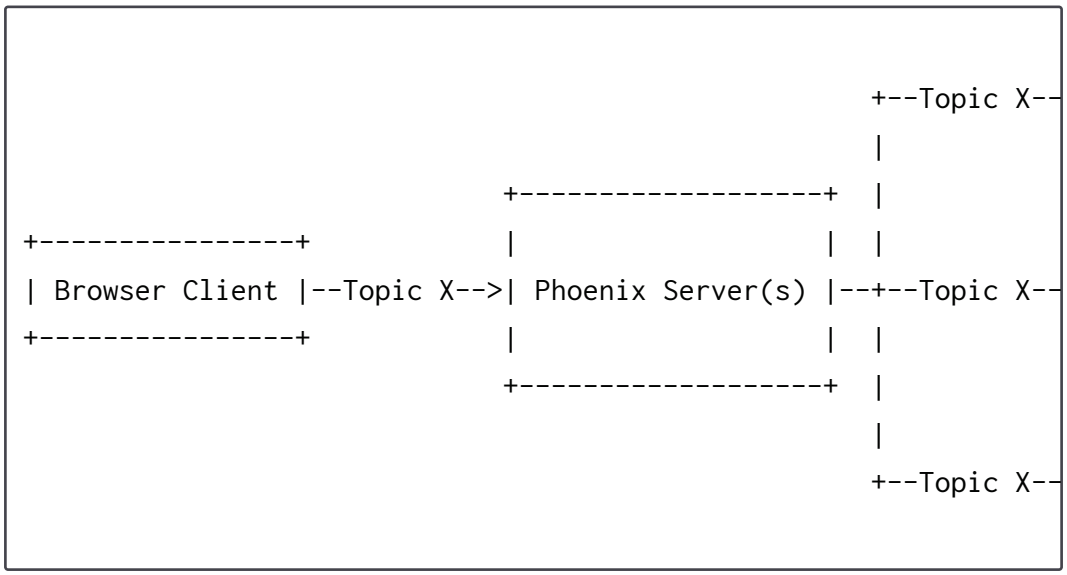
Channels

Channels are an exciting part of Phoenix that enable soft real-time communication with and between millions of connected clients. Some possible use cases include:

- Chat rooms and APIs for messaging apps
- Breaking news, like "a goal was scored" or "an earthquake is coming"
- Tracking trains, trucks, or race participants on a map
- Events in multiplayer games
- Monitoring sensors and controlling lights
- Notifying a browser that a page's CSS or JavaScript has changed (this is handy in development)

Conceptually, Channels are pretty simple. Clients connect and subscribe to one or more topics, whether that's `public_chat` or `updates:user1`. Any message sent on a topic, whether from the server or from a client, is sent to all clients subscribed to that topic (including the sender, if it's subscribed), like this:



Channels can support any kind of client: a browser, native app,

smart watch, embedded device, or anything else that can connect to a network. All the client needs is a suitable library; see the Client Libraries section below. Each client library communicates using one of the "transports" that Channels understand. Currently, that's either Websockets or long polling, but other transports may be added in the future.

Unlike stateless HTTP connections, Channels support long-lived connections, each backed by a lightweight BEAM process, working in parallel and maintaining its own state.

This architecture scales well; Phoenix Channels can support millions of subscribers with reasonable latency on a single box, passing hundreds of thousands of messages per second. And that capacity can be multiplied by adding more nodes to the cluster.

The Moving Parts

Although Channels are simple to use from a client perspective, there are a number of components involved in routing messages to clients across a cluster of servers. Let's take a look at them.

Overview

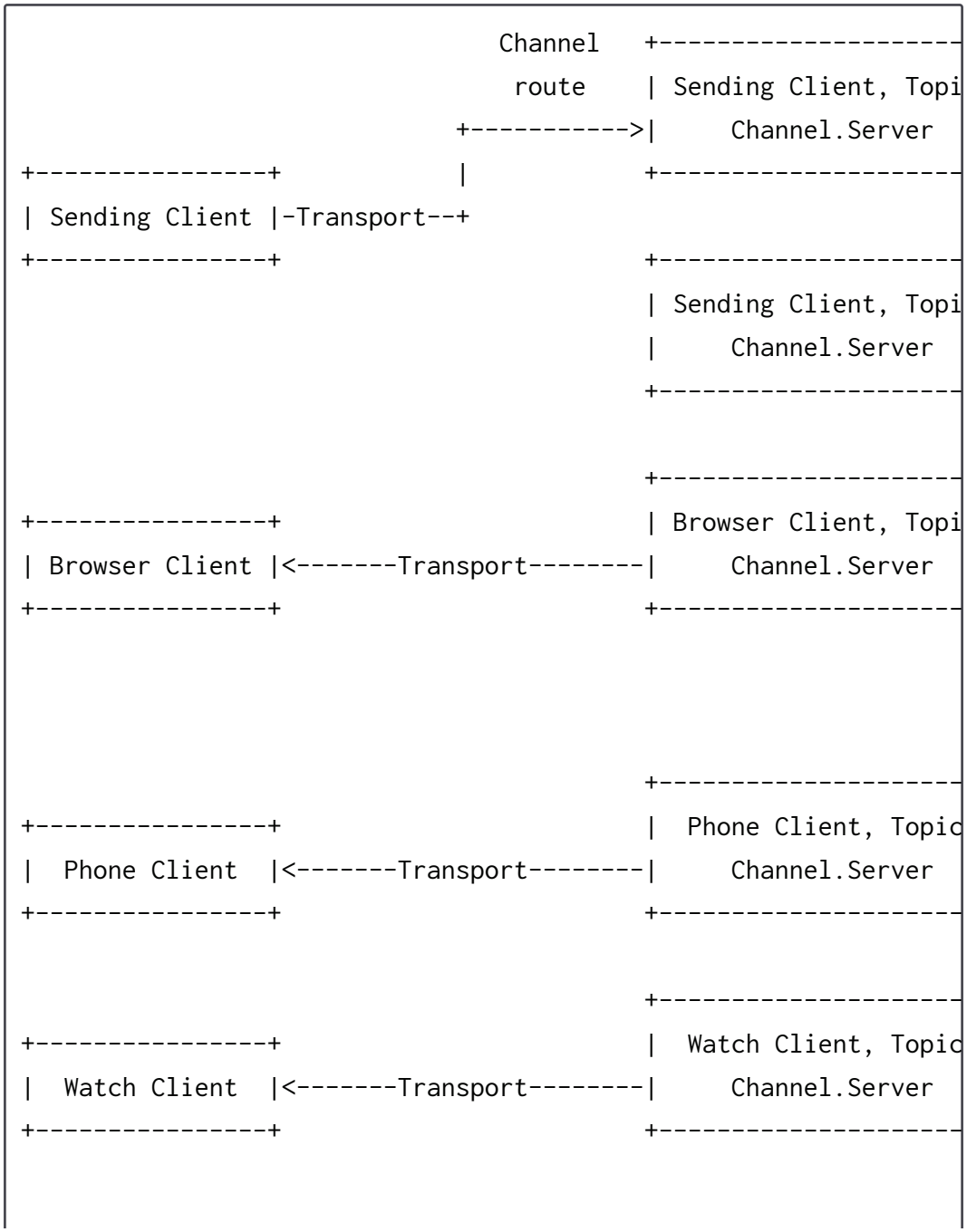
To start communicating, a client connects to a node (a Phoenix server) using a transport (eg, Websockets or long polling) and joins one or more channels using that single network connection. One channel server process is created per client, per topic. The appropriate socket handler initializes a `%Phoenix.Socket` for the channel server (possibly after authenticating the client). The channel server then holds onto the `%Phoenix.Socket{}` and can maintain any state it needs within its `socket.assigns`.

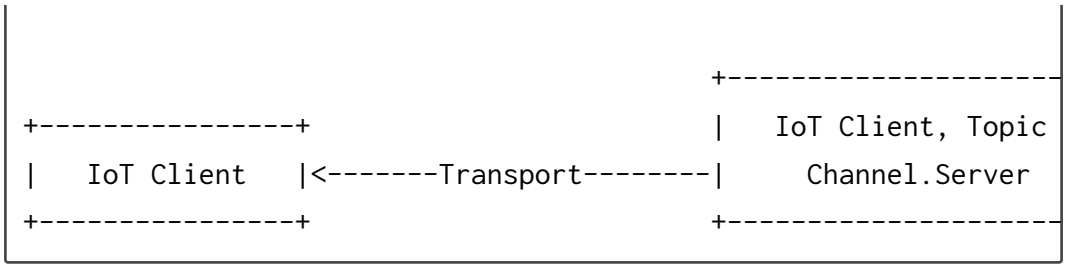
Once the connection is established, each incoming message from a client is routed, based on its topic, to the correct channel server. If the channel server asks to broadcast a message, that message is

sent to the local PubSub, which sends it out to any clients connected to the same server and subscribed to that topic.

If there are other nodes in the cluster, the local PubSub also forwards the message to their PubSubs, which send it out to their own subscribers. Because only one message has to be sent per additional node, the performance cost of adding nodes is negligible, while each new node supports many more subscribers.

The message flow looks something like this:





Endpoint

In your Phoenix app's `Endpoint` module, a `socket` declaration specifies which socket handler will receive connections on a given URL.

```
socket "/socket", HelloWeb.UserSocket,
  websocket: true,
  longpoll: false
```

Phoenix comes with two default transports: websocket and longpoll. You can configure them directly via the `socket` declaration.

Socket Handlers

Socket handlers, such as `HelloWeb.UserSocket` in the example above, are called when Phoenix is setting up a channel connection. Connections to a given URL will all use the same socket handler, based on your endpoint configuration. But that handler can be used for setting up connections on any number of topics.

Within the handler, you can authenticate and identify a socket connection and set default socket assigns.

Channel Routes

Channel routes are defined in socket handlers, such as `HelloWeb.UserSocket` in the example above. They match on the topic string and dispatch matching requests to the given Channel

module.

The star character `*` acts as a wildcard matcher, so in the following example route, requests for `room:lobby` and `room:123` would both be dispatched to the `RoomChannel`.

```
channel "room:*", HelloWeb.RoomChannel
```

Channels

Channels handle events from clients, so they are similar to Controllers, but there are two key differences. Channel events can go both directions - incoming and outgoing. Channel connections also persist beyond a single request/response cycle. Channels are the highest level abstraction for realtime communication components in Phoenix.

Each Channel will implement one or more clauses of each of these four callback functions - `join/3`, `terminate/2`, `handle_in/3`, and `handle_out/3`.

Topics

Topics are string identifiers - names that the various layers use in order to make sure messages end up in the right place. As we saw above, topics can use wildcards. This allows for a useful `"topic:subtopic"` convention. Often, you'll compose topics using record IDs from your application layer, such as `"users:123"`.

Messages

The `Phoenix.Socket.Message` module defines a struct with the following keys which denotes a valid message. From the Phoenix.Socket.Message docs.

- `topic` - The string topic or `"topic:subtopic"` pair namespace,

such as `"messages"` or `"messages:123"`
- `event` - The string event name, for example `"phx_join"`
- `payload` - The message payload
- `ref` - The unique string ref

PubSub

Typically, we don't directly use the Phoenix PubSub layer when developing Phoenix applications. Rather, it's used internally by Phoenix itself. But we may need to configure it.

PubSub consists of the `Phoenix.PubSub` module and a variety of modules for different adapters and their `GenServer`s. These modules contain functions which are the nuts and bolts of organizing Channel communication - subscribing to topics, unsubscribing from topics, and broadcasting messages on a topic.

The PubSub system also takes care of getting messages from one node to another, so that it can be sent to all subscribers across the cluster. By default, this is done using Phoenix.PubSub.PG2, which uses native BEAM messaging.

If your deployment environment does not support distributed Elixir or direct communication between servers, Phoenix also ships with a Redis Adapter that uses Redis to exchange PubSub data. Please see the Phoenix.PubSub docs for more information.

Client Libraries

Any networked device can connect to Phoenix Channels as long as it has a client library. The following libraries exist today, and new ones are always welcome.

Official

Phoenix ships with a JavaScript client that is available when generating a new Phoenix project. The documentation for the

JavaScript module is available at https://hexdocs.pm/phoenix/js/; the code is in phoenix.js.

3rd Party

- Swift (iOS)
  - SwiftPhoenix
- Java (Android)
  - JavaPhoenixChannels
- C#
  - PhoenixSharp
  - dn-phoenix
- Elixir
  - phoenix_gen_socket_client
- GDScript (Godot Game Engine)
  - [GodotPhoenixChannels)(https://github.com/alfredbaudisch/GodotPhoenixChannels)

Tying it all together

Let's tie all these ideas together by building a simple chat application. After generating a new Phoenix application we'll see that the endpoint is already set up for us in `lib/hello_web/endpoint.ex`:

```
defmodule HelloWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :hello

  socket "/socket", HelloWeb.UserSocket
  ...
```

```
end
```

In `lib/hello_web/channels/user_socket.ex`, the `HelloWeb.UserSocket` we pointed to in our endpoint has already been created when we generated our application. We need to make sure messages get routed to the correct channel. To do that, we'll uncomment the "room:*" channel definition:

```
defmodule HelloWeb.UserSocket do
  use Phoenix.Socket

  ## Channels
  channel "room:*", HelloWeb.RoomChannel
  ...
```

Now, whenever a client sends a message whose topic starts with `"room:"`, it will be routed to our RoomChannel. Next, we'll define a `HelloWeb.RoomChannel` module to manage our chat room messages.

Joining Channels

The first priority of your channels is to authorize clients to join a given topic. For authorization, we must implement `join/3` in `lib/hello_web/channels/room_channel.ex`.

```
defmodule HelloWeb.RoomChannel do
  use Phoenix.Channel

  def join("room:lobby", _message, socket) do
    {:ok, socket}
  end
  def join("room:" <> _private_room_id, _params, _socket) do
    {:error, %{reason: "unauthorized"}}
  end
end
```

For our chat app, we'll allow anyone to join the `"room:lobby"` topic, but any other room will be considered private and special authorization, say from a database, will be required. (We won't worry about private chat rooms for this exercise, but feel free to explore after we finish.)

To authorize the socket to join a topic, we return `{:ok, socket}` or `{:ok, reply, socket}`. To deny access, we return `{:error, reply}`. More information about authorization with tokens can be found in the `Phoenix.Token` documentation.

With our channel in place, let's get the client and server talking.

Phoenix projects come with webpack by default, unless disabled with the `--no-webpack` option when you run `mix phx.new`.

The `assets/js/socket.js` defines a simple client based on the socket implementation that ships with Phoenix.

We can use that library to connect to our socket and join our channel, we just need to set our room name to `"room:lobby"` in that file.

```
// assets/js/socket.js
// ...
socket.connect()

// Now that you are connected, you can join channels with a topic
let channel = socket.channel("room:lobby", {})
channel.join()
  .receive("ok", resp => { console.log("Joined successfully", res
  .receive("error", resp => { console.log("Unable to join", resp)

export default socket
```

After that, we need to make sure `assets/js/socket.js` gets imported into our application JavaScript file. To do that,

uncomment the last line in `assets/js/app.js`.

```
// ...
import socket from "./socket"
```

Save the file and your browser should auto refresh, thanks to the Phoenix live reloader. If everything worked, we should see "Joined successfully" in the browser's JavaScript console. Our client and server are now talking over a persistent connection. Now let's make it useful by enabling chat.

In `lib/hello_web/templates/page/index.html.eex`, we'll replace the existing code with a container to hold our chat messages, and an input field to send them:

```
<div id="messages"></div>
<input id="chat-input" type="text"></input>
```

Now let's add a couple of event listeners to `assets/js/socket.js`:

```
// ...
let channel           = socket.channel("room:lobby", {})
let chatInput         = document.querySelector("#chat-input")
let messagesContainer = document.querySelector("#messages")

chatInput.addEventListener("keypress", event => {
  if(event.keyCode === 13){
    channel.push("new_msg", {body: chatInput.value})
    chatInput.value = ""
  }
})

channel.join()
  .receive("ok", resp => { console.log("Joined successfully", res
  .receive("error", resp => { console.log("Unable to join", resp)
```

```
export default socket
```

All we had to do is detect that enter was pressed and then `push` an event over the channel with the message body. We named the event `"new_msg"`. With this in place, let's handle the other piece of a chat application where we listen for new messages and append them to our messages container.

```
// ...
let channel           = socket.channel("room:lobby", {})
let chatInput         = document.querySelector("#chat-input")
let messagesContainer = document.querySelector("#messages")

chatInput.addEventListener("keypress", event => {
  if(event.keyCode === 13){
    channel.push("new_msg", {body: chatInput.value})
    chatInput.value = ""
  }
})

channel.on("new_msg", payload => {
  let messageItem = document.createElement("li")
  messageItem.innerText = `[${Date()}] ${payload.body}`
  messagesContainer.appendChild(messageItem)
})

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

export default socket
```

We listen for the `"new_msg"` event using `channel.on`, and then append the message body to the DOM. Now let's handle the incoming and outgoing events on the server to complete the picture.

Incoming Events

We handle incoming events with `handle_in/3`. We can pattern match on the event names, like `"new_msg"`, and then grab the payload that the client passed over the channel. For our chat application, we simply need to notify all other `room:lobby` subscribers of the new message with `broadcast!/3`.

```
defmodule HelloWeb.RoomChannel do
  use Phoenix.Channel

  def join("room:lobby", _message, socket) do
    {:ok, socket}
  end
  def join("room:" <> _private_room_id, _params, _socket) do
    {:error, %{reason: "unauthorized"}}
  end

  def handle_in("new_msg", %{"body" => body}, socket) do
    broadcast!(socket, "new_msg", %{body: body})
    {:noreply, socket}
  end
end
```

`broadcast!/3` will notify all joined clients on this `socket`'s topic and invoke their `handle_out/3` callbacks. `handle_out/3` isn't a required callback, but it allows us to customize and filter broadcasts before they reach each client. By default, `handle_out/3` is implemented for us and simply pushes the message on to the client, just like our definition. We included it here because hooking into outgoing events allows for powerful message customization and filtering. Let's see how.

Intercepting Outgoing Events

We won't implement this for our application, but imagine our chat

app allowed users to ignore messages about new users joining a room. We could implement that behavior like this where we explicitly tell Phoenix which outgoing event we want to intercept and then define a `handle_out/3` callback for those events. (Of course, this assumes that we have a `Accounts` context with an `ignoring_user?/2` function, and that we pass a user in via the `assigns` map). It is important to note that the `handle_out/3` callback will be called for every recipient of a message, so more expensive operations like hitting the database should be considered carefully before being included in `handle_out/3`.

```
intercept ["user_joined"]

def handle_out("user_joined", msg, socket) do
  if Accounts.ignoring_user?(socket.assigns[:user], msg.user_id)
    {:noreply, socket}
  else
    push(socket, "user_joined", msg)
    {:noreply, socket}
  end
end
```

That's all there is to our basic chat app. Fire up multiple browser tabs and you should see your messages being pushed and broadcasted to all windows!

Socket Assigns

Similar to connection structs, `%Plug.Conn{}`, it is possible to assign values to a channel socket. `Phoenix.Socket.assign/3` is conveniently imported into a channel module as `assign/3`:

```
socket = assign(socket, :user, msg["user"])
```

Sockets store assigned values as a map in `socket.assigns`.

Using Token Authentication

When we connect, we'll often need to authenticate the client. Fortunately, this is a 4-step process with Phoenix.Token.

Step 1 - Assign a Token in the Connection

Let's say we have an authentication plug in our app called `OurAuth`. When `OurAuth` authenticates a user, it sets a value for the `:current_user` key in `conn.assigns`. Since the `current_user` exists, we can simply assign the user's token in the connection for use in the layout. We can wrap that behavior up in a private function plug, `put_user_token/2`. This could also be put in its own module as well. To make this all work, we just add `OurAuth` and `put_user_token/2` to the browser pipeline.

```
pipeline :browser do
  ...
  plug OurAuth
  plug :put_user_token
end

defp put_user_token(conn, _) do
  if current_user = conn.assigns[:current_user] do
    token = Phoenix.Token.sign(conn, "user socket", current_user
    assign(conn, :user_token, token)
  else
    conn
  end
end
```

Now our `conn.assigns` contains the `current_user` and `user_token`.

Step 2 - Pass the Token to the JavaScript

Next we need to pass this token to JavaScript. We can do so inside a script tag in `web/templates/layout/app.html.eex` right above the app.js script, as follows:

```
<script>window.userToken = "<%= assigns[:user_token] %>";</script
<script src="<%= Routes.static_path(@conn, "/js/app.js") %>"></sc
```

Step 3 - Pass the Token to the Socket Constructor and Verify

We also need to pass the `:params` to the socket constructor and verify the user token in the `connect/3` function. To do so, edit `web/channels/user_socket.ex`, as follows:

```
def connect(%{"token" => token}, socket, _connect_info) do
  # max_age: 1209600 is equivalent to two weeks in seconds
  case Phoenix.Token.verify(socket, "user socket", token, max_ag
    {:ok, user_id} ->
      {:ok, assign(socket, :current_user, user_id)}
    {:error, reason} ->
      :error
  end
end
```

In our JavaScript, we can use the token set previously when to pass the token when constructing the Socket:

```
let socket = new Socket("/socket", {params: {token: window.userTo
```

We used `Phoenix.Token.verify/4` to verify the user token provided by the client. `Phoenix.Token.verify/4` returns either `{:ok, user_id}` or `{:error, reason}`. We can pattern match on that return in a `case` statement. With a verified token, we set the user's id as the value to `:current_user` in the socket. Otherwise, we return `:error`.

Step 4 - Connect to the socket in JavaScript

With authentication set up, we can connect to sockets and channels from JavaScript.

```
let socket = new Socket("/socket", {params: {token: window.userTo
```

```
socket.connect()
```

Now that we are connected, we can join channels with a topic:

```
let channel = socket.channel("topic:subtopic", {})
channel.join()
  .receive("ok", resp => { console.log("Joined successfully", re
  .receive("error", resp => { console.log("Unable to join", resp

export default socket
```

Note that token authentication is preferable since it's transport agnostic and well-suited for long running-connections like channels, as opposed to using sessions or authentication approaches.

Fault Tolerance and Reliability Guarantees

Servers restart, networks split, and clients lose connectivity. In order to design robust systems, we need to understand how Phoenix responds to these events and what guarantees it offers.

Handling Reconnection

Clients subscribe to topics, and Phoenix stores those subscriptions in an in-memory ETS table. If a channel crashes, the clients will need to reconnect to the topics they had previously subscribed to. Fortunately, the Phoenix JavaScript client knows how to do this. The server will notify all the clients of the crash. This will trigger each client's `Channel.onError` callback. The clients will attempt to reconnect to the server using an exponential back off strategy. Once they reconnect, they'll attempt to rejoin the topics they had previously subscribed to. If they are successful, they'll start receiving messages from those topics as before.

Resending Client Messages

Channel clients queue outgoing messages into a `PushBuffer`, and send them to the server when there is a connection. If no connection is available, the client holds on to the messages until it can establish a new connection. With no connection, the client will hold the messages in memory until it establishes a connection, or until it receives a `timeout` event. The default timeout is set to 5000 milliseconds. The client won't persist the messages in the browser's local storage, so if the browser tab closes, the messages will be gone.

Resending Server Messages

Phoenix uses an at-most-once strategy when sending messages to clients. If the client is offline and misses the message, Phoenix won't resend it. Phoenix doesn't persist messages on the server. If the server restarts, unsent messages will be gone. If our application needs stronger guarantees around message delivery, we'll need to write that code ourselves. Common approaches involve persisting messages on the server and having clients request missing messages. For an example, see Chris McCord's Phoenix training: client code and server code.

Presence

Phoenix ships with a way of handling online users that is built on top of Phoenix.PubSub and Phoenix channels. The usage of presence is covered in the presence guide.

Example Application

To see an example of the application we just built, checkout the project phoenix_chat_example.

You can also see a live demo at http://phoenixchat.herokuapp.com/.