Distributed Systems Project 4- Part II
Twitter Simulator using Phoenix-Elixir

Group Details:

Members
1: Manjary Modi ,UFID: 38408368, manjary.modi@ufl.edu
2: Rameshwari Obla Ravikumar,UFID: 16161302, rameshwari.oblar@ufl.edu

Steps to run the application:

1. In shell, navigate to the directory where mix.exs exists
2. To deploy the application, run the following commands and ignore any warnings:

[To get all the dependencies of the application]
>mix deps.get
[To compile the dependencies]
>mix deps.compile
[To get node_modules]
>npm install
[To create the database]
>mix ecto.create
[To migrate the database]
>mix ecto.migrate
[To run the application]
>mix phoenix.server

The phoenix server will get started at http://localhost:4000. If it does not get started automatically, paste the link that is displayed in the console to your browser.

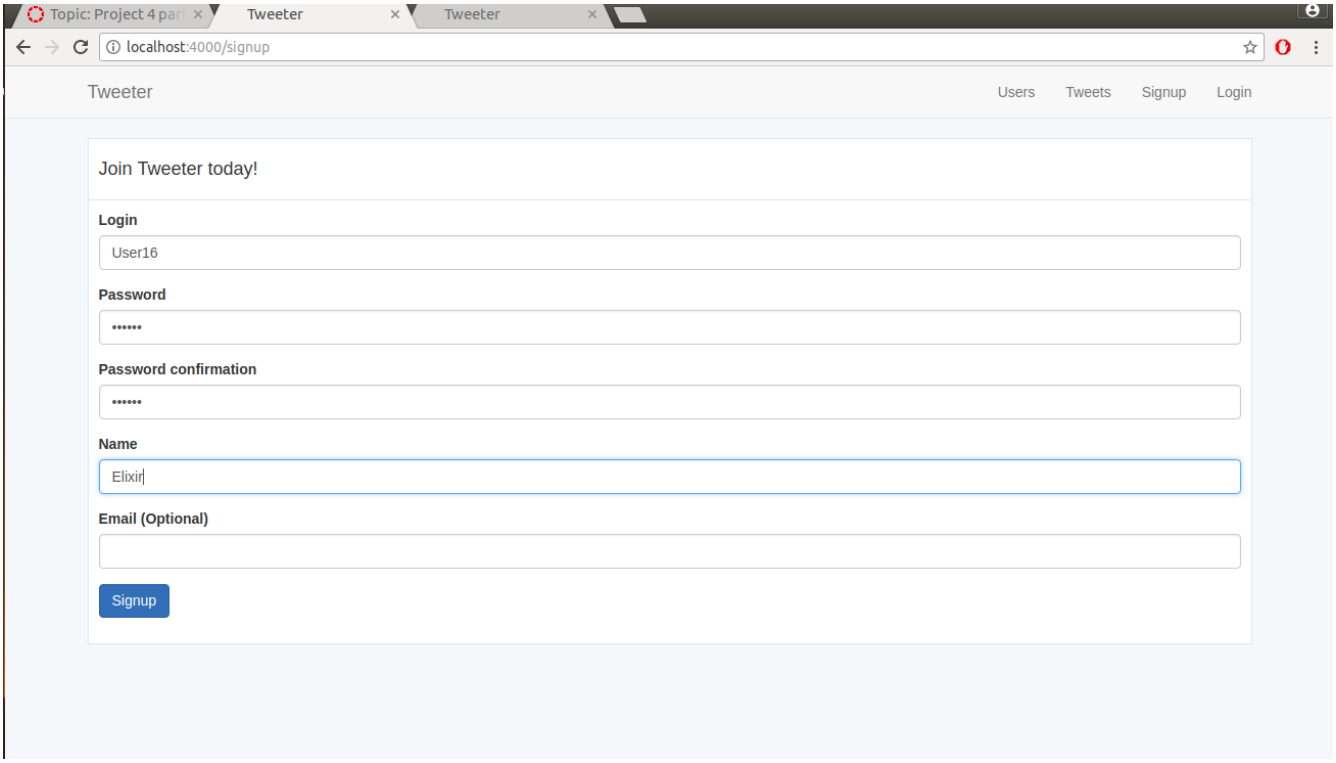
Some of the specific modules that were used in the application are:
1. Comeonin – Login Authentication
2. Presence – Real-time user tracking

Tweeter Implementation:

Tweeter(our twitter simulator) server engine runs on phoenix web framework and implements a web socket interface with clients. We have implemented client interface using UI and below are some cool functionalities of Tweeter.

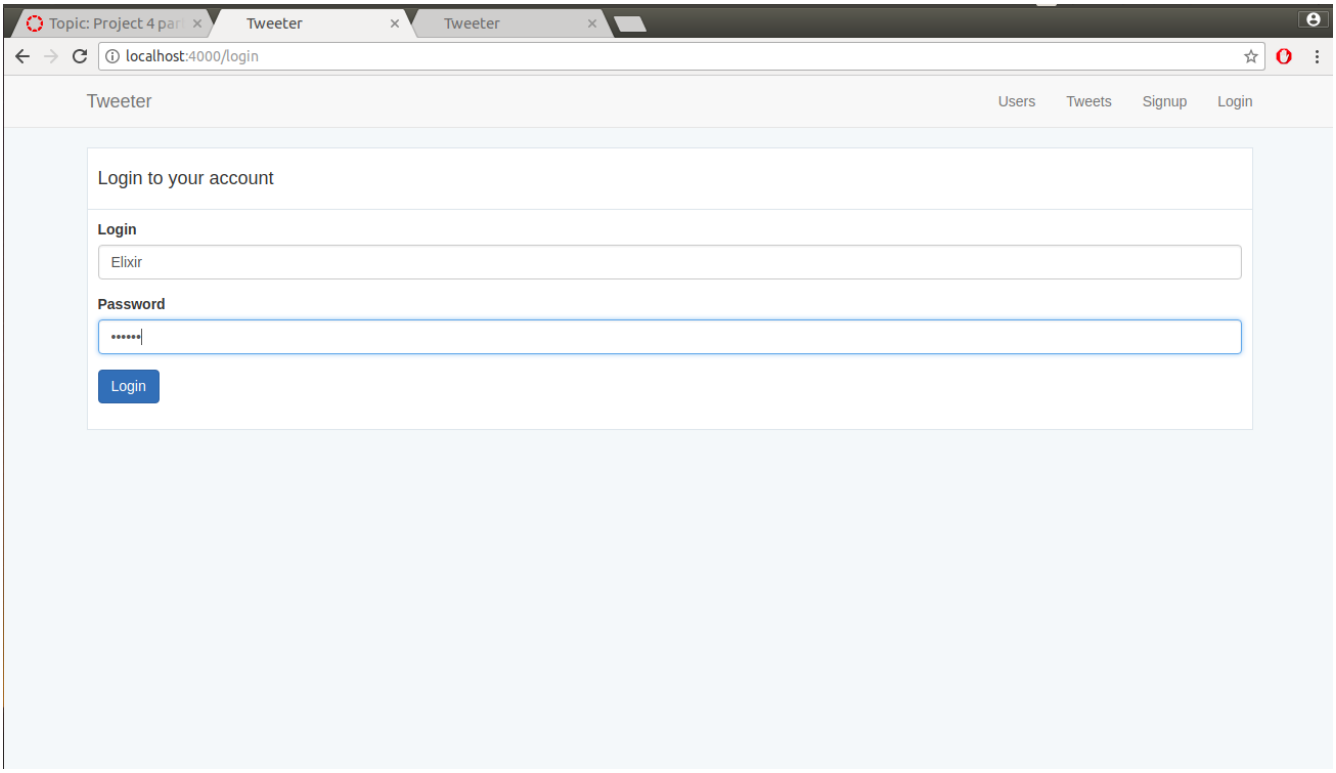
1. Signup

A new user can sign up to be a member of tweeter by using this functionality. The user has to provide a login name, password and Original Name. We have implemented authentication for password using comeonin module and password is also hashed for security purpose using Bcrypt.



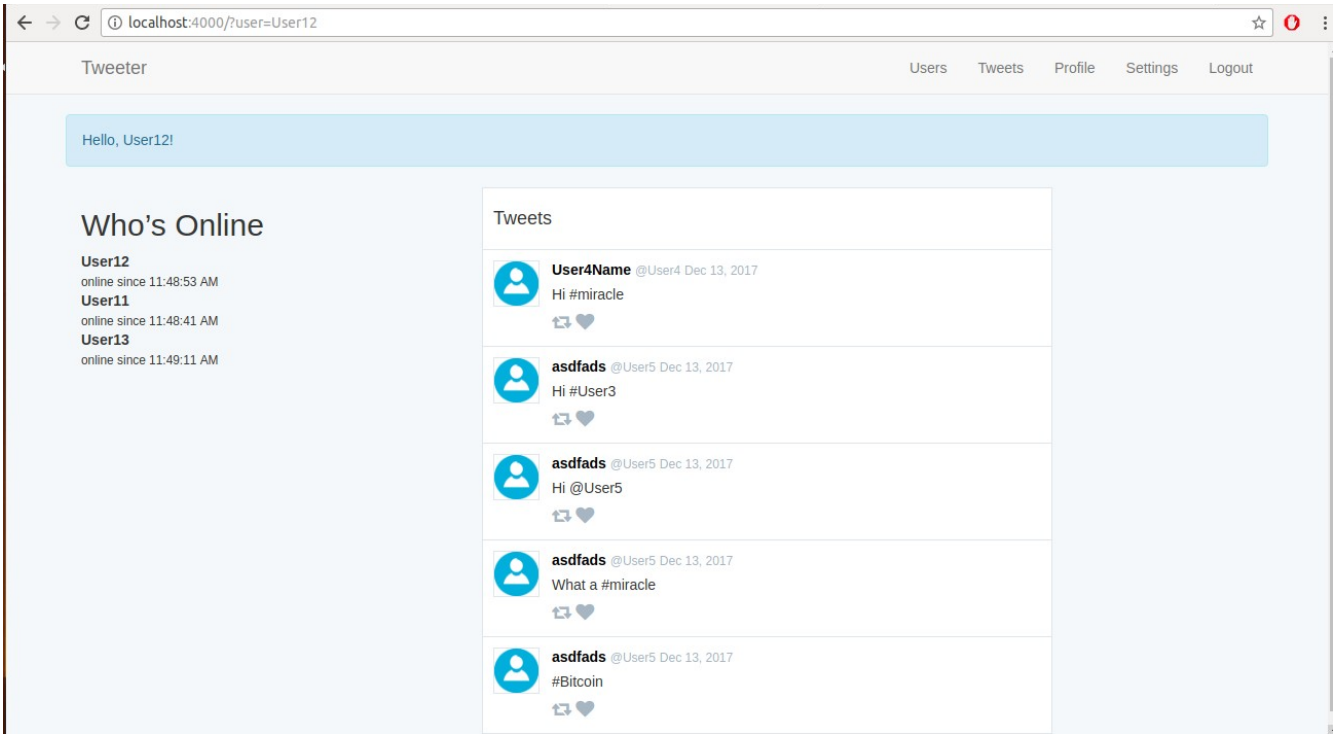
2. Login/Logout

Existing user can log in to the application by providing the login name and password provided during sign up. The typed password will be displayed in password format(not plain text). A user can logout of the application anytime by clicking the logout button.



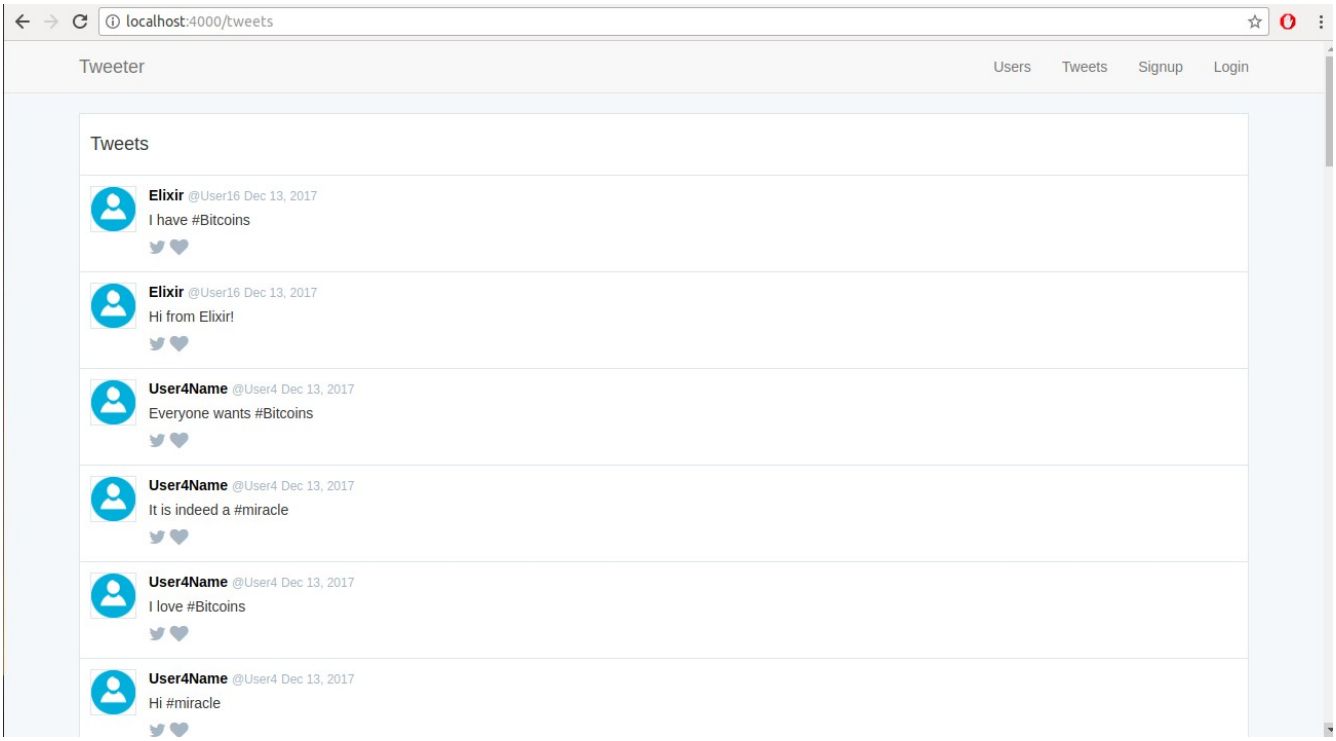
3. Home Page

The home page of Tweeter displays real-time user online information. It shows which user is online and when he/she logged in. It also shows the recent tweets from all the user. A signed in user can navigate to his profile from the home page.



4. Tweets

This page displays all the tweets from the database. Each tweet has a profile picture of the user, original name of the user, user handle(@user), date in which the tweet was posted, the tweet text, retweet option and the heart button(like). Retweeting on a tweet will post the same tweet on the user's profile.



5. Profile

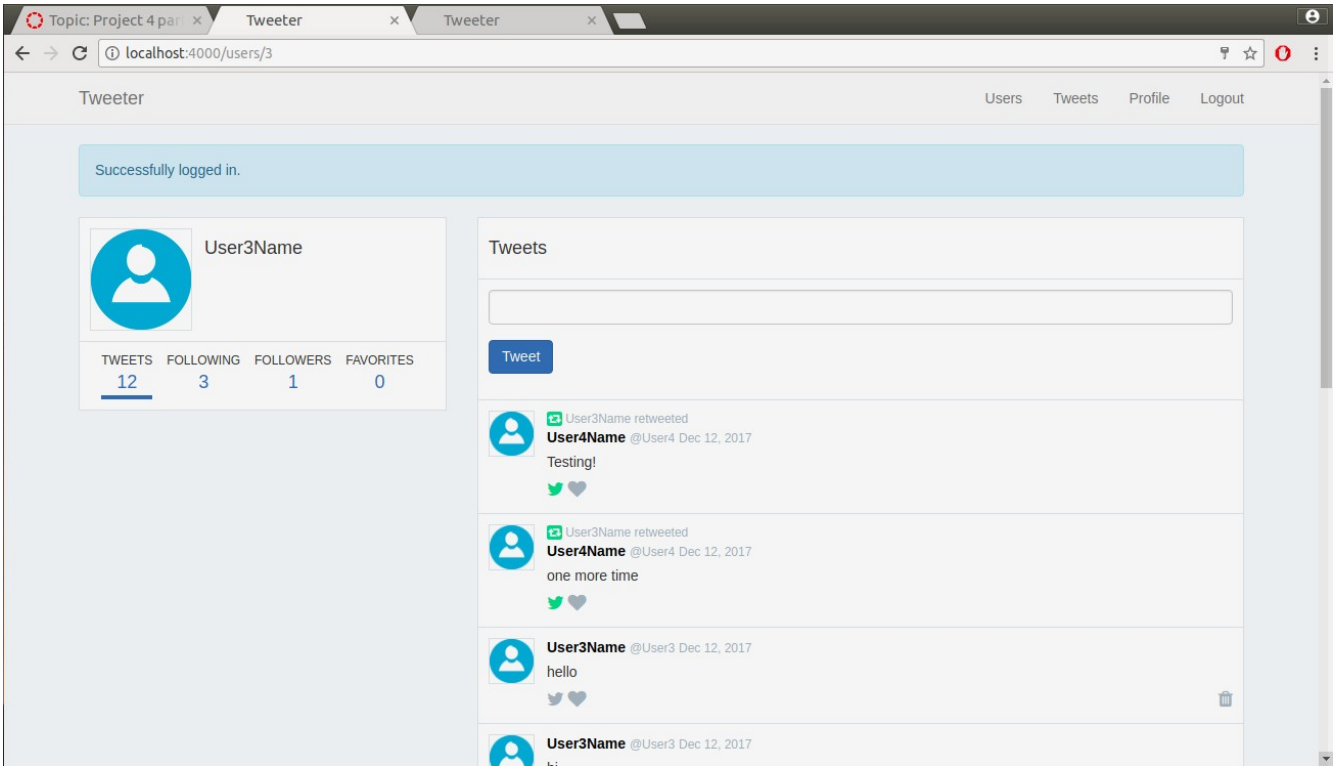
The profile describes the user information like photograph, number of tweets, following and follower information and tweets that he liked(under favorites).

The tweets section displays all the tweets and retweets tweeted by the user.

User can post tweet using #hashtag and @user mentions.

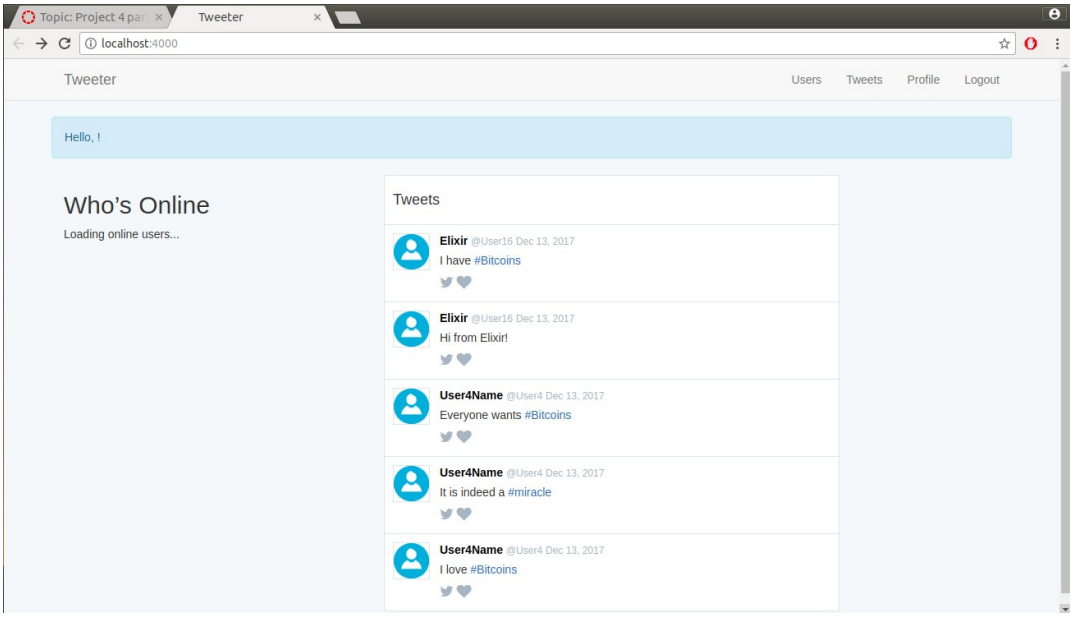
Any user can follow any other user and this information appears in the user profile.

Tweets can also be deleted by clicking on the trash icon.

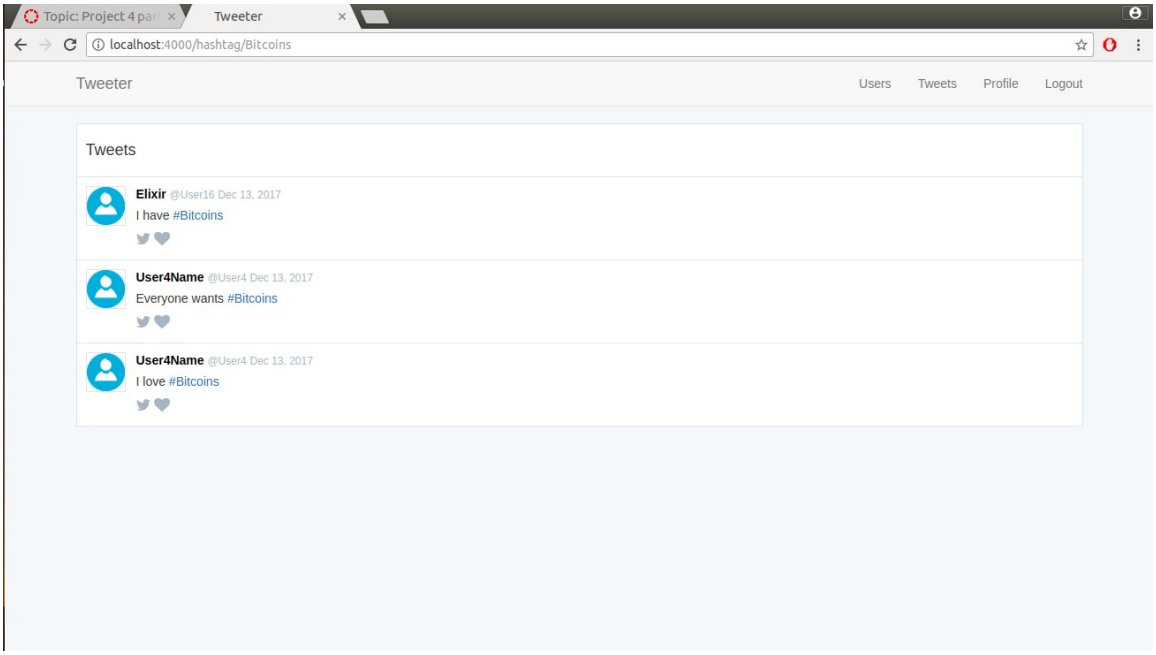


6. Hashtag search

By default all the hashtags appear as a link and clicking on that particular hashtag link will display all the tweets containing that hashtag.



By clicking on #Bitcoins:



All the activities and information provided by the user gets stored in the Ecto database.

How Tweeter works?

Introduction

Tweeter code has a number of distinct parts, each with its own purpose and role to play in building the web application:

Endpoint is responsible for handling start and end of the request lifecycle. It handles all aspects of requests and dispatches it to a designated router

Router parses incoming requests and dispatches them to the correct controller/action/channels, passing parameters as needed.

Controllers provide functions, called *actions*, to handle requests. It is also used to prepare data and pass it into views,invoke rendering via views.

Views act as a presentation layer.

Templates contain the html.eex files that contain html formatting for the pages as well as elixir commands that needs to be executed.

Channels are important part of our application is it acts as a wrapper to the websocket connection. It helps to manage sockets for easy real time communication

PubSub underlies the channel layer and allows clients to subscribe to *topics.*

Layers

1. Server

Tweeter uses Cowboy as a server which in turn uses Erlang. It captures all the activity that happens real-time in the application and renders it the console. It also records the activities of the websocket and displays when client gets connected with our server.

2. Database

Tweeter uses Ecto module which acts as a database wrapper for Elixir. With Ecto, we can read and write to different databases and model our domain data. An Ecto Repo acts as the repository and every database operation is done via this repository. Ecto Schemas are our data definitions. Ecto Queries tie both schemas and repositories together. Ecto Changesets are used to declare transformations we need to perform on our data before our application can use it.

3. Channels

Tweeter uses channels to add soft real time features to the application. Senders broadcast messages about topics. Receivers subscribe to topics so that they can get those messages. Our channel has a Javascript client that handles message passing.

A socket allows the client-server connection to remain open so the client and server can continue to exchange messages as long as the user remains on the page.

A User Socket module will be present in web/channel/user_soceket.ex. This module is used for handling socket authentication in a single place.

A user has one connection to the server (the socket) but can join and leave many “rooms” (the channel), and will only be able to send and receive tweets in channels they’ve connected to. Our RoomChannel module can handle as many different “twitter rooms” as we need. But we are dealing with a single tweeter channel available to all users.

Welcome | user_socket.ex | presence.ex

```elixir
defmodule App.UserSocket do
  use Phoenix.Socket

  ## Channels
  channel "room:Tweeter", App.RoomChannel

  ## Transports
  transport :websocket, Phoenix.Transports.WebSocket

  # Socket params are passed from the client and can
  # be used to verify and authenticate a user. After
  # verification, you can put default assigns into
  # the socket that will be set for all channels, ie
  #
  #     {:ok, assign(socket, :user_id, verified_user_id)}

  def connect(%{"user" => user}, socket) do
    {:ok, assign(socket, :user, user)}
  end

  def id(_socket), do: nil
end

```

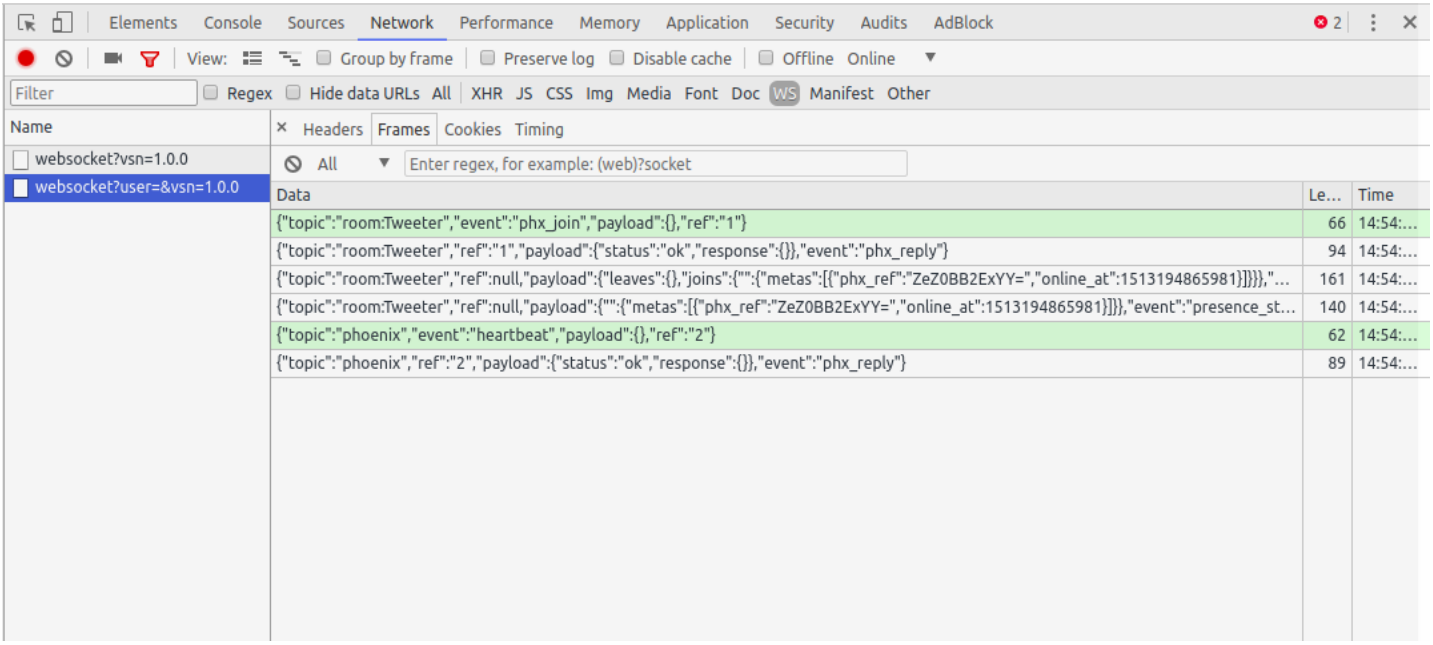
Welcome | user_socket.ex | room_channel.ex

```elixir
defmodule App.RoomChannel do
  use App.Web, :channel
  alias App.Presence

  def join("room:Tweeter", _, socket) do
    IO.puts"Reached!"
    send self(), :after_join
    {:ok, socket}
  end

  def handle_info(:after_join, socket) do
    Presence.track(socket, socket.assigns.user, %{
      online_at: :os.system_time(:milli_seconds)
    })
    IO.puts("socket is #{inspect socket}")
    push socket, "presence_state", Presence.list(socket)
    IO.puts " connected"
    {:noreply, socket}
  end

  def handle_in("message:new", message, socket) do
    IO.puts("socket is #{inspect socket}")
    broadcast! socket, "message:new", %{
      user: socket.assigns.user,
      body: message,
      timestamp: :os.system_time(:milli_seconds)
    }
    {:noreply, socket}
  end

end
```

When we call room.join(“room:Tweeter”) in our JavaScript, this module’s after_join function will call Presence.track to start tracking that user’s presence in this room. When a client signs in, there may already be other users online, so we push the current state of who else is online (Presence.list) back to the user via a “presence_state” event.

JSON Request/Response

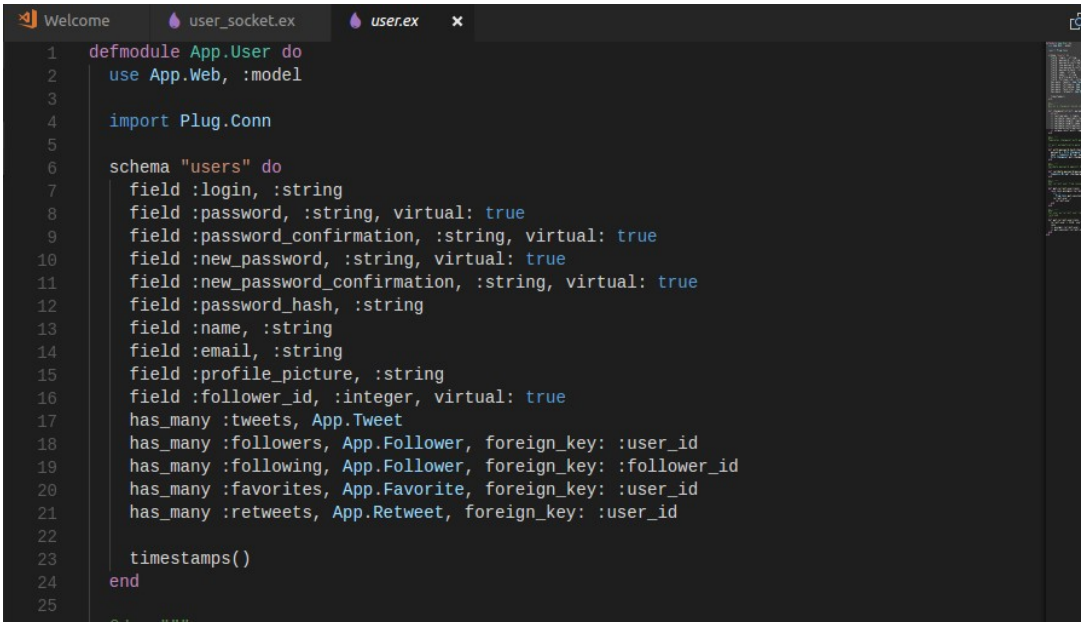
One noteworthy thing with the use of phoenix channels is that there is no need for explicit conversion of requests/response to and from json as Phoenix internally handles this conversion. This is evident from inspection of web traffic:



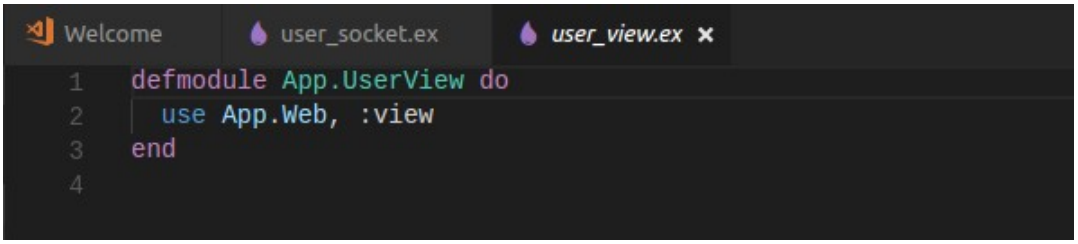
5. MVC Pattern for Client Implementation

All the pages in our application follow Model-View-Controller pattern. For every single functionality, there is a separate model, view and controller files developed in elixir. The model provides the schema and translation, view provides the presentation interface and controller handles the information exchange. All the mvc elixir files have clear naming conventions and it would be easy to understand the structure of the submitted code.

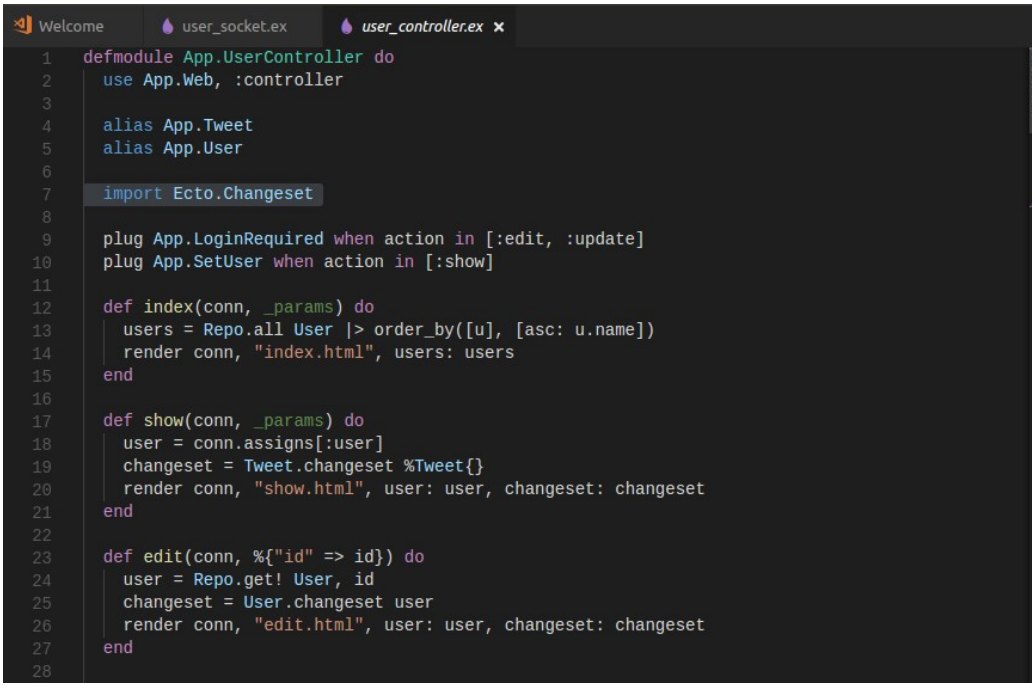
Model



View



Controller



Performance of Tweeter with UI

We have built the application using phoenix framework with an UI -based client. Every information from UI like user signup, login, tweeting, etc are being captured in the database. Once the socket connection gets created, we were able to see the real-time transactions – display of online users, time at which they logged in, tweets, etc. We have tested the UI with 100 clients and there was no lag in the performance. Since UI approach was taken for this project(as per the recent announcement from TA), we were not able to simulate thousands of clients at a time like in part 1. Limitation of the browser made us to capture the results offline.

We anticipate that there will be 10% performance degradation in Tweeter as compared to the pure elixir message passing approach as there will some time lost in storing the data to the database and broadcasting the data to connected clients via websockets.