What did I learn

About Tags

Using channels in Phoenix

MARCH 4, 2018

In the previous articles, we have build interface for our chat application. We have also implemented the authentication functionality so our users can register and login to the app. Now it is time to implement the main feature of our application. The communication between users. We are going to cover the basics of working with Phoenix Channels.

Channels in Phoenix one of the coolest features. Even though that is not that hard to use.

Channels bring the interactive functionality to the web page. Usually, when you load the page, you are sending a request to a server and the server responds back with the HTML template. Once the rendering of the page is done, a server has no way to communicate back to your browser.

The things are changed with channels. When you load the page, it can establish a connection between a client (your browser) and a server. Once the connection is established, they can send messages back and forth.

By default, Phoenix uses WebSockets as a transport protocol. But you can also use Long Polling instead.

Today we are going to implement the chatting functionality in our project. So let’s get started.

Establish a connection

To implement channel functionality we need to work on both sides. For the client, it would be a JavaScript and the related to sockets `assets/js/socket.js` file.

For the server, we are going to work in `lib/prater_web/channels/` directory.

So first things first.

The `assets/js/socket.js` file already contains some implementation for us, which we need to update for our needs.

First, we change the channel name on the line

```
let channel = socket.channel("topic:subtopic'
```

to be

```
let channel = socket.channel("room:lobby", {]
```

If we look at the code in that file, we can see how that `channel` object is used.

```
channel.join()
  .receive("ok", resp => { console.log("Joine
  .receive("error", resp => { console.log("Ur
```

The code calls the `join()` function at the `channel` and then waits for a response. For now, in both cases, it outputs the result back in the browser console. Which is absolutely great for as at the moment, and even can work well for us.

If we take a look at the comment at the very of that file

```
// NOTE: The contents of this file will only
// you uncomment its entry in "assets/js/app.
```

We can see we need to enable sockets functionality on the client.

So let's open that file and uncomment the following line

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

That is actually all preparations related to the client side. Let's move on to the server side.

We already have a `lib/prater_web/channels/user_socket.ex` file in place. That file is an entry point for all the socket connections. We configure the basic settings there.

Let's uncomment the following line there.

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

The `"room:*"` is the event name and `RoomChannel` is the module which would handle incoming events. You can name it differently in your projects, but in our case, the name "Room" works great. So I will keep that.

We have linked a channel with the `RoomChannel` module, which we don't have yet. So let's create it with the following content: In `lib/prater_web/channels/room_channel.ex`

```
defmodule PraterWeb.RoomChannel do
  use PraterWeb, :channel

  def join(channel_name, _params, socket) do
    {:ok, %{channel: channel_name}, socket}
  end
end
```

We have a module and we injecting a channel related functionality into it.

Then we have defined the `join/3` callback, which receives: Channel name, additional params, and a socket struct. The callback is responsible for handling incoming connections. It also have to return back a tuple with one of the following formats:

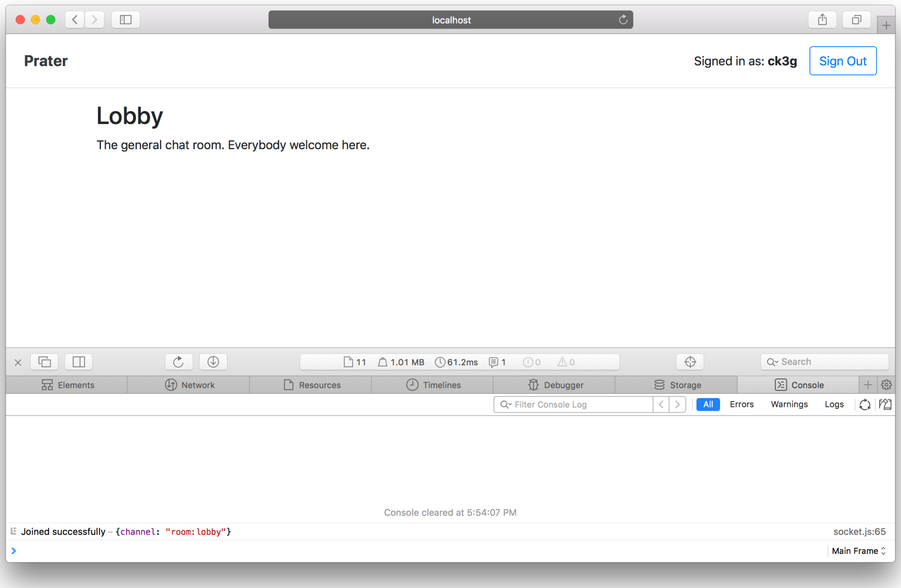
```
{:ok, Phoenix.Socket.t()}
{:ok, map(), Phoenix.Socket.t()}
{:error, map()}
```

So we can respond with OK and a socket struct. We can respond with OK, return additional information and then return the socket struct. Or we can respond with an error.

We are using the second approach mostly to test it out.

That is pretty much we need to join the channel.

If we move to any room page and refresh the page with developer tools opened, we can see a `console.log` message with the information we have responded back.



Lobby

The general chat room. Everybody welcome here.

```
Console cleared at 5:54:07 PM
Joined successfully – {channel: "room:lobby"}                socket.js:65
```

Sending messages

Since we have the connection established we can start sending messages to the socket.

Let's create a form with a text input and a button, which we will use for sending messages.

Paste the following markup in the `lib/prater_web/templates/room/show.html.eex` file.

```
<ul class="list-group mt-5" id="messages"></u

<form class="input-group mt-3" id="new-messag
  <input type="text" class="form-control" pla
  <div class="input-group-append">
    <button class="btn btn-outline-success" t
  </div>
</div>
```

We have also described a container for our future messages.

Now in the `assets/js/socket.js` file, we need to handle submission of the form.

```
document.querySelector("#new-message").addEve
  e.preventDefault()
  let messageInput = e.target.querySelector('

  channel.push('message:add', { message: mess

  messageInput.value = ""
});
```

We grab the value of the text input, we push it to the channel and then remove the value from the input.

We can try to send a message now. If the check the server log we would see that we have an incoming message:

```
[debug] INCOMING "room:lobby" on "room:lobby'
  Transport:  Phoenix.Transports.WebSocket
  Parameters: %{"message" => "Hello there!"}
```

So it goes to the server, but nobody handles it.

To get the message on the server side, we need to implement the following callback in the `RoomChannel` module.

```
def handle_in("message:add", %{"message" => c
  broadcast!(socket, "room:lobby:new_message'
  {:reply, :ok, socket}
end
```

As the first argument, we are matching the event name with the event we were using while sending that message. As the second argument, we grab the content of the message. The third argument contains the socket struct.

If we have several connections established with the server and someone sends a message, we definitely want all other users to get that message. That is why the `broadcast!` function is used here.

We are broadcasting a new event `"room:lobby:new_message"` to a `socket` with the Map `%{content: content}`. So everyone who subscribed gets the message.

Now, on the client side, we need to listen to those events and do something with them.

We can achieve that with the following functionality:

```
channel.on("room:lobby:new_message", (message
  console.log("message", message)
});
```

We are listening for incoming messages and then log the content in the browser's console.

Try it. We are getting those messages there.

Let's proceed and display it on the page.

```
channel.on("room:lobby:new_message", (message
  console.log("message", message)
  renderMessage(message)
});

const renderMessage = function(message) {
  let messageTemplate = `
```

```
      <li class="list-group-item">${message.cor
    `
    document.querySelector("#messages").innerHT
  };
```

Once we get the new message we are building a message template and add it to our messages container.

Now if you check it, new messages should appear on the page.

Split the conversation into rooms

At this moment you can notice what it does not matter in which room you are located right now you will get the same conversation. That happens because we have our channel name hardcoded by now.

What we need to do instead is to use different channels for different rooms.

We need to identify where is user located by now. Let's pass the room ID into our client-side code.

Add the following line to the `lib/prater_web/templates/room/show.html.eex`

```
<script>window.channelRoomId = "<%= @room.id
```

Then in the `assets/js/socket.js` let's grab it:

```
let channelRoomId = window.channelRoomId
```

and pass to channel room. Change

```
let channel = socket.channel("room:lobby", {}
```

to

```
let channel = socket.channel(`room:${channelR
```

We also need to wrap joining logic into `if` statement. In case we are not in the room's page.

```
if (channelRoomId) {
}
```

To void confusion, here is the complete piece of the code of how should it look like:

```
let channelRoomId = window.channelRoomId;

if (channelRoomId) {
  // Now that you are connected, you can joir
  let channel = socket.channel(`room:${channe
  channel.join()
    .receive("ok", resp => { console.log("Joi
    .receive("error", resp => { console.log('

  channel.on("room:lobby:new_message", (messa
    console.log("message", message)
    renderMessage(message)
  });

  document.querySelector("#new-message").addE
    e.preventDefault()
    let messageInput = e.target.querySelector
```

```
      channel.push('message:add', { message: me

      messageInput.value = ""
    });
  }
```

Now if we refresh the page, we can see that we are successfully joined to “room:1” channel. `1` in that case is the ID of the room. It would wary from room to room.

I’m sure there are better ways to organize that code, but just to keep the changes at minimum let’s leave it like that for now.

We still have a hardcoded event name `"room:lobby:new_message"`. Let’s fix that as well.

First, we are going to change that event name to

```
`room:${channelRoomId}:new_message`
```

The next part would be to navigate to `lib/prater_web/channels/room_channel.ex` and work on the code there.

We have a `handle_in/3` function which uses that event. But the function has no clue about any Room ID.

What do we do?

We know that we are passing right Room ID into the `join` function. So we can extract it from there.

Let’s change the function in the following way:

```
def join("room:" <> room_id, _params, socket)
  {:ok, %{channel: channel_name}, socket}
end
```

You might be wondering, what does the following expression mean `"room:" <> room_id`? That is the power of pattern matching.

The `<>` sign works as a string concatenation in Elixir. We also know that we can provide two expressions on the left and right hand of `=` operator to match some values into variables.

Let's jump to `iex` and see it works. Basically we have the following expression:

```
iex> "room:" <> room_id = "room:503"
```

We can see that expression on the left side matches with the string on the right side. They have a similar pattern. The only missing piece is a string right after colon sign. So now if we check the value of `room_id` we would see it contains a string "503".

```
iex> room_id
"503"
```

That means our `join/3` function can grab a room ID now.

So what do we do next? We still need to pass it down to `handle_in/3` function.

The socket struct, which we are receiving as a third argument an passing further, is very similar to connection struct which we have in the Phoenix. That means we can extend it with additional information. In our case, we can assign a room ID.

```
def join("room:" <> room_id, _params, socket)
  {:ok, %{channel: "room:#{room_id}"}, assigr
end
```

Now in the `handle_in/3` function we fetch it and pass into event name while broadcasting a message:

```
def handle_in("message:add", %{"message" => (
  room_id = socket.assigns[:room_id]
  broadcast!(socket, "room:#{room_id}:new_mes
  {:reply, :ok, socket}
end
```

Check it, now you will receive messages only related to the room you are in.

Socket authentication

Now our users can send messages. Those messages are split into rooms. But we can't see who exactly is sending those messages. It would be nice to see a username in front of every message.

Our socket should somehow know about a user as well.

If we look at the top of the `assets/js/socket.js` file, we can see how our `socket` object is being initialized.

```
let socket = new Socket("/socket", {params: {
```

It expects the user token to be passed from `window.userToken` variable.

So first, we need to define that variable.

In the `lib/prater_web/templates/layout/app.html.eex` file, right before the `<script>` tag, we are using to add `js/app.js`, we need to add the following line.

```
<script>window.userToken = "<%= assigns[:user
<script src="<%= static_path(@conn, "/js/app.
```

It grabs the values from assigned variables to our connection. Which we don't have yet.

Where do we do that? We already have a `SetCurrentUser` plug, where we assign `:current_user`, so it seems to be a right place.

We need to update the code, sign the token and assign it to the connection. The `call` function should look like:

```
def call(conn, _params) do
  user_id = Plug.Conn.get_session(conn, :curr

  cond do
    current_user = user_id && Repo.get(User,
      token = Phoenix.Token.sign(conn, "user
      conn
      |> assign(:current_user, current_user)
      |> assign(:user_signed_in?, true)
      |> assign(:user_token, token)
    true ->
      conn
      |> assign(:current_user, nil)
      |> assign(:user_signed_in?, false)
  end
end
```

Now, any params we are passing into a socket initialization

```
let socket = new Socket("/socket", {params: {
```

we can grab in the `connect/2` function of the `UserSocket` module.

```
  def connect(_params, socket) do
    {:ok, socket}
  end
```

Here we need to retrieve the token, check if it is valid and assign the user ID to the socket. We also need to respond with an error if the token is not valid.

Let's extend our `connect/2` function

```
  @max_age 24 * 60 * 60
  def connect(%{"token" => token}, socket) do
    case Phoenix.Token.verify(socket, "user tok
      {:ok, user_id} ->
        {:ok, assign(socket, :current_user_id,
      {:error, _reason} ->
        :error
    end
  end

  def connect(_params, _socket), do: :error
```

We have set the max age of the token to avoid it to be valid forever.

We also keep another version of the function just in case if there is no token provided.

Ok, now we have a current user ID assigned to the socket. We can fetch it and pass into a message map:

```
  alias Prater.Repo
  alias Prater.Auth.User

  def handle_in("message:add", %{"message" => c
    room_id = socket.assigns[:room_id]
    user = Repo.get(User, socket.assigns[:curre
    message = %{content: content, user: %{userr

    broadcast!(socket, "room:#{room_id}:new_mes

    {:reply, :ok, socket}
  end
```

The last piece we need to do is to update our message template in the `assets/js/socket.js`

```
  const renderMessage = function(message) {
    let messageTemplate = `
      <li class="list-group-item">
        <strong>${message.user.username}</stror
        ${message.content}
      </li>
    `
    document.querySelector("#messages").innerHT
  };
```

The complete demo of the whole article you can see below:

Wrapping up

We just covered the basics of working with Phoenix Channels. We have a working solution which allows our users to communicate with each other. All the communication is split into rooms and we can see the username of every user who sends a message.

Of course, we have a bunch of missing stuff. We are not persisting those messages in the DataBase. If we reload the page the conversation is gone. We will proceed with those improvements in the next articles.

The complete implementation you can find on GitHub page.

See you soon.

Elixir Phoenix Channels

Like it? Share it.

Y Buy me a coffee

Subscribe to get updates email address

Subscribe

Powered by Jekyll with Type Theme

0 Comments **What did I learn** 1 **Login**

Recommend 3 Tweet Share Sort by Best

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON **WHAT DID I LEARN**

Introduction to OTP, GenServers and

4 comments • 2 years ago

Avatarxavvvier — Hi Vitaly,I think you have an error when you start to define the

React Native: How to use FlatList

1 comment • a year ago

AvatarRamiro Decono — Very concise and clear information. Thanks!