Extracted from:

Real-Time Phoenix

Build Highly Scalable Systems with Channels

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Real-Time Phoenix

Build Highly Scalable Systems with Channels

Stephen Bussey

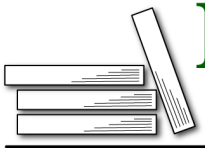
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Real-Time Phoenix

Build Highly Scalable Systems with Channels

Stephen Bussey

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Sean Dennis
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-719-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2020

Use Channels in a Cluster

It is critical to run multiple servers when you are deploying a production application. Doing so provides benefits for scalability and error tolerance. For example, the ability to double the number of servers in the event of higher load is much more powerful than doubling the number of cores on the single server. It can take a few minutes (or less!) to add more machines but could take much longer to move the application to a different machine with more cores. There may also be a time when a single machine is fully utilized, and you cannot add more CPU cores or memory.

Elixir makes connecting a cluster of BEAM nodes very easy. However, we have to ensure that we're building our application to run across multiple nodes without error. Phoenix Channels handles a lot of this for us due to PubSub being used for all message broadcasts, which we'll look at next.

Connecting a Local Cluster

Let's jump right in by starting a local Elixir node (instance of our application) with a name:

```
$ iex --name server@127.0.0.1 -S mix phx.server
[info] Access HelloSocketsWeb.Endpoint at http://localhost:4000
iex(server@127.0.0.1)1>
```

We use the `--name` switch to specify a name for our node. You can see the name on the input entry line; ours is located at `server@127.0.0.1`. Let's start a second node:

```
$ iex --name remote@127.0.0.1 -S mix
Interactive Elixir (1.6.6) - press Ctrl+C to exit (type h() ENTER for help)
iex(remote@127.0.0.1)1> Node.list()
[]
```

We started a second node that doesn't run a web server by starting `mix` instead of `mix phx.server`. We used a different name, `remote@127.0.0.1`, which gives us two nodes running on the same host domain. You can use `Node.list/0` to view all currently connected nodes and see that there are none. Let's correct that:

```
iex(remote@127.0.0.1)1> Node.list()
[]
iex(remote@127.0.0.1)2> Node.connect(:"server@127.0.0.1")
true
iex(remote@127.0.0.1)3> Node.list()
[:"server@127.0.0.1"]
```

We run `Node.connect/1` from our remote node to connect to the server node. This creates a connected cluster of nodes that can be verified by running `Node.list/0`

again. Try running Node.list/0 on the server node; you will see it contains the remote node name.

This is all that we have to do to take advantage of Phoenix PubSub's standard distribution strategy powered by pg2. We can broadcast a message from our remote node, which is incapable of serving Sockets, and see it on a client that is connected to a Socket on our main server. Let's try this out:

First, connect to the ping topic to establish the connection.

```
$ wscat -c 'ws://localhost:4000/socket/websocket?vsn=2.0.0'
> ["1","1","ping","phx_join",{}]
< ["1","1","ping","phx_reply",{"response":{},"status":"ok"}]
```

Next, broadcast a message *from the remote node*.

```
iex(r@127)> HelloSocketsWeb.Endpoint.broadcast("ping", "request_ping", %{})
:ok
```

Finally, you can see that the ping request made it to the client:

```
< [null,null,"ping","send_ping",{"from_node":"server@127.0.0.1"}]
```

The node that sent the message to the client is server@127.0.0.1, but we sent our broadcast from remote@127.0.0.1. This means that the message was distributed across the cluster and intercepted by the PingChannel on our server node.

This demo shows that we can have a message originate anywhere in our cluster, and the message will make it to the client. This is critical for a correctly working application that runs on multiple servers, and we get it for very low cost by using Phoenix PubSub.

In practice, our remote node would be serving Socket connections, and the entire system would be placed behind a tool that balances connections between the different servers. You could emulate this locally by changing the HTTP port in the application configuration, then connecting to the new port with wscat.

hello_sockets/config/dev.exs

```
config :hello_sockets, HelloSocketsWeb.Endpoint,
  http: [port: String.to_integer(System.get_env("PORT") || "4000")],
```

You can now start the remote server in HTTP serving mode by prepending PORT=4001 to the command. You will need to restart the original server@127.0.0.1 server as well.

```
$ PORT=4001 iex --name remote@127.0.0.1 -S mix phx.server
[info] Running Web.Endpoint with cowboy 2.6.3 at 0.0.0.0:4001 (http)
[info] Access Web.Endpoint at http://localhost:4001
iex(remote@127.0.0.1)1>
```

You can experiment with sending messages between the different nodes to confirm that they are delivered in either direction. You'll learn about cluster deployments in greater detail in Chapter 11, Deploy Your Application to Production, on page ?.

Channel distribution is very powerful and easy to get started with out-of-the-box. However, there are some challenges with it, which we'll explore next.

Challenges with Distributed Channels

Distribution provides immense benefits to the scalability of our application, but it comes with costs as well. A distributed application has potential problems that a single-node application won't experience. A single-node application may be the right call in some circumstances, such as a small internal application, but we often must deliver our applications to many users that require the performance and stability that are provided by distribution.

Here are a few of the challenges that we'll face when distributing our application. These problems are not specific to Elixir—you would experience the same problems when building a distributed system in any language.

- We cannot be sure that we have fully accurate knowledge of the state of remote nodes at any given time. We can use techniques and algorithms to reduce uncertainty, but not completely remove it.
- Messages may not be transmitted to a remote node as fast as we'd expect, or at all. It may be fairly rare for messages to be dropped completely, but message delays are much more common.
- Writing high-quality tests becomes more complicated as we have to spin up more complex scenarios to fully test our code. It is possible to write tests in Elixir that spin up a local cluster to simulate different environments.
- Our clients may disconnect from a node and end up on a different node with different internal state. We must accommodate this by having a central source of truth that any node can reference; this is most commonly a shared database.

The easiest principle to get started with is having a central source of truth that all nodes can read from when a process, such as a Channel, starts. We will use this technique throughout the book. The other challenges involve using proven data structures and algorithms for key tasks of our distributed application. In part II, you'll learn about Phoenix Tracker for distributed

process tracking, and you have already learned about PubSub's mesh approach to message broadcasting.

Let's look at different ways to customize Channel behavior. These exercises get into a bit more code than we've seen so far, which makes them quite fun!

Customize Channel Behavior

A Phoenix Channel is backed by a `GenServer` that lets it receive messages and store state. We can take advantage of this property of Channels to customize the behavior of our Channel on a per-connection level. This allows us to build flows that are not possible (or would be much more complex) with standard message broadcasting, which can't easily send messages to a single client.

We can't customize the behavior of Sockets as much due to their process structure. We'll focus our attention strictly on Channel-level customization for these examples by walking through several different patterns that use `Phoenix.Socket.assign/3` and message sending.

Send a Recurring Message

We sometimes need to send data to a client in a periodic way. One use case of this is to refresh an authentication token every few minutes to ensure that a client always has a valid token. This is useful because it is possible to overwhelm a server if all clients ask for a token at the same time.

Our Channel will send itself a message every five seconds by using `Process.send_after/3`. This flow will be started when the Channel process initializes, but it would be possible to start the flow in our `handle_in` callback as well, in response to a client-initiated message.

First, add a new "recurring" Channel route to the AuthSocket module.

hello_sockets/lib/hello_sockets_web/channels/auth_socket.ex
```
channel "recurring", HelloSocketsWeb.RecurringChannel
```

This Channel route makes our new Channel available. Let's create the RecurringChannel.

hello_sockets/lib/hello_sockets_web/channels/recurring_channel.ex
```
defmodule HelloSocketsWeb.RecurringChannel do
  use Phoenix.Channel

  @send_after 5_000

  def join(_topic, _payload, socket) do
    schedule_send_token()
    {:ok, socket}
  end

  defp schedule_send_token do
    Process.send_after(self(), :send_token, @send_after)
  end
end
```

We leverage our join callback in order to schedule a message to self() for five seconds in the future. This starts a timer that will cause the message :send_token to be delivered. Now, let's define the :send_token message handler.

hello_sockets/lib/hello_sockets_web/channels/recurring_channel.ex
```
def handle_info(:send_token, socket) do
  schedule_send_token()
  push(socket, "new_token", %{token: new_token(socket)})
  {:noreply, socket}
end

defp new_token(socket = %{assigns: %{user_id: user_id}}) do
  Phoenix.Token.sign(socket, "salt identifier", user_id)
end
```

We use handle_info/2, as we would in a standard GenServer, to handle the :send_token message. The first thing we do is schedule another message so the flow will run forever. We then use push/3 to send a newly signed Phoenix.Token to the client.

The Socket.assigns.user_id property set in AuthSocket.connect/2 provides the user information needed when we sign our token. Socket.assigns is a great way to bridge the gap between the initial connection and ongoing business logic, as it allows us to pass information that was initially provided in the connection request to the Channel.

Now let's add a subscription to RecurringChannel in our JavaScript:

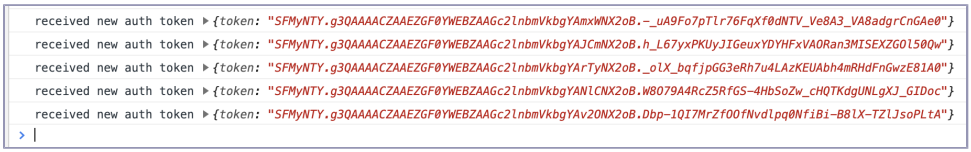
hello_sockets/assets/js/socket.js

```
const recurringChannel = authSocket.channel("recurring")

recurringChannel.on("new_token", (payload) => {
  console.log("received new auth token", payload)
})

recurringChannel.join()
```

We are using our JavaScript client to observe this example, as we previously configured it to connect to the AuthSocket. Refresh your web page to see that the client is receiving a new unique token every five seconds. You will see log statements in your console, like this:



This will continue forever because our Channel does not have any logic to stop it. We are sending a message every five seconds for observation purposes, but we would normally set this duration closer to the expiration time of our token.

Let's look at a more advanced Channel customization that intercepts outgoing messages.