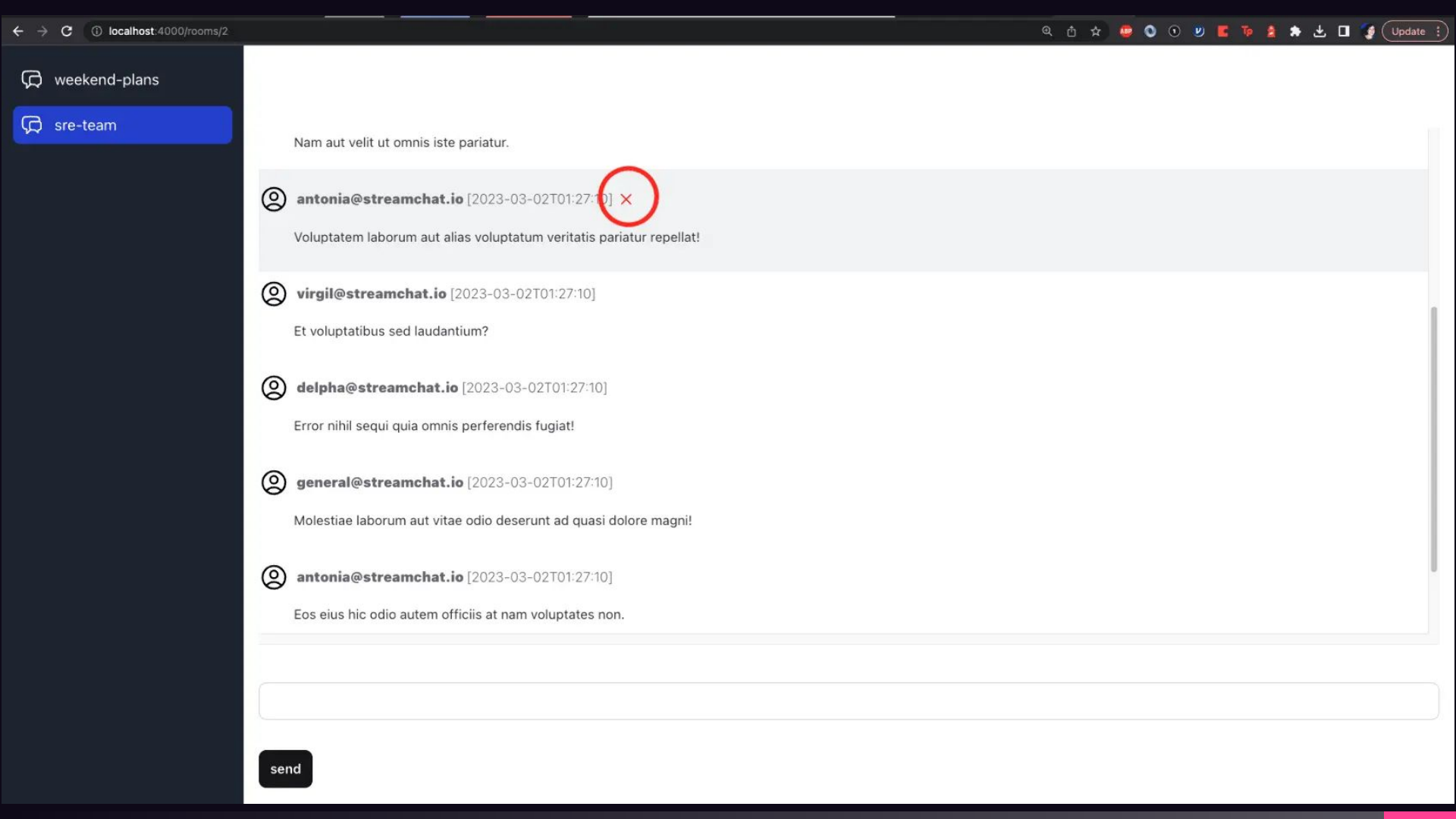
Another example of the declarative nature of LiveView

We don’t have to teach LiveView how to update a message in place

We just tell it *which* message to update

One last bit of streams functionality before you go...

Deleting a message


localhost:4000/rooms/2
Update
weekend-plans
sre-team
Nam aut velit ut omnis iste pariatur.
antonia@streamchat.io [2023-03-02T01:27:10]
Voluptatem laborum aut alias voluptatum veritatis pariatur repellat!
virgil@streamchat.io [2023-03-02T01:27:10]
Et voluptatibus sed laudantium?
delpha@streamchat.io [2023-03-02T01:27:10]
Error nihil sequi quia omnis perferendis fugiat!
general@streamchat.io [2023-03-02T01:27:10]
Molestiae laborum aut vitae odio deserunt ad quasi dolore magni!
antonia@streamchat.io [2023-03-02T01:27:10]
Eos eius hic odio autem officiis at nam voluptates non.
send

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

The call to `steam_delete` returns a socket with an assigns that looks something like this...

```
streams: %{
  __changed__: MapSet.new([:messages]),
  messages: %Phoenix.LiveView.LiveStream{
    name: :messages,
    dom_id: #Function<3.113057034/1 in Phoenix.LiveView.LiveStream.new/3>,
    inserts: [],
    deletes: ["messages-20"]
  }
}
```

This instructs LiveView to remove the item with that DOM ID from the rendered list of @streams.messages.

That’s all we need to delete a stream item

We tell LiveView *what* to delete

Not *how* to delete it

The framework handles the details of “how”

Actually I lied, I want to show you one more thing!

Bonus: infinite scrollback with JS bindings

Up until recently, you needed JS Hooks

But not anymore!

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

```elixir
def handle_event("load_more", _, socket) do
  {:noreply, paginate_messages(socket)
end
```

```elixir
def paginate_messages(%{assigns: %{last_message_id: id}} = socket) do
  previous_ten = Chat.previous_ten_messages(last_message_id)
  socket
    |> assign(:last_message_id, Enum.first(previous_ten).id)
    |> stream(:messages, previous_ten, at: 0)
end
```

- Calling `stream/4` on an existing stream will bulk insert the new items on the client while leaving the existing items in place.

More things you can do with streams

You can reset a stream—emptying it out or overriding its contents.

- You can “limit” a stream so that only a certain number of items are rendered at a time.

Let’s wrap up

We've touched on just some basic examples of how to use streams to manage data in your live views

And we looked at some stream functionality under the hood to see how LiveView handles the details

Streams allow LiveView to handle large data-sets efficiently

They’re powerfully declarative

An integrate with LiveView's JavaScript offerings seamlessly

With just a few lines of code we implemented an infinite scrollback

You never have to teach LiveView *how* to operate

You tell LiveView *what* to do

And the framework handles the hard parts for you.

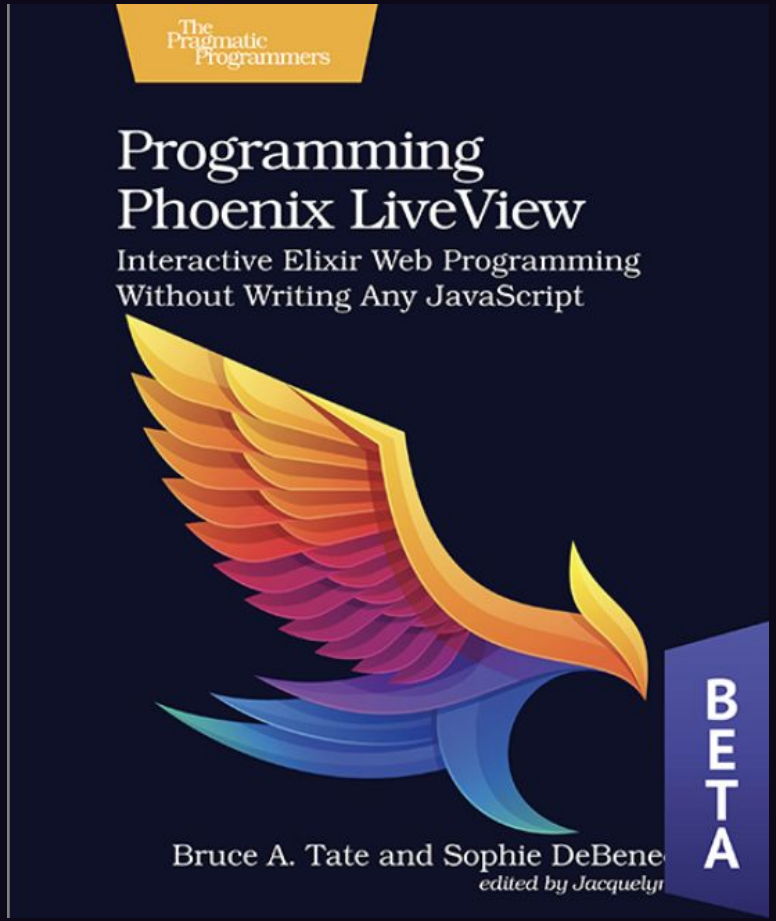
And the framework handles the hard parts for you.

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**