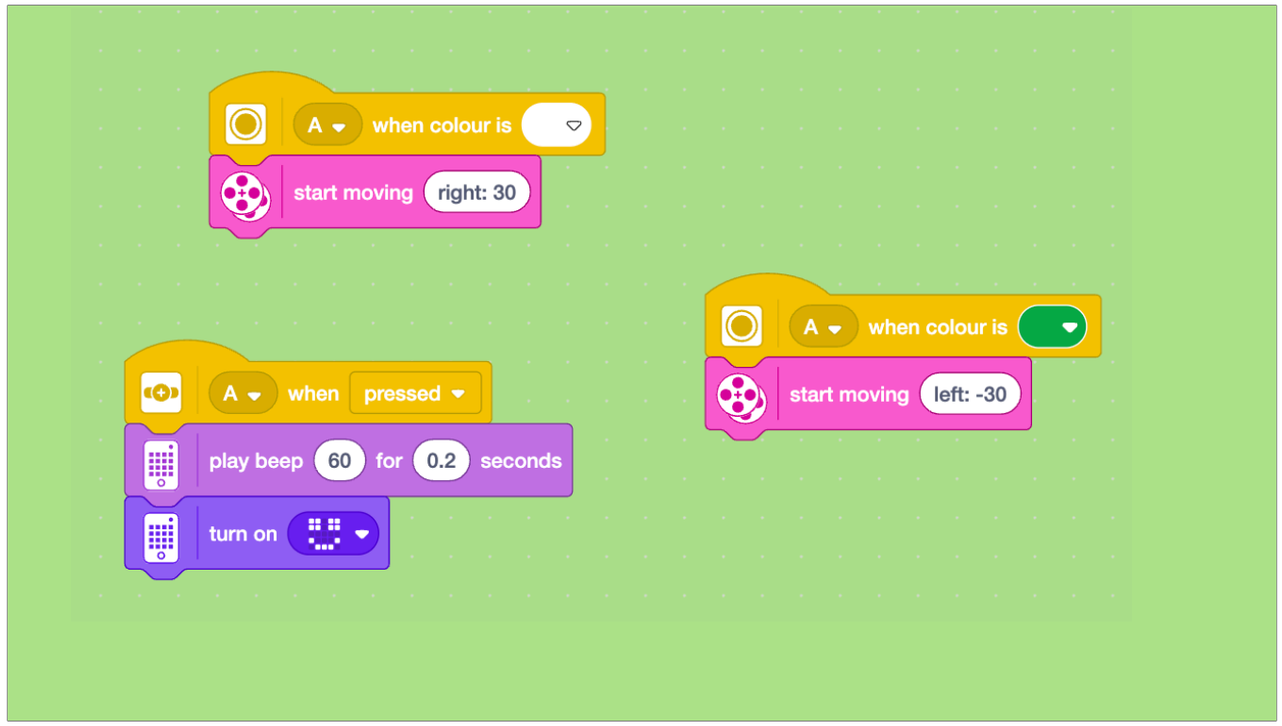
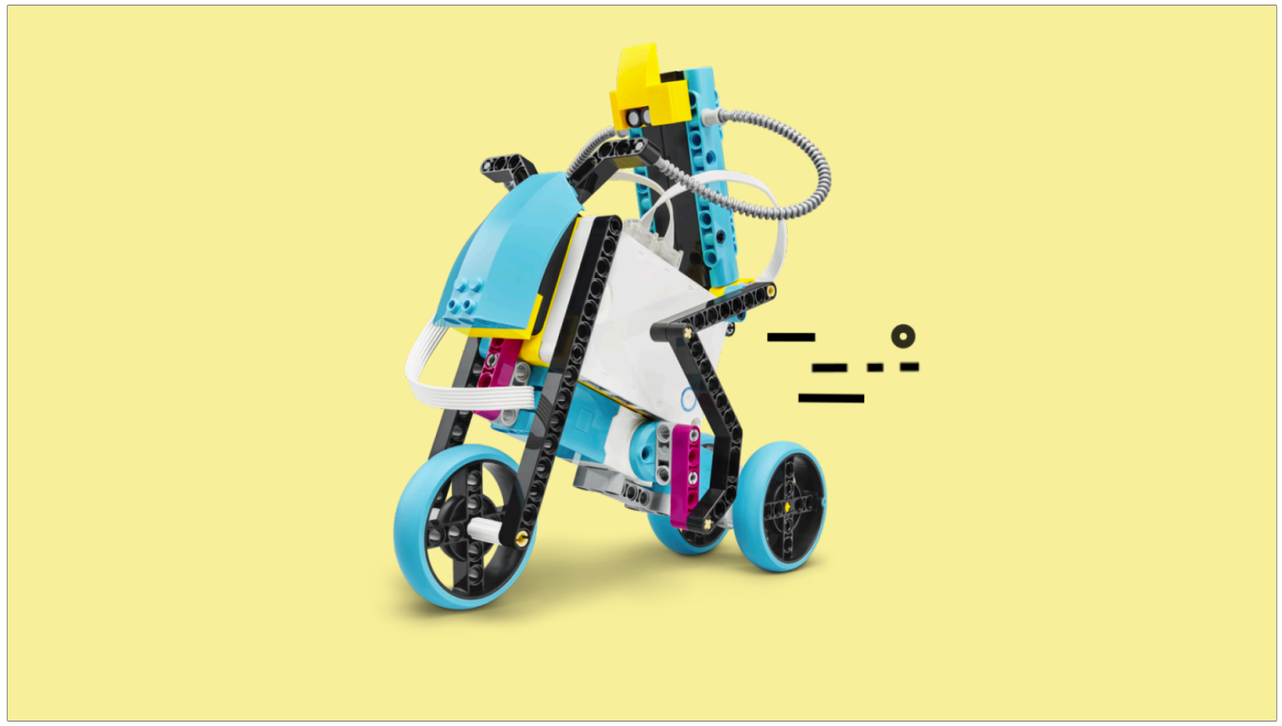
This is going to be a hodgepodge of stuff I learned about Phoenix that might be a little off the beaten path. It'll be bit all over the place: Something on Channels, Phoenix PubSub and Phoenix Presence. For folks experienced in Phoenix there's likely not going to be much of value here. But for folks like me where Phoenix is still relatively new, I'm hoping there's something here from you. But before we get into the details, it might be worth addressing why there's a LEGO robot on this slide.

Sometime before my current gig at Maersk, I work for a few years as a Lead Developer at LEGO Education where I got to lead the software team as we worked on what would eventually become SPIKE Prime and SPIKE Essential.
The elevator pitch of which is…

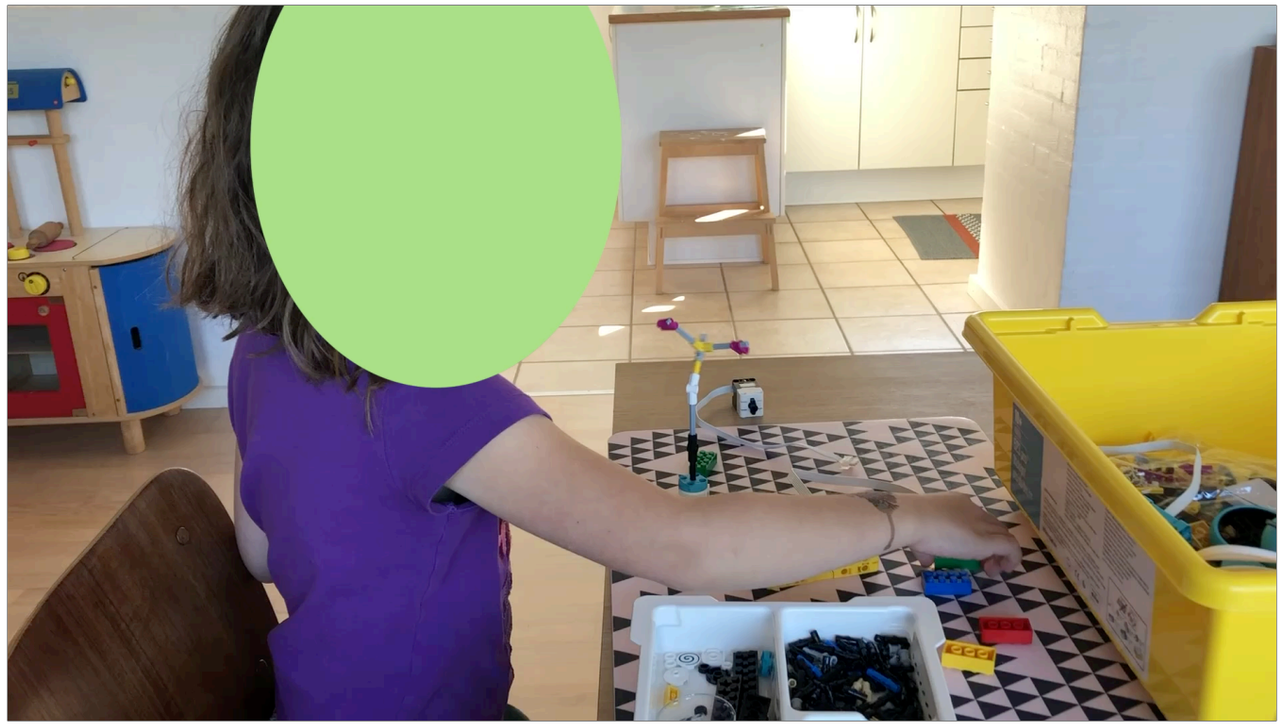


To allow kids to use a programming language that looks like this (this is Scratch)…



To make something like this do the most awesome things…
So that they can learn physics, maths and programming…. All while

Feeling like they are Inventing, Learning and having fun. And I’d like to feel like we made it.



This is my youngest, Sofie 10 now but, age 6 at the time, explaining to me that which colours make her sculpture move the fastest (its red…red is a very fast color)). You'll see the sculpture wobble a bit. Just before this it was falling over when rotating, which lead to a really nice chat about the connection between balance and symmetry. And that you can look for symmetry along multiple axis.
Not a concepts I'm sure I'd have been able to impart on her if she wasn't playing, having fun and being able to touch and feel what was going on.

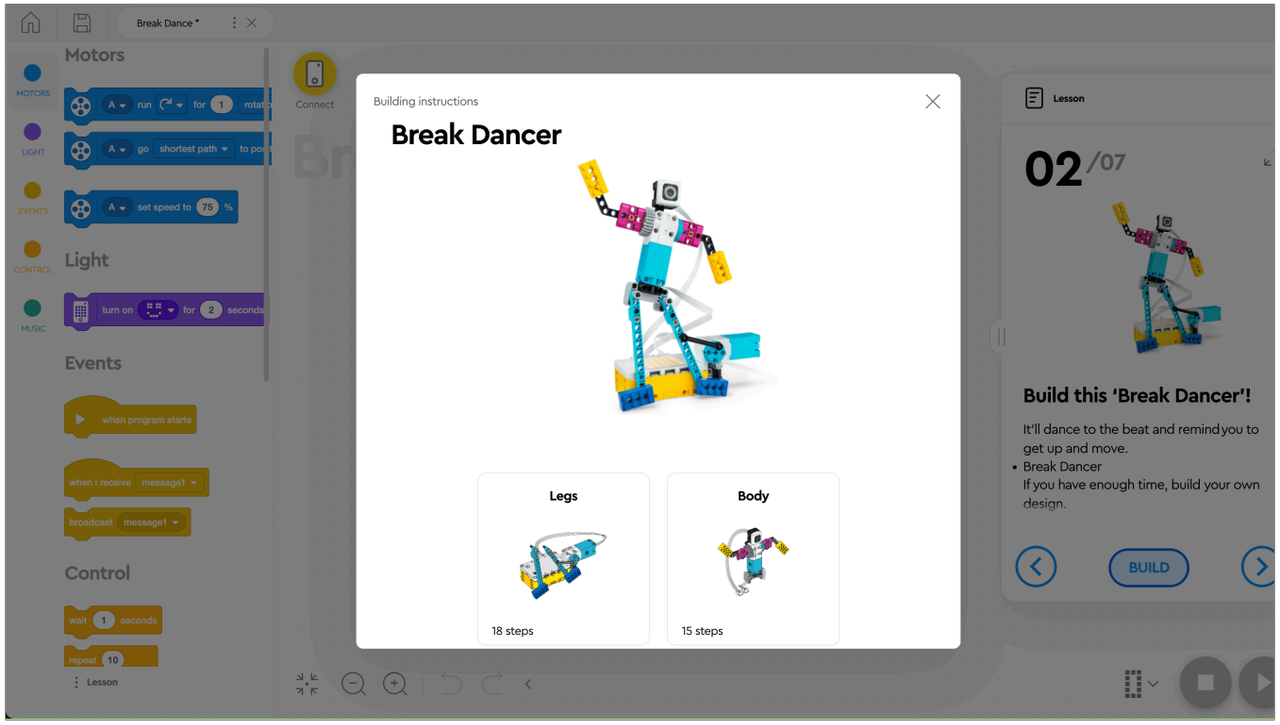
We failed

But we also failed (and quite significantly I feel)

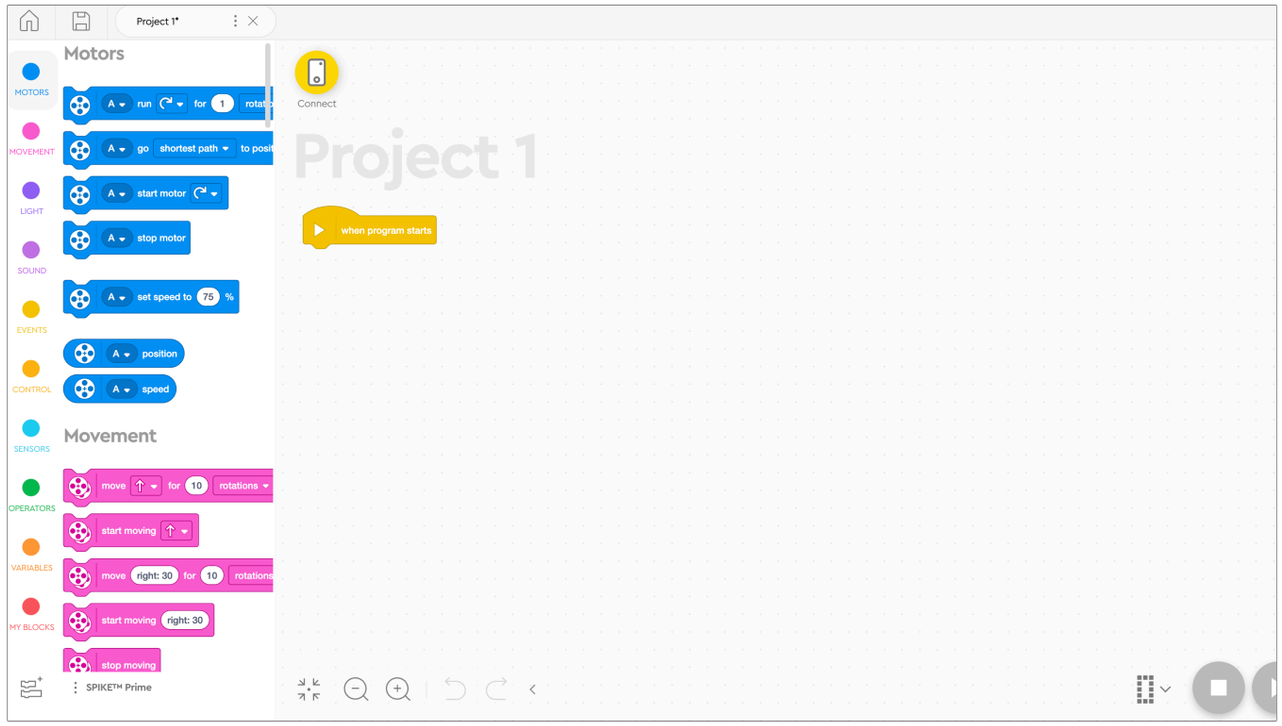
… create active,
COLLABORATIVE,
lifelong learners.

- LEGO Education

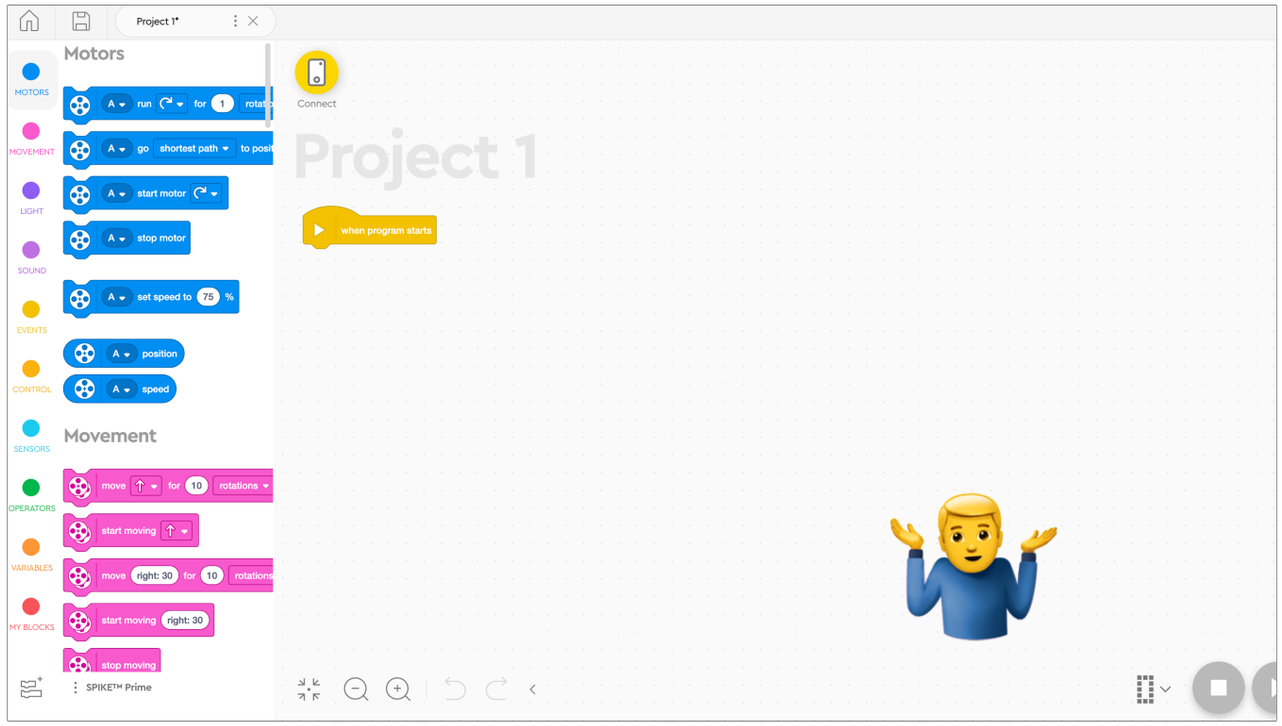
Because part of the LEGO Education vision was to “Create active Collaborative, lifelong learners”
And the tried taking the COLLABORATIVE part serious.
The product was designed so that two students would share a single set.



We had “buddy-builds” where two students would collaborate on constructing different parts of a model and then integrate it.



But when it came to the programming part… we kinda just gave up.



They’ll just have to work on the same computer. Oh.. and we’ll add a feature so that they can share the project file…and share it by mailing it around. That should do it, right?

Covid

Then all the students got sent home. For a long time. So they couldn’t share a set. They couldn’t collaborate on building. They couldn’t even collaborate on coding.

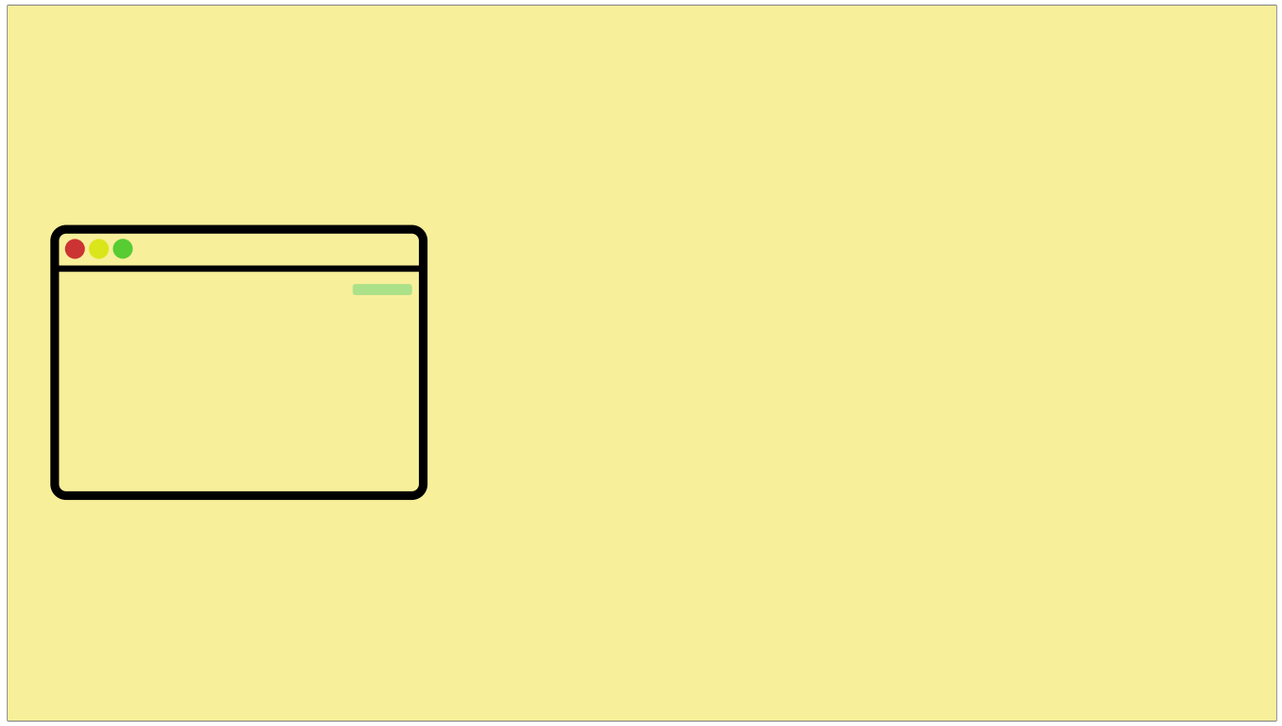
And to someone who *really* enjoys pair programming…

…that felt like defeat.

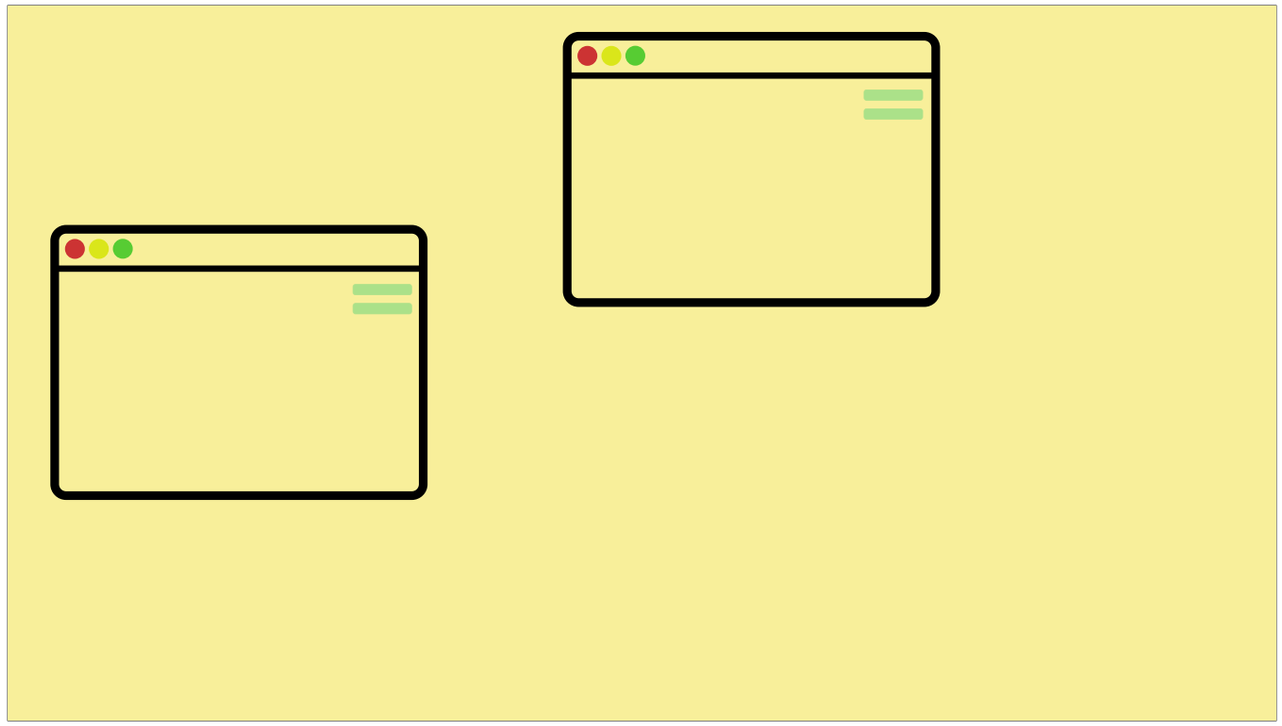
(But we just didn’t have any good ideas on how to solve it)

But we just didn’t have any good ideas for how to solve it. We we’re in this great big beast of a React app and really didn’t have the tools to fix this readily available to us.

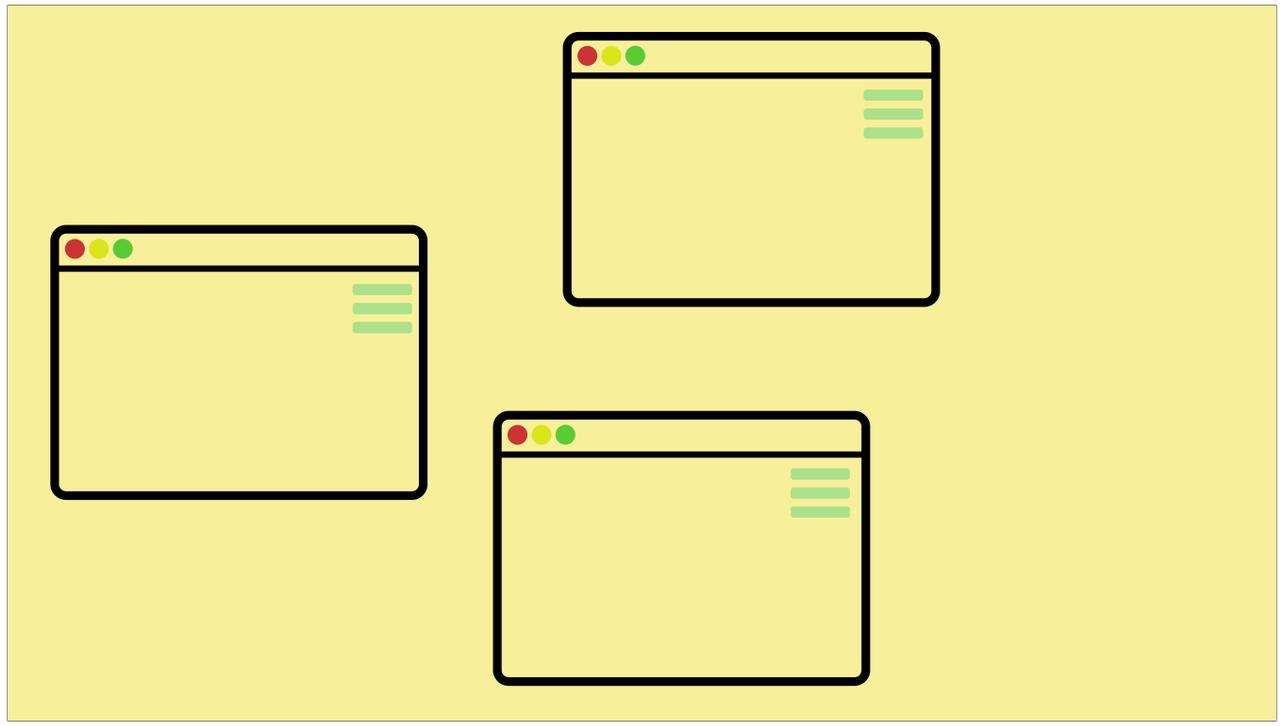
Turns out Phoenix does have those tools. Built in. Ready to go. And incredibly easy to use. So I went about trying to build something.



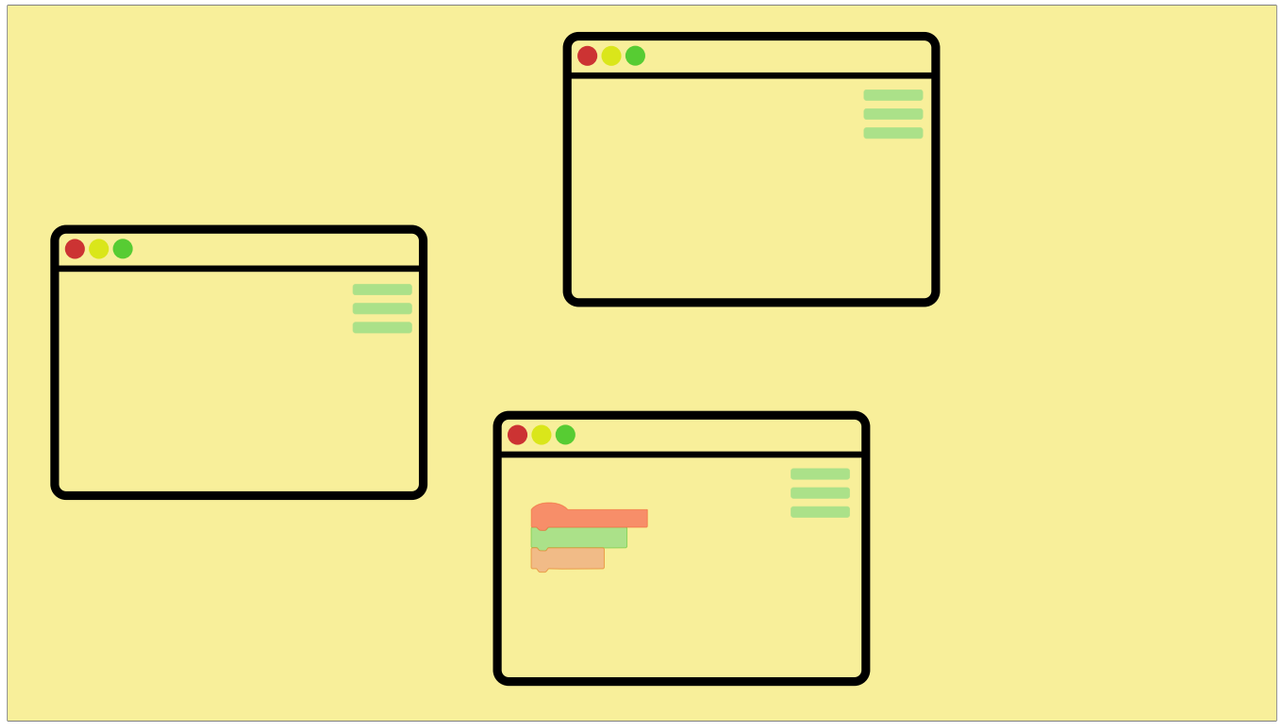
So the idea is as follows. Users can open the website



Be aware of each other

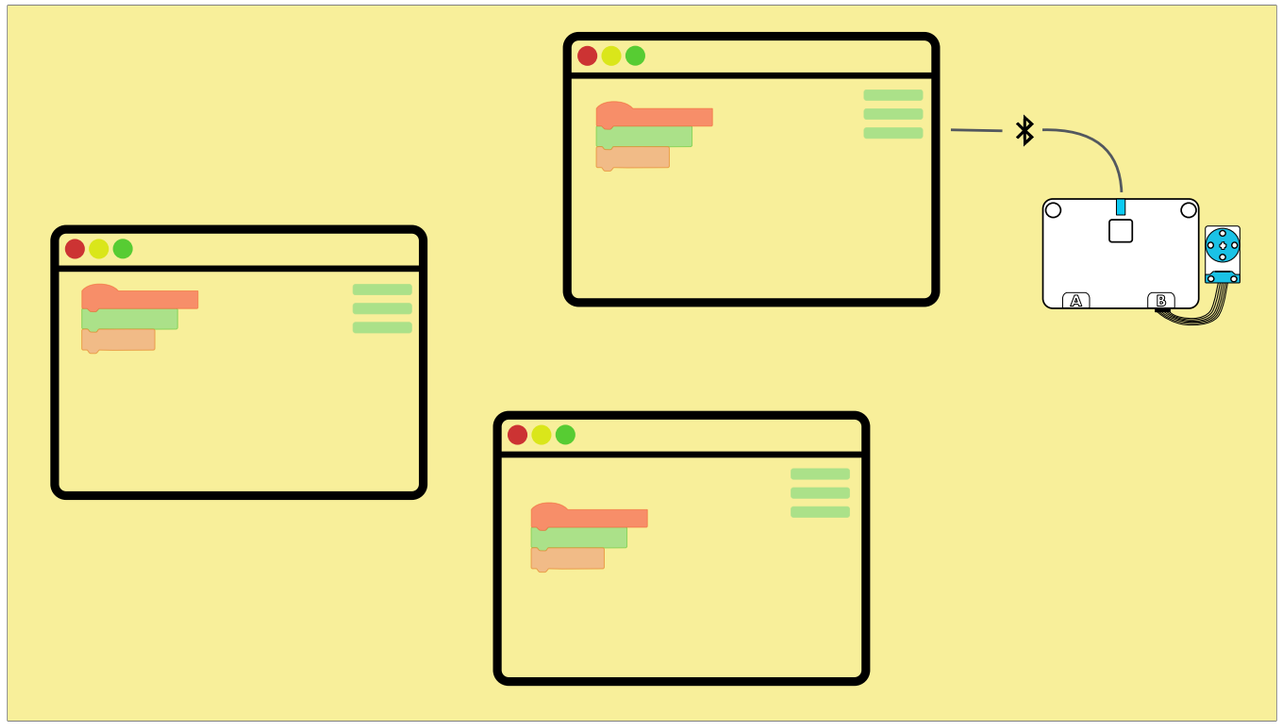


As more join…

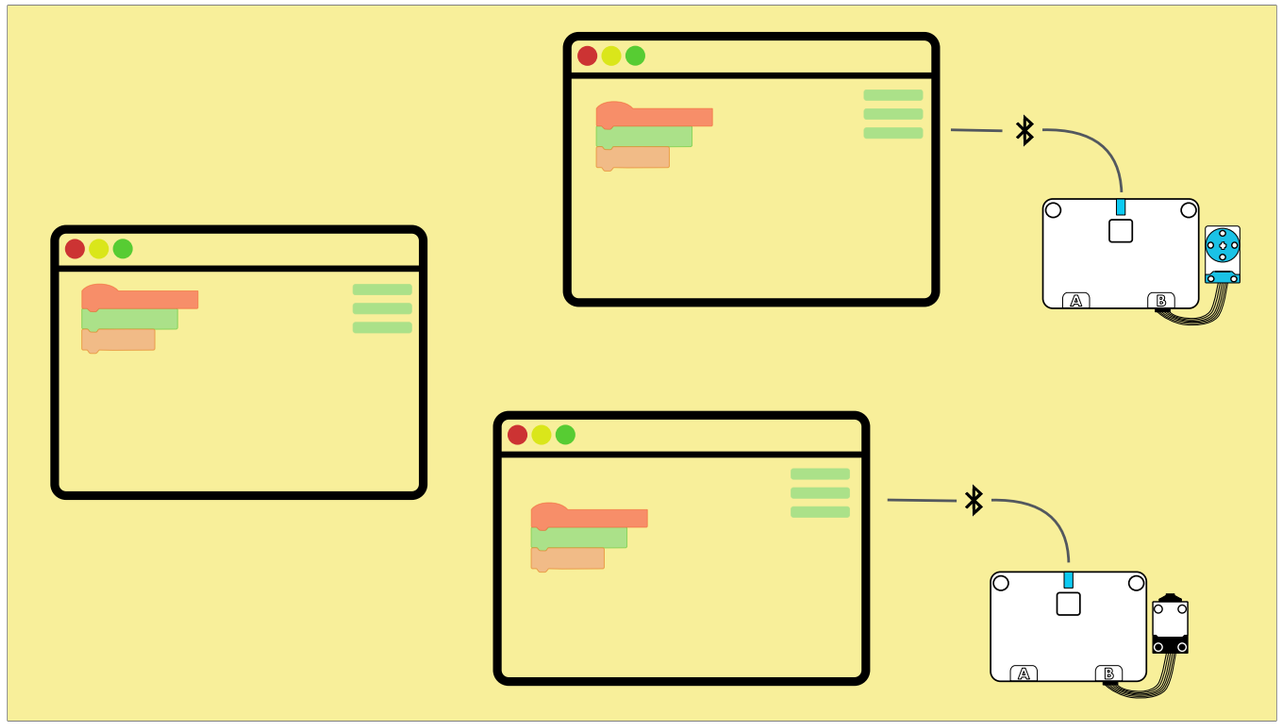


And collaboratively start building a Scratch program

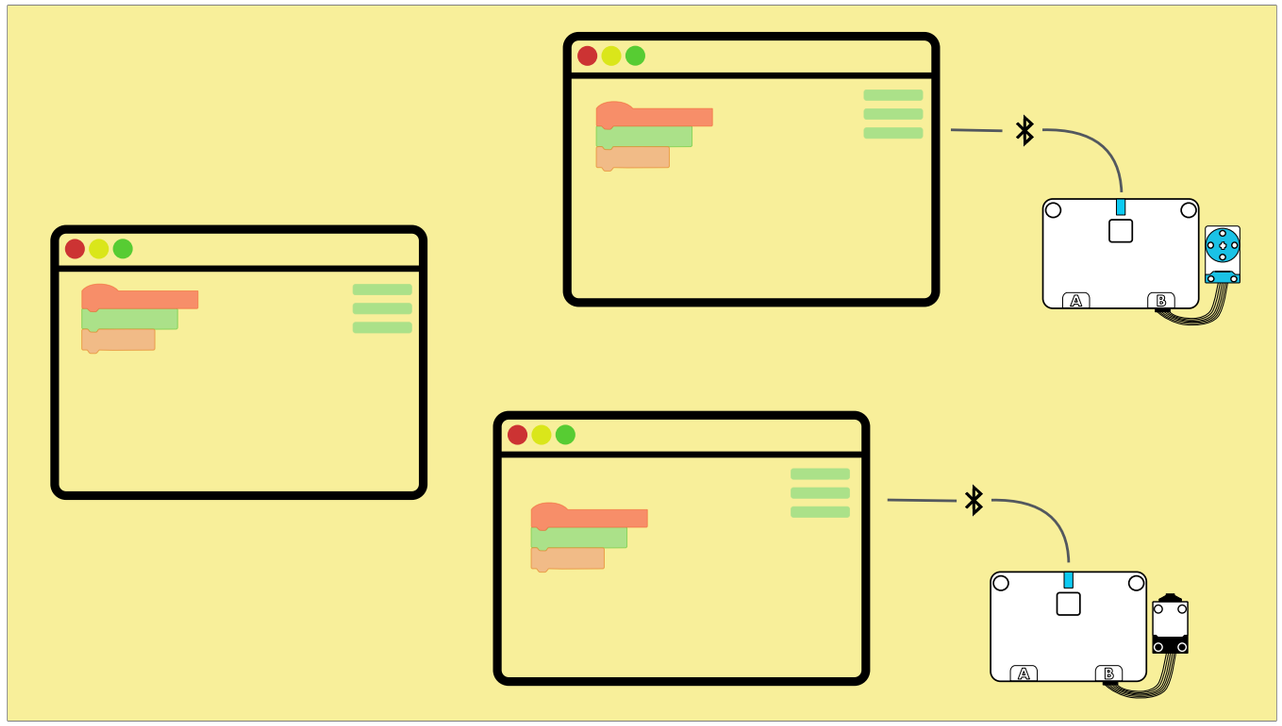
That syncs seamlessly across all the participants…



Where robots can be connected….



By multiple users



And where input to one robot…. [click]…can be detected by the program they created.[click]…. And end up affecting another robot.

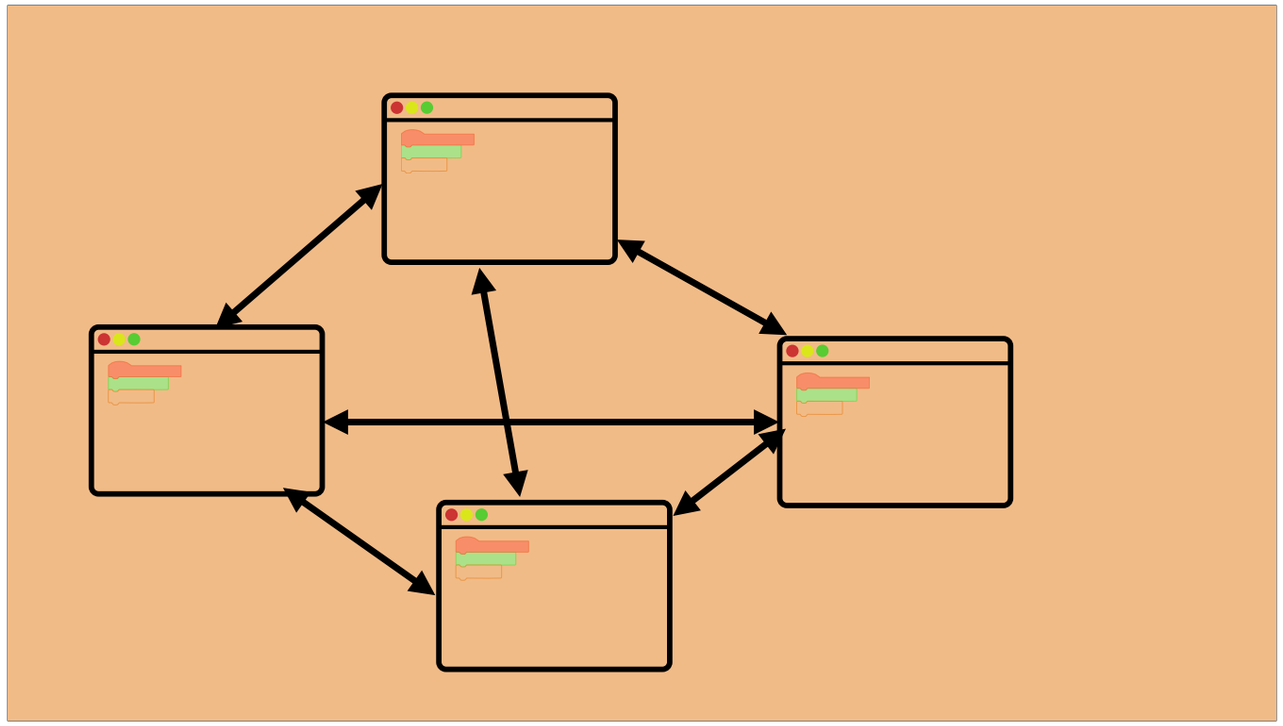
Demo

Channels

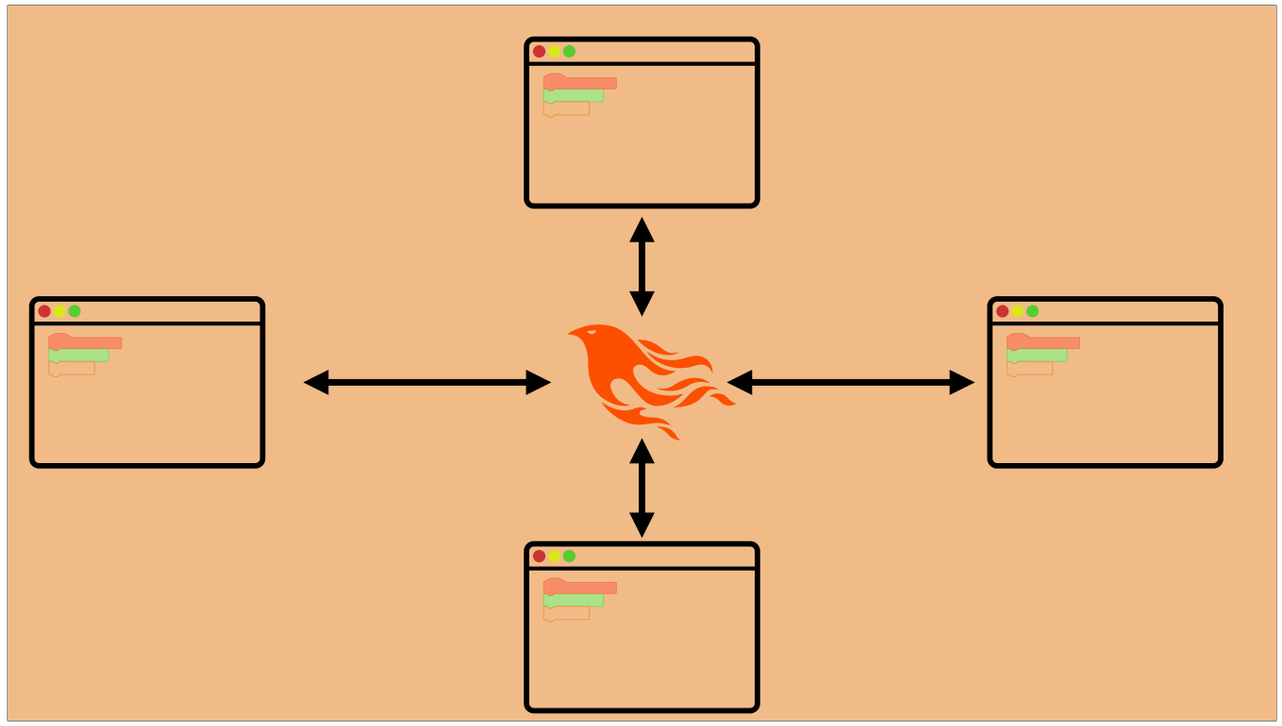
Channels are what powers almost all of what you just saw…

TLDR; WebSockets

And the TLDR; for channels are that they are WebSocket connections between the users' browser and the Phoenix app.



The first challenge to tackle is: How to keep the programming canvases in sync between all the users by sharing changes between any browser connected to the app.



And Channels makes this almost trivial… so lets see what it takes to setup a Channel in Phoenix

Phoenix: Endpoint, Socket & Channel

```
socket("/live", Phoenix.LiveView.Socket, websocket: [connect_info: [session: @session_options]])

socket("/workspace_sync", BrickScriptCollectiveWeb.WorkspaceSocket,
  websocket: true,
  longpoll: false
)
```

```
defmodule BrickScriptCollectiveWeb.WorkspaceSocket do
  use Phoenix.Socket

  channel "workspace:*", BrickScriptCollectiveWeb.WorkspaceChannel

  @spec connect(any, any, any) :: {:ok, any}
  def connect(_params, socket, _connect_info) do
    {:ok, socket}
  end

  @spec id(any) :: nil
  def id(_socket), do: nil
end
```

```
defmodule BrickScriptCollectiveWeb.WorkspaceChannel do
  use BrickScriptCollectiveWeb, :channel

  @impl true
  @spec join(<<_::96>>, any, any) :: {:ok, any}
  def join("workspace:42", _payload, socket) do
    {:ok, socket}
  end
end
```

This the setup you need in Phoenix. You'll get most of it for free with the Phoenix channel generator mix task.

Let's start with Endpoint.ex: Indicate that we allow socket connections on "/worksspace_sync" (You'll notice "/live" here, because, yeah, live views are built on top of Phoenix Channels).

You point to a Model to handle any connections with the Socket module… a socket can have multiple channels. You define a topic_pattern (wildcards allowed), and a module to handle that channel. And that brings us to the last bit of code, the channel handler.
Pretty simple for now (we'll add a bit to it later).
Squint and this should feel familiar if you've worked with LiveView. Handle the join, optionally put some state into the socket assigns, return the {:ok, socket} tuple.

And just like LiveView - each socket connection is it's own process.

Phoenix: JavaScript

```
import { Socket } from "phoenix"

// Connect to the socket
let socket = new Socket("/workspace_sync", { params: { token: window.userToken } })
socket.connect()

// Select your channel...
export let channel = socket.channel("workspace:42", {})

//... and join it
channel.join()
  .receive("ok", resp => { console.log("Joined successfully", resp) })
  .receive("error", resp => { console.log("Unable to join", resp) })

export default socket


export const workspace_update_callback = (update) => {
  channel.push("workspace_update", update)
}
```

So far so good, our phoenix app accepts WebSocket connections. But how do we connect? Well, the phoenix generator gives you all you need.
Connect a WebSocket, join a channel.

Notice the last function here that I've added (I don't know why I opted for snake_case here…). Channel.push is all you need to send a message over the web socket.
Phoenix handles most datatypes you can throw at it - ArrayBuffers get special handling

Phoenix: All Together Now

Sending messages

```
export const workspace_update_callback = (update) => {
  channel.push("workspace_update", update)
}
```

Receiving messages

```
channel.on("workspace_update", payload => {
  const updatedProject = projectData()
  updatedProject.targets[1].blocks = payload.blocks

  loadProject(updatedProject)
});
```

We've got most of our puzzle pieces in place. It's mostly autogenerated code. Let's take a look at the herculean effort required to share the canvas updates between all the browsers.

Javascript side first: Ok, not to bad. Sending and receiving messages is pretty simple… you use your channel.

Phoenix: All Together Now

```
export const workspace_update_callback = (update) => {
  channel.push("workspace_update", update)
}
```

Event Name

Message Payload

```
channel.on("workspace_update", payload => {
  const updatedProject = projectData()
  updatedProject.targets[1].blocks = payload.blocks

  loadProject(updatedProject)
});
```

You send an event name and a message payload.
You receive the same things… bit of pattern-matching like API on the JS side, allows you to listen to specific event types if you need it.

But surely the backend is going to be tough to deal with… lets see how much bigger our Channel Handler Module has become….ready?

Phoenix: All Together Now

```elixir
defmodule BrickScriptCollectiveWeb.WorkspaceChannel do
  use BrickScriptCollectiveWeb, :channel

  @impl true
  @spec join(<<_::96>>, any, any) :: {:ok, any}
  def join("workspace:42", _payload, socket) do
    {:ok, socket}
  end

  def handle_in("workspace_update", payload, socket) do
    broadcast(socket, "workspace_update", payload)
    {:noreply, socket}
  end
end
```

That’s it. Receive the message. Broadcast it to all the connected sockets. _Easy_.

Phoenix: A Small Wrinkle

```
channel.on("workspace_update", payload => {
  const updatedProject = projectData()
  updatedProject.targets[1].blocks = payload.blocks

  if (payload.workspace_id === workspace.id) {
    return
  } else {
    const updatedProject = projectData()
    updatedProject.targets[1].blocks = payload.blocks
  }
});
```

Broadcast will broadcast to _all_ sockets, so you might need to protect yourself agains your own messages. It makes sense, but it’s something that might catch you off guard.

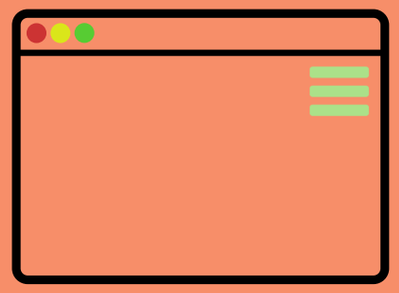
Phoenix: A Few Additional Details

- Use ‘push/3’ to send to a specific socket
- Broadcast from anywhere with ‘MyAppWeb.Endpoint.broadcast!/2’

Phoenix Presence

Awareness of Other Users

Phoenix Presence: “Provides Presence tracking of processes and users”



It was what you saw at the right side of the screen. A way of indicating that there are other folks using the site.

Phoenix Presence: The Basics

```elixir
defmodule YourLiveView.Index do
  def mount(_params, _session, socket) do
    # Enroll into presence. Ie. "I'm here 👋"
    Presence.track(self(), "users", socket.assigns[:user_name], %{})
    # Subscribe to changes
    Phoenix.PubSub.subscribe(PubSub, "users")

    # Grab the list of users, stick it in the assigns so what we can render it
    {:ok, socket |> assign(:connected_users, Presence.list("users"))}
  end

  def handle_info(%Phoenix.Socket.Broadcast{event: "presence_diff", payload: diff}, socket) do
    folks_who_joined = diff.joins
    folks_who_left = diff.leaves
    folks_who_are_here = Presence.list("users")

    # Grab the list of users, stick it in the assigns so what we can render it
    {:ok, socket |> assign(:connected_users, Presence.list("users"))}
  end
end
```

To use Phoenix Presence, you'll need do a bit of configuration in Application.ex and then you can start using it in your liveviews.

Two parts to this: Tell Presence that a user has joined via Presence.track and Subscribing to a Pubsub topic and listening to presence_diff messages.
Throw Presence.list() into your assigns and you're ready to render the list of users. Easy!

You will notice in the "presence_diff" that as part of the diff message we get two lists: folks who have joined and folks who have left.

Phoenix Presence: A Wrinkle

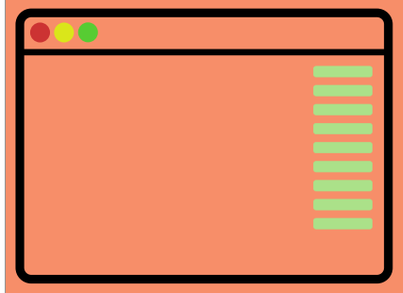
```elixir
def mount(_params, _session, socket) do
  if connected?(socket) do
    Presence.track(self(), "users", socket.assigns[:user_name], %{})

    # You only want to call this once (or you get duplicate events)
    Phoenix.PubSub.subscribe(PubSub, "users")

    {:ok, socket |> assign(:connected_users, Presence.list("users"))}
  end
end
```

There is one wrinkle though. LiveViews first get the “dead render” and then the “live render” once the websocket connects. You’re totally allowed to subscribe to the PubSub twice, but you’ll then get the presence_diff events twice…. Which feels icky. Best solution I could come up with was to only track and subscribe once the web socket has connected via the “connected” function. I don’t love it.

Phoenix Presence: Everything Fits Together



```elixir
defmodule YourLiveView.Index do

  def handle_info(%Phoenix.Socket.Broadcast{event: "presence_diff", payload: diff}, socket) do
    folks_who_are_here = Presence.list("users")

    # Doing this will cause you to send and replace the full list on every update
    {:ok, socket |> assign(:connected_users, Presence.list("users"))}
  end
end
```

My first thought when adding presence was: Why am I getting a list of leaves and joins when I always end up sending the full list?

It’s because you shouldn’t

Use `stream` for lists

One thing that continually amazes me about Phoenix is how well thought trough things seem. So when I find things that look dumb, that’s a sign there is something I’ve missed. And in this case it was ‘stream’. The list of users is a …ehh.. list.. and you should probably use “stream” over “assign” for lists.

Phoenix Presence: Stream

```elixir
defmodule YourLiveView.Index do
  def handle_info(%Phoenix.Socket.Broadcast{event: "presence_diff", payload: diff}, socket) do
    folks_who_are_here = Presence.list("users")

    {:ok,
     socket
     # Tell the stream how to construct DOM Id's from users
     |> stream_configure(:connected_users, dom_id: &user_to_id/1)
     |> stream(:connected_users, folks_who_are_here)
  end

  def handle_info(%Phoenix.Socket.Broadcast{event: "presence_diff", payload: diff}, socket) do
    # Clear out the folks who left
    socket =
      diff.leaves
      |> Enum.reduce(socket, fn leaver, acc ->
        stream_delete_by_dom_id(acc, :connected_users, leaver.dom_id)
      end)

    # Add the folks who joined
    socket =
      diff.joins
      |> Enum.reduce(socket, fn joiner, acc ->
        stream_insert(acc, :connected_users, joiner)
      end)

    {:noreply, socket}
  end
end
```

Changing over to streams gives us a few really neat features: Stream_delete_by_dom_id and stream_insert. This means we’re now only sending over info on the items that we’ve deleted and the items added. Smaller payload. Smaller DOM change.
This works because of “stream_configure” which allows you to configure how phoenix should assign DOM_IDs for items in the list you stream.

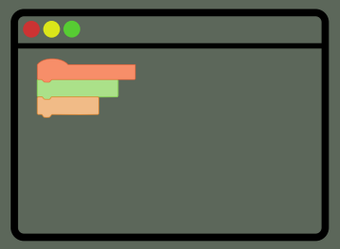
PubSub

Realtime Publisher/Subscriber service

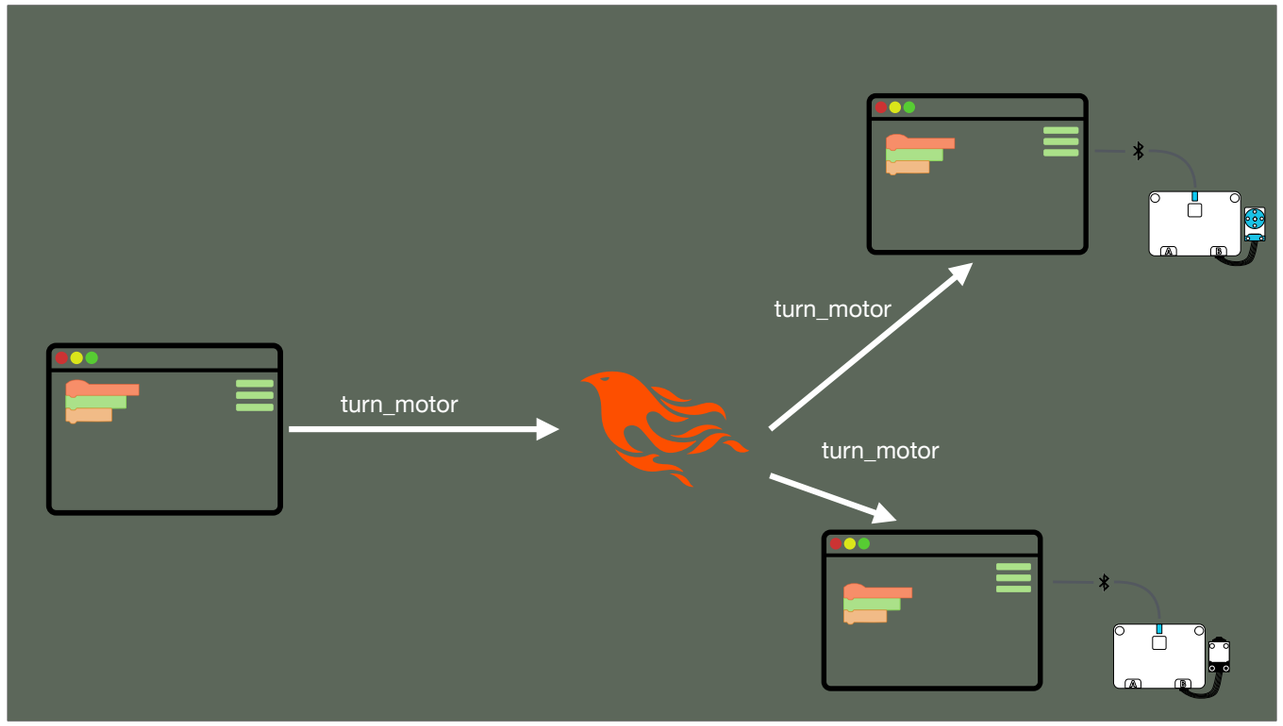
Here at the end I'll touch very briefly on Phoenix Pubsub.

The Problem

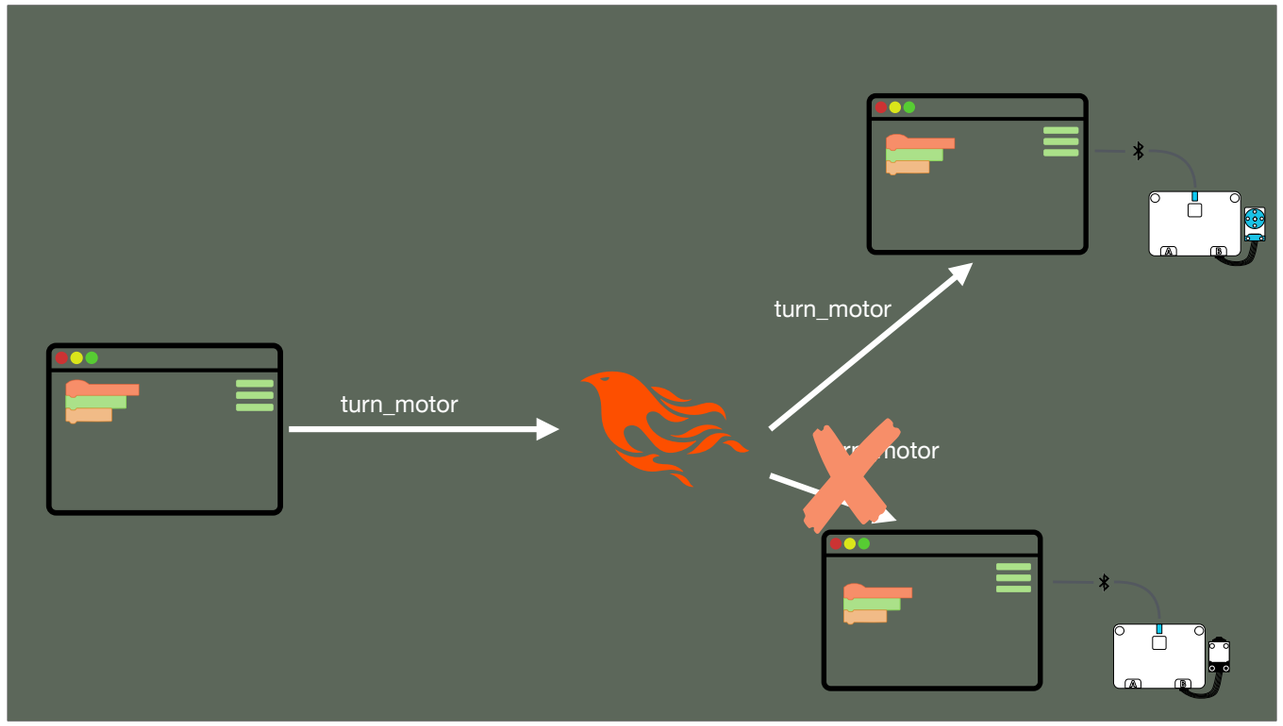
Scratch: Scratch-Blocks and Scratch-VM



- Scratch-blocks is the UI
- Scratch-VM runs the program - but only runs in one browser



There’s a complication to talking to robots: You need to be careful what messages you send to them. It’s no use sending a “Motor turn” command on a robot that has a force attached.



I’ve got a GenServer running for each connected robot that keeps track of what is attached to the robot. That GenServer also has the socket to the browser that the robot is connected to.

The solution?

Phoenix PubSub

Sending Messages

```
def handle_in("vm_command", payload, socket) do
  PubSub.broadcast(BrickScriptCollective.PubSub, "vm_command", {:vm_command, payload})
  {:noreply, socket}
end
```

Receiving messages

```
def init([owning_socket]) do
  PubSub.subscribe(BrickScriptCollective.PubSub, "vm_command")

  {:ok, %{owning_socket: owning_socket, robot: %Robot{}}}
end
```

```
def handle_info(message, state) do
  updated_state = handle_event(message, state)

  {:noreply, updated_state}
end
```

Phoenix PubSub. You send messages to a topic.
Elsewhere you subscribe to a topic and they you start receiving messages.

PubSub is Foundational

You may have noticed in the last slide that I just referred to a already existing PubSub.

PubSub is Foundational

```
def handle_in("vm_command", payload, socket) do
  PubSub.broadcast(BrickScriptCollective.PubSub, "vm_command", {:vm_command, payload})
  {:noreply, socket}
end
```

That bit here. Why did that already exist?

It’s because PubSub is foundational.
How does a channel push a message or broadcast? It uses PubSub. How does Presence let you know about joins and leaves? You subscribe to a PubSub.

Channels build on PubSub. LiveView builds on Channels. So you can see that its underpins a lot of what makes Phoenix great.

“It uses Distributed Elixir, directly exchanging notifications between servers”

HexDocs for Phoenix.Pubsub

Pubsub is already super useful for sending messages across processes without directly coupling them, but it _also_ works across nodes.
Meaning that if you broadcast something on a channel topic. That broadcast can span multiple nodes.

And that… is pretty neat.

Thank you!