# Elixir Phoenix Workshop

# What we will build

A very minimalistic version of a Chat App, with Phoenix Features will include :

- User Management
- Form Authentication
- JWT token for session management
- Minimal (very very) Chat interface
- Room Management

| NOTE | Let's dig in |
|---|---|

# Create a Phoenix App - tag : creation

We will build an App called Slang, Sla for Slack, and ng because why not!?!
The name of the app will be used in Module name, so if you want to be able to copy paste easily use `slang` as App name

```
mix phx.new folder_name --app slang
```

It will generate a Phoenix Project in the folder specified You will get asked to download dependencies, you can safely accept :D

## A tour of the Generated Code

```
|
|_ _build # where the package will be build
|_ assets  # static assets, Javascript, CSS, images, webpack
|_ config  # conf for the app
|_ deps    # dependencies binaries
|_ lib     # it's where our code will be.
|_ priv    # database migration and target for static build
|_ test    # TESTS for the sake of your mind!
mix.exs    # project description, dependencies, build config
README.md  # teh README, every project need one
```

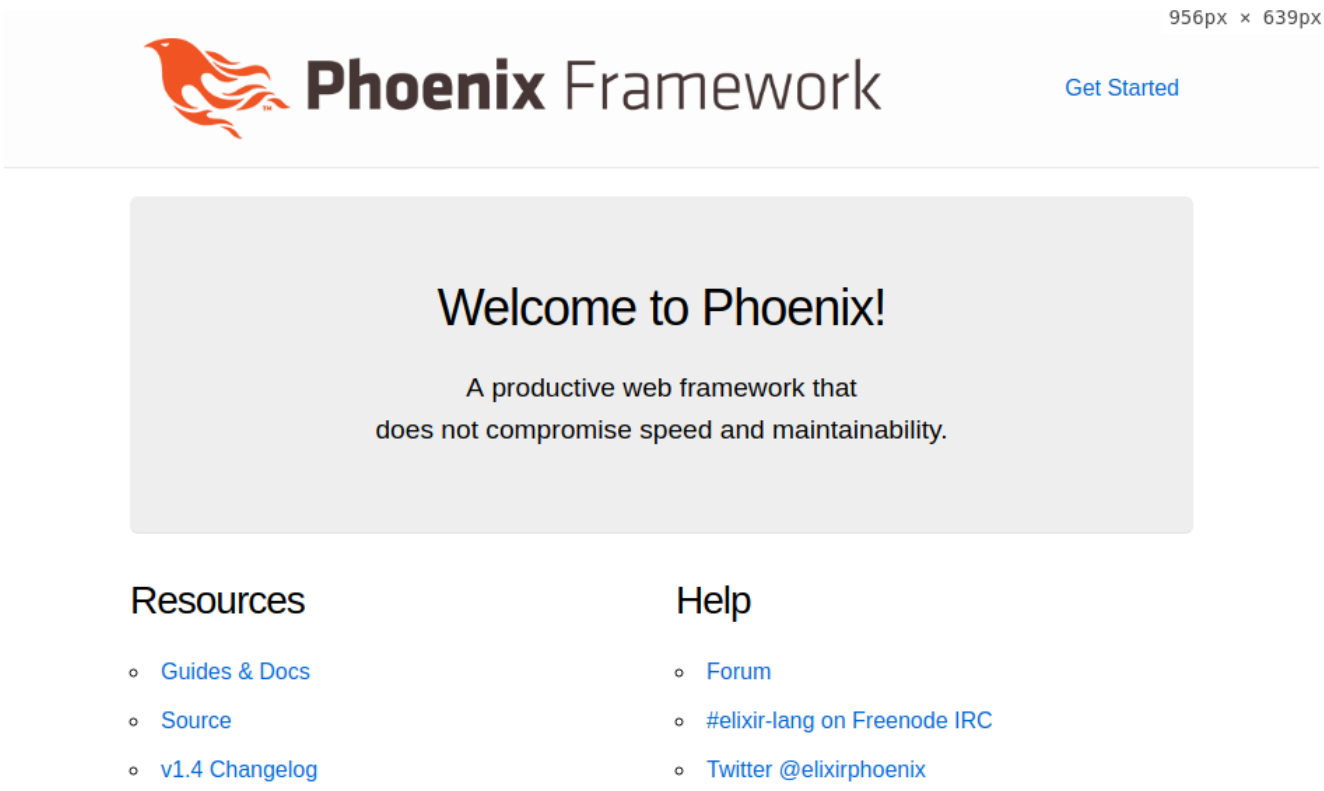
Not too much spoiler, I will get in lengthy details :) Ask me for more during the workshop!

## Run the project

First Create your database, and run the migrations

```
mix ecto.create
mix ecto.migrate
```

now let's open a browser and navigate to http://localhost:4000/




if you see something like that, you are on the right path fellow Alchemists :)

# Authentication debut, User Creation

We will use another code generator. To generate the different form for our users management

Let's go :

```
mix phx.gen.html Accounts User users email:string password:string
password_confirmation:string
```

Generated a bunch of stuff… first let's follow the instructions.

place this line in lib/slang_web/router.ex

*lib/slang_web/router.ex*

```
  scope "/", SlangWeb do
    pipe_through :browser

    get "/", PageController, :index

    resources "/users", UserController
  end
```

and the run the migration

```
mix ecto.migrate
```

now let's have a look to the apps !

```
mix phx.server
```

Where to look for nothing changed?

```
mix phx.routes
```

You will get a list of routes. Let's go to http://localhost:4000/users as a starting point

Oh nice an empty list. Not surprising we might need to create some user…
Continue to http://localhost:4000/users/new (you have a link on the previous page)
keep playing with the different screen for a moment

We have something that nearly looks like a user sign up process :D

Ok awesome on to the next level, stop saving clear password !!! We are not Sony !!!

# Securing user sign up

## Virtual Attributes in Schema

let's replace those 2 password attribute in the schema to make them virtual in lib/slang/accounts/user.ex

*lib/slang/accounts/user.ex*

```
    field :password, :string, virtual: true
    field :password_confirmation, :string, virtual: true
```

and add a new field to store the password_hash

*lib/slang/accounts/user.ex*

```
    field :password_hash, :string
```

Ok we just changed the database table underneath. If you look at your postgres table there's those 2 fields, `password` and `password_confirmation`... not quite what we will store now...
We are going to store `password_hash`
Let's create a migration to modify our level

ok smarter but not quite nearly done We need to hash the password, and for that we need a dependency

```
mix ecto.gen.migration change_user_table
```

it generate a file in the `priv/repo/migrations`

let's open it
damn it's empty we need to find a way to do our changes...
take a look on this page : https://hexdocs.pm/ecto_sql/Ecto.Migration.html
and look for something like change or alter or something

once you know what to do run your migration

```
mix ecto.migrate
```

and check your table in postgres pgadmin3 or 4 or anything you want and see that it has changed :)

ok if you can't make it on the migration, here it is :

*priv/repo/migrations/_timestamp_change_user_table.exs*

```
defmodule Slang.Repo.Migrations.Change_user_table do
  use Ecto.Migration

  def change do
    alter table("users") do
      add :password_hash, :text
      remove :password
      remove :password_confirmation
    end
  end
end
```

# Hashing the password

let's open mix.exs and add a dependency to hash password

*mix.exs*

```
  defp deps do
    [
    ...
        {:comeonin, "~> 4.0"},
        {:bcrypt_elixir, "~> 0.12"},
    ...
    ]
```

now get those new dependencies and restart our dev environment

```
mix deps.get
mix phx.server
```

And we are back up :D

## Windows concerns

you need some sort of compiler instaled for bcrypt to work. On Windows if you don't already have one installed, you can follow frenshkosh way of fixing the problem

you can work with Chocolatey to get the adapted tools.

**Chocolatey** check this link https://chocolatey.org/docs/installation folllow steps for Installing Chocolatey chapter

**Visual C++ Build Tools**

```
choco install VisualCppBuildTools
```

(and follow steps).

add C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin in your path

Thanks frenshkosh for the solution :D

Let's hash this password !

We will do that in the changeset of the schema.
Changesets allow filtering, casting, validation and definition of constraints when manipulating structs
When we want to create a record in the database we just pass a Map describing the schema through a changeset pipeline and forward that to the Repo The Repo is the "instance" that handles the order we pass to the database.

Ok back to our little complicated changeset. replace the current changeset in the User Schema `lib/slang/accounts/user.ex` by this

*lib/slang/accounts/user.ex*

```
defmodule

  ...

  alias Slang.Accounts.User

  ...

  @doc false
  def changeset(%User{} = user, attrs) do
    user
    |> cast(attrs, [:email, :password, :password_confirmation])
    |> validate_required([:email, :password, :password_confirmation])
    |> validate_format(:email, ~r/@/) # Check that email is valid
    |> validate_length(:password, min: 8) # Check that password length is >= 8
    |> validate_confirmation(:password) # Check that password ===
password_confirmation
    |> unique_constraint(:email)
    |> put_password_hash # Add put_password_hash to changeset pipeline
  end

  defp put_password_hash(changeset) do
    case changeset do
      %Ecto.Changeset{valid?: true, changes: %{password: pass}}
        ->
          put_change(changeset, :password_hash, Comeonin.Bcrypt.hashpwsalt(pass))
      _ ->
          changeset
    end
  end
```

What does it do ?

the alias is a macro that enable the compiler to use the User Schema inside the changeset itself. It's like an import in other language

- cast : will compare types of the map attributes(attrs) and the types of the User schema
- validate_required : check that everything has been set in the map
- validate_format : validate that the email as a @ somewhere
- validate_length : makes it more complicated to set a password….
- validate_confirmation : will compare the `password` and `password_confirmation`, it's a build in validator… Neat huh?
- unique_constraint : verify that email isn't already registered
- put_password_hash : it's our user defined function bellow

put_password_hash verify that the changeset is valid with an awesome pattern match that verify things

andcat same time assign password to pass
then use Comeonin (our new dependency) to hash the password and return a new changeset with the password_hash attributes set correctly with the hash

tag after those changes is `hashing_password`

pfffewwww… not a bad thing done

Now we can create a user and store it's hashed password and life is good and create, except we have no screen to login !

# Login Screen

We will play with phoenix forms and understand how router forms controllers and views interact ! How supremely excited it is ! (is that too much?)

Let's take a look at this http://localhost:4000/users/new

I'd say that it look close to a login screen… We'll use that as a base for our new login screen !

But first let's create a url for this login page, in `lib/slang_web/router.ex`

Before we change something let me give you some kind of explanations
How is this router working :

First thing we notice is that pipeline,

*lib/slang_web/router.ex*

```
  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end
```

What's a pipeline? As it's name let think it's a pipeline through which every request entering phoenix will transit. Each step of the pipeline is an elixir plug. A plug is A specification for composable modules between web applications, this is the building block of nearly every web application in Elixir.

so here we will go through:

- verifiy that the request speaks html
- attach the session to the connection (req/resp in phoenix)
  the session is stored in a JWT token as a cookie comming from the browser. yes we are stateless by default :D
- attach the flash message - specific part of the session to store message for the user,
  it's a build in mechanism in phoenix - we want to display messages to our user when an action is done during a navigation (saving a form for example)

- protect_from_forgery is pretty explicit - it protects from forgery :)
- secure the browser headers - another security plug to protect the users and the app.

Next in the router is another pipeline used for api - we will see that later in the workshop

Then comes this Scope :

*lib/slang_web/router.ex*

```
  scope "/", SlangWeb do
    pipe_through :browser

    get "/", PageController, :index

    resources "/users", UserController

  end
```

Ok the scope is where the actual routing is done.
As you can see this is where we use our browser pipeline.
So for a connection to be handled by this scope it will have to comply with the pipeline :browser

*lib/slang_web/router.ex*

```
    get "/", PageController, :index
```

Defines a endpoint at the root of the webapp
If the request is a GET it will be handled by the `index` function PageController refered to as the atom :index

*lib/slang_web/router.ex*

```
    resources "/users", UserController
```

resources define a complete set of urls for CRUD operation on entity it's the equivalent of all this:

*mix phx.routes*

```
user_path  GET     /users           SlangWeb.UserController :index
user_path  GET     /users/:id/edit  SlangWeb.UserController :edit
user_path  GET     /users/new       SlangWeb.UserController :new
user_path  GET     /users/:id       SlangWeb.UserController :show
user_path  POST    /users           SlangWeb.UserController :create
user_path  PATCH   /users/:id       SlangWeb.UserController :update
           PUT     /users/:id       SlangWeb.UserController :update
user_path  DELETE  /users/:id       SlangWeb.UserController :delete
```

Here is what we are looking for, this `new` route let's try to create a new route to our login page. I want my login page to be at the root of my webapp : http://localhost:4000/login

*lib/slang_web/router.ex*

```
    get "/login", UserController, :login
```

let's had that at the end of our scope you can try the URL in your browser right now, but it will fail since the UserController can't handle our login operation yet. Let's fix that !

Open `lib/slang_web/controller/user_controller.ex`

We have a lot of function in there… each one is handling an operation from the router. This is where the logic on the connection is handled. Please don't put too much application logic here. It's not the best place, your logic should go in the `lib/slang` folder in it's respectable domain :)

Now we really need this login action to be implemented, since we want to copy the new page, let's also copy the new function from our UserController and rename it login

*lib/slang_web/controller/user_controller.ex*

```
  def login(conn, _params) do
    changeset = Accounts.change_user(%User{})
    render(conn, "new.html", changeset: changeset)
  end
```

if we point to the `new.html` template, phoenix is happily showing you the new form, but we want a login form… so :

*lib/slang_web/controller/user_controller.ex*

```
  def login(conn, _params) do
    changeset = Accounts.change_user(%User{})
    render(conn, "login.html", changeset: changeset)
  end
```

Now Phoenix is not happy anymore, he commands us to create a login template… let's oblige :)

just copy paste this file `lib/slang_web/templates/user/new.html.eex` to `lib/slang_web/templates/user/login.html.eex`

It renders something ! Awesome ! Well done fellow Alchemists !

Now let's customize our new login page ! Take a look at the `lib/slang_web/templates/user/login.html.eex` Oy Oy Oy it refers to some kind of form.html. That's a composition pattern used in the templates. When Phoenix generate the template, it generate only one form and reuse for all the actions : creation and update

Since we are not aiming for reusability just copy the `lib/slang_web/templates/user/form.html.eex` to `lib/slang_web/templates/user/form_login.html.eex` and update the login template to reflect the change

*lib/slang_web/templates/user/login.html.eex*

```
<%= render "form_login.html", Map.put(assigns, :action, Routes.user_path(@conn,
:create)) %>
```

and the `lib/slang_web/templates/user/form_login.html.eex`

*lib/slang_web/templates/user/form_login.html.eex*

```
<%= form_for @changeset, @action, fn f -> %>
  <%= if @changeset.action do %>
    <div class="alert alert-danger">
      <p>Oops, something went wrong! Please check the errors below.</p>
    </div>
  <% end %>

  <%= label f, :email %>
  <%= text_input f, :email %>
  <%= error_tag f, :email %>

  <%= label f, :password %>
  <%= password_input f, :password %>
  <%= error_tag f, :password %>

  <div>
    <%= submit "Login" %>
  </div>
<% end %>
```

Last but not least for this part,
the user will not find the login URL by itself, we need to create a link for that on the main page !

*lib/slang_web/templates/layout/app.html.eex*

```
        <nav role="navigation">
          <ul>
            <li><a href="https://hexdocs.pm/phoenix/overview.html">Get
Started</a></li>
            <li>
              <%= link "New Login", to: Routes.user_path(@conn, :login) %>
            </li>
          </ul>
        </nav>
```

so we define a link with the phoenix helper function link, and we link to a `Routes.login_path` But where is it comming from? How do I know ?

*mix phx.routes*

```
-> % mix phx.routes
page_path  GET     /               SlangWeb.PageController :index
user_path  GET     /users          SlangWeb.UserController :index
user_path  GET     /users/:id/edit  SlangWeb.UserController :edit
user_path  GET     /users/new      SlangWeb.UserController :new
user_path  GET     /users/:id      SlangWeb.UserController :show
user_path  POST    /users          SlangWeb.UserController :create
user_path  PATCH   /users/:id      SlangWeb.UserController :update
           PUT     /users/:id      SlangWeb.UserController :update
user_path  DELETE  /users/:id      SlangWeb.UserController :delete
user_path  GET     /login          SlangWeb.UserController :login
```

See that last line in bold? it's where you can find the function we are using in the template.

The `@conn` parameter is the way we refer to arguments passed to the template, so yeah the `conn` is passed as an argument to the template, that should not be a great surprise, since the connection hold the state of the request.

We can see how the param are passed to the template in the controller

*lib/slang_web/controllers/user_controller.ex*

```
  def index(conn, _params) do
    users = Accounts.list_users()
    render(conn, "index.html", users: users)
  end
```

we will pass the list of users in a Map with the key `users`

it is used in the template like this :

*lib/slang_web/templates/user/index.html.eex*

```
<%= for user <- @users do %>
    <tr>
      <td><%= user.email %></td>
      <td><%= user.password %></td>
      <td><%= user.password_confirmation %></td>

      <td>
        <%= link "Show", to: Routes.user_path(@conn, :show, user) %>
        <%= link "Edit", to: Routes.user_path(@conn, :edit, user) %>
        <%= link "Delete", to: Routes.user_path(@conn, :delete, user), method:
:delete, data: [confirm: "Are you sure?"] %>
      </td>
    </tr>
<% end %>
```

we see that it iterates in the list of users using the `@users` syntax. you can also see that the `user` is defined inside the template and doesn't need the `@` that the way to differenciate between template arguments and scoped variable of the template. you can also see how to use the `for` loop in Elixir with the backward arrow `←`

For cosmetic reason I want that when I click on the Elixir Phoenix Logo I am redirected to the index of the app, instead of the Phoenix web page.

```
      <section class="container">

        ...

        <%= link to: Routes.page_path(@conn, :index), class: "phx-logo" do %>
          <img src="<%= Routes.static_path(@conn, "/images/phoenix.png") %>"
alt="Phoenix Framework Logo"/>
        <% end %>
      </section>
```

We replace the `<a href` that surround the `img` with the Phoenix Template way to create a link.
Same as before, we used the `Routes` helper function to redirect to the `:index` of the app.

That the end of this part concentrated on the templating system of Phoenix

The tag of the code in the repo is : `forms_template_login`

# Managing JWT Token with Guardian

That's the worst part (in terms of complexity) - you can avoid it by moving to the tag `user_authentication_finished`

Now that we have an awesome page where we can have a user input there credential, It might be usefull to process those credential to authenticate the user !

Let's do that right now !

first we defined a `form_login` but what does it do?

*lib/slang_web/templates/user/form_login.eex*

```
<%= form_for @changeset, @action, fn f -> %>
  <%= if @changeset.action do %>
    <div class="alert alert-danger">
      <p>Oops, something went wrong! Please check the errors below.</p>
    </div>
  <% end %>

  <%= label f, :email %>
  <%= text_input f, :email %>
  <%= error_tag f, :email %>

  <%= label f, :password %>
  <%= password_input f, :password %>
  <%= error_tag f, :password %>

  <div>
    <%= submit "Login" %>
  </div>
<% end %>
```

if looks like that a function, `form_for`
that takes 3 params, a changeset (a keyvalue map), an action (!!) and a function.

so `@action` is what will be called on submit and the function as a parameter is simply a way to like the html tag with the elixir code. This function receive a parametre f, that is passed to each component, label, text_input, error_tag, etc…

But the action seems to come from a controller. Let's look at the `user_controller`

*lib/slang_web/controllers/user_controller.ex*

```
  def login(conn, _params) do
    changeset = Accounts.change_user(%User{})
    render(conn, "login.html", changeset: changeset)
  end
```

Hum nothing here… except we are rendering a login.html, let's look at this template

*/lib/slang_web/templates/user/login.html.eex*

```
<h1>Login</h1>

<%= render "form_login.html", Map.put(assigns, :action, Routes.user_path(@conn,
:create)) %>

<span><%= link "Back", to: Routes.user_path(@conn, :index) %></span>
```

Yeah here ! That's where we define the action when we add a new keyvalue to the assigns map. `assigns` is

the named of the map used to hold the variable of the template

So the action is a route pointing to the `:create` action in `UserController` Looking at the `mix phx.routes` it's a POST which make sense since we want to push a form

But we don't want to create a user… we want to validate its email / password

The Plan : - Add a new route to handle the post submition - Create a function in the controller to handle to POST - Delegate the check to the Accounts Module - Find the user by it's email from the database - Compare the hashed password from the database, with the form hashed password - Give all that back to the controller for it to display result to the user.

Let's create a new Route in the router for that.

*lib/slang_web/router.ex*

```
  scope "/", SlangWeb do

    ...

    post "/validate_login", UserController, :validate_login
  end
```

nothing new, a post to a controller function

in the controller let's create that function

*lib/slang_web/controllers/user_controller.ex*

```
  def validate_login(conn, %{"user" => login_params}) do
    case Accounts.form_sign_in(login_params["email"], login_params["password"], conn)
do
      {:ok, conn} ->
        conn
        |> put_flash(:info, "Login successfully.")
        |> redirect(to: Routes.page_path(conn, :index))
      {:error, :unauthorized} ->
        changeset = Accounts.change_user(%User{})
        conn
        |> put_flash(:error, "Couldn't log you in")
        |> render("login.html", changeset: changeset)
    end
  end
```

it delegate the check of the password to the Accounts Module if form_sign_in return the type `{:ok, conn}` everything is ok
and we can redirect the user to the home page, and display a message about the success if form_sign_in returns an error, user stays on login page, and an error is displayed

first we need a function the get a user from the database by it's email adress .lib/slang/accounts/accounts.ex

```
defmodule Slang.Accounts do

  import Comeonin.Bcrypt, only: [checkpw: 2, dummy_checkpw: 0]
  ...
```

first import the cryptographic function that we will use `checkpw` and `dummy_checkpw`

*lib/slang/accounts/accounts.ex*

```
  ...
  defp get_by_email(email) when is_binary(email) do
    case Repo.get_by(User, email: email) do
      nil ->
        dummy_checkpw()
        {:error, "Login error."}
      user ->
        {:ok, user}
    end
  end
```

Repo is the data repository, it handle calls to the database. It has a convinient function that will retrieve one entity called `get_by`, takes 2 args, first is the entity, and second is a field and it's value to query

`dummy_checkpw` will create a delay when the user is not found, mimiking a password hash comparaison, to avoid email fishing on your app.

if a user is found it return it. if no user is found it return an error.

ok now we need to create this form_sign_in function in the Accounts module

*lib/slang/accounts/accounts.ex*

```
  ...

  def form_sign_in(email, password, conn) do
    case email_password_auth(email, password) do
      {:ok, _user} ->
        {:ok, conn}
      _ ->
        {:error, :unauthorized}
    end
  end

  ...
```

it delegate the password checkin to the email_password_auth function

*lib/slang/accounts/accounts.ex*

```
  ...

  defp email_password_auth(email, password) when is_binary(email) and
is_binary(password) do
    with {:ok, user} <- get_user_by_email(email),
    do: verify_password(password, user)
  end

  ...
```

call the get_user_by_email function to retrieve the entity from the database if it does, it calls another function to verify the password

*lib/slang/accounts/accounts.ex*

```
  ...

  defp verify_password(password, %User{} = user) when is_binary(password) do
    if checkpw(password, user.password_hash) do
      {:ok, user}
    else
      {:error, :invalid_password}
     end
  end

  ...
```

`checkpw` is a function provided by Comeonin (our crypto dependency) to check the password against it's hashed version.

if the user password is validated it is returned else it return as error with an invalid_password

And we are done ! We can check the password of the user and be sure it's validated

> **NOTE** git tag at this point is : `password_check_done`

Only thing we need to do now is store that in the JWT token of the app. That's the job of Guardian

first add the guardian dependency:

*mix.exs*

```
  defp deps do
    [
    ...
      {:guardian, "~> 1.2.1"},
    ...
    ]
```

then configure it:

*config/config.exs*

```
# Guardian config
config :slang, Slang.Guardian,
       issuer: "slang",
       secret_key: "a2rLR0uj4W1nQ3h3kTCONdx/jGtKlnrHvu0AUv7EwLLthDFTvsjXNL3SRZbPd7x/"
```

a word on config,
there's 2 part to the config :

- config.exs witch is the default / main config,
- dev.exs / prod.exs which is a loaded by the config.exs based on the MIX_ENV environment variable

in order to use guardian we need to give it some implementation

*lib/guardian.ex*

```
defmodule Slang.Guardian do
  use Guardian, otp_app: :slang

  def subject_for_token(user, claims) do
    sub = to_string(user.id)
    {:ok, sub}
  end

  def subject_for_token(, _) do
    {:error, :reason_for_error}
  end

  def resource_from_claims(claims) do
    id = claims["sub"]
    resource = Slang.Accounts.get_user!(id)
    {:ok,  resource}
  end

  def resource_from_claims(_claims) do
    {:error, :reason_for_error}
  end
end
```

Now Guardian is configured and ready to be used.

Let's call a login function in our `form_sign_in` on success of password check like that :

*lib/slang/accounts/accounts.ex*

```
  ...
  def form_sign_in(email, password, conn) do
    case email_password_auth(email, password) do
      {:ok, user} ->
        {:ok, login(conn, user)}
      _ ->
        {:error, :unauthorized}
    end
  end
  ...
```

*lib/slang/accounts/accounts.ex*

```
  def login(conn, user) do
    conn
      |> Guardian.Plug.sign_in(user)
  end
```

the `Guardin.Plug.sign_in(user)` returns a connection context that contains the token with our user

Principale in it. Now the JWT token containing the session will also hold the user definition in the encoded payload.

We now need to unwrap this data and make it available in the session of the app.

We will do that by add a new pipeline to the scope. Creating your own pipeline to get data from the JWT Token into our session

*lib/slang_web/current_user.ex*

```
defmodule SlangWeb.CurrentUser do
  import Plug.Conn

  def init(opts), do: opts

  def call(conn, _opts) do
    current_user = Guardian.Plug.current_resource(conn)

    # Assigns the current user if it exists, and whether it is an admin or not
    assign(conn, :current_user, current_user)
  end

end
```

Creates a Plug, that only job is to look into the connection and extract the current_user using Guardian helper function `current_resource`

Now on to the pipeline

*lib/slang_web/auth_browser_pipeline.ex*

```
defmodule SlangWeb.Guardian.AuthBrowserPipeline do
  use Guardian.Plug.Pipeline, otp_app: :Slang,
  module: Slang.Guardian,
  error_handler: Slang.AuthErrorHandler

  plug Guardian.Plug.VerifySession
  plug Guardian.Plug.LoadResource, allow_blank: true
  plug SlangWeb.CurrentUser
end
```

We use others Guardian helper function to : - Verifiy the Session in the request - Load the resources from header and store them in the Elixir Connection - and decode the CurrentUser into the connection also.

Using our new pipeline in the router

*lib/slang_web/router.ex*

```
  pipeline :with_session do
        plug SlangWeb.Guardian.AuthBrowserPipeline
    end
```

add it to the scope

```
  scope "/", SlangWeb do
    pipe_through [:browser, :with_session]
```

Our logged in user can now be retrieved from the connection

all we need is a way to see it.

Let's improve our main page and handle the login / logout If user is Logged In, display it's email, and a logout link. But if user is not logged In, display a login Link.

Once again we will modify the header of our main page :

*lib/slang_web/templates/layout/app.html.eex*

```
        <nav role="navigation">
          <ul>
<li><a href="https://hexdocs.pm/phoenix/overview.html">Get Started</a></li>
            <li>
              <%= if assigns[:current_user] do %>
                <%= link "Logout", to: Routes.user_path(@conn, :logout), method:
"post" %>
                <%= assigns.current_user.email %>
                <%= assigns.current_user.id %>
              <% else %>
                <%= link "Login", to: Routes.user_path(@conn, :login) %>
                not connected
              <% end %>
            </li>
          </ul>
        </nav>
```

All we had previously is the link for Login (in italic) Now we have the logic based on the presence or not of the `:current_user` in the `assigns` map (that's the way to check if a key is present in a map `my_map[:key_to_check]`)

Oh look a logout route ! Since we have not defined it previously Phoenix will complain (vehemently with that) about that route note present in the router !

let's add a POST route for the logout

*lib/slang_web/router.ex*

```
scope "/", SlangWeb do
  ...
      post "/logout", UserController, :logout
  ...
end
```

Add the Action in the controller :

*lib/slang_web/controllers/user_controller.ex*

```
def logout(conn, _params) do
  conn
  |> Accounts.logout
  |> redirect(to: Routes.page_path(conn, :index))
end
```

Once again the controller will just orchestrate and delegate the action to our domain-like accounts

*lib/slang/accounts/accounts.ex*

```
def logout(conn) do
  conn
  |> Guardian.Plug.sign_out
end
```

And we are all set !

User is authenticated, saved in the session for easy use, and it's displayed for the user convenience.

All this noodling is done, the worst part is over, now it will be more fun !

Once again, if you've been bored about all those copy paste to do in your app code, just checkout the branch `user_authentication_finished` copy and paste is somewhere else and attack the next chapter that will be a lot more fun !

# The Chat !

## Models

We want a chat, with Rooms to chat inside ! So we want **messages** from **users** inside **rooms** Sounds doable !

Ok generating those **message** and **room** entity

First the Room in a Chat context

```
mix phx.gen.html Chat Room rooms name:string description:string
```

And then the Message in the same context but with a relation to the Room and the User Each message is from a user and is going in a room :D

```
mix phx.gen.html Chat Message messages text:string sender_id:references:users
room_id:references:rooms
```

follow the instructions on screen and add the 2 resources in the router

*lib/slang_web/router.ex*

```
  scope "/", SlangWeb do
    ...
    resources "/rooms", RoomController
    resources "/messages", MessageController
    end
```

and apply the migration to the database

```
mix ecto.migrate
```

You can look at the app and the new shema added

```
mix phx.server
```

http://localhost:4000/rooms/ http://localhost:4000/messages/

What would be nice is that the app when it start add some default room
I want a Lobby for everyone and a Troll room for them to Rage :D

We can do that in Phoenix in a seed file.

Better be carefull if you don’t want to have doubles in the database.

*priv/repo/seeds.exs*

```
alias Slang.Chat
alias Slang.Chat.Room
alias Slang.Accounts
alias Slang.Accounts.User
alias Slang.Repo

# USERS
case Repo.get_by(User, email: "nicolas.savois@gmail.com") do
  nil ->
    {:ok, _changeset } = Accounts.create_user %{
      email: "nicolas.savois@gmail.com",
      password: "niconico",
      password_confirmation: "niconico"
    }
  %User{} = user ->
    IO.inspect("User #{user.email} already exists, nothing to do")
end

# Default Rooms
rooms = [
  %{ name: "Lobby", desc: "default room"},
  %{ name: "Troll Den", desc: "Where troll shout and each others"}
]

Enum.map(rooms, fn %{name: room_name, desc: room_desc} ->
  case Repo.get_by(Room, name: room_name) do
    nil ->
      {:ok, _changeset} = Chat.create_room %{
        name: room_name,
        description: room_desc
      }
    %Room{} = room ->
      IO.inspect("Room #{room.name} already exists, nothing to do")
  end
end
)
```

We check if the User with email nicolas.savois@gmail already exists. If not we create it we are using the create_user that hash our password. Don't do that in production... the password are in plain text...

For rooms we use a list of map to describe our rooms, and map through this list to check if it already exists and create it if not.

Planting the seeds :

```
mix run priv/repo/seeds.exs
```

We can check our new records in the screens
http://localhost:4000/rooms/
and
http://localhost:4000/users/

Seems we can now add a new concept to the Mix, Phoenix channel !

the Channels enables real time communication between the elixir phoenix server and the client
It's based on websocket and litteraly handle MILLIONS of simultaneous connection on the same server
They tested it ! Here

Ok… 40 Core, and 128Go Memory… but still 2 Million connections !!!! on the same machine ! and that was 4 years ago… I wonder what it can handle now !

Ok enough rambling… let's use them !

Generate the template for the channel !

```
mix phx.gen.channel Room
```

don't forget to had it to your socket

*lib/channels/user_socket.ex*

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

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

let's take a look at the generated channel:

*lib/channels/room_channel.ex*

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

  def join("room:lobby", payload, socket) do
    if authorized?(payload) do
      {:ok, socket}
    else
      {:error, %{reason: "unauthorized"}}
    end
  end

  # Channels can be used in a request/response fashion
  # by sending replies to requests from the client
  def handle_in("ping", payload, socket) do
    {:reply, {:ok, payload}, socket}
  end

  # It is also common to receive messages from the client and
  # broadcast to everyone in the current topic (room:lobby).
  def handle_in("shout", payload, socket) do
    broadcast socket, "shout", payload
    {:noreply, socket}
  end

  # Add authorization logic here as required.
  defp authorized?(_payload) do
    true
  end
end
```

2 methods : - join : let's you controller who joins what ! - handle_in/3 : handle incomming message

What we will do it use a handle_in for new message and rebroacast that message accross the channel.

Let's try that !

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

The server is just moving things around get messages and rebroacasting it.

Need an interface to test that. In the main page of our app let's clean and play with that

First enable the channel connection from the webpage It's done with some kind of Javascript - I see your

joyfull look… don’t hide it !

*assets/js/app.js*

```
...
// Import local files
//
// Local files can be imported directly using relative paths, for example:
import socket from "./socket"
```

uncomment that last line

Now edit the socket.js to implement a minimal logic to test the channel. We will only push a message after the browser is connected and listen for message, we display everything in the console to validate the round trip

*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.on("new_msg", payload => {
  console.log(payload)
})

channel.join()
  .receive("ok", resp => {
    console.log("Joined successfully", resp)
    channel.push("new_msg", {msg: "test message", user_id:window.userId})
  })
  .receive("error", resp => { console.log("Unable to join", resp) })

export default socket
```

`channel.on` will "listen" on message topic for a payload and act accordingly `channel.push` let us push on messages topic with a payload

*lib/slang_web/templates/layout/app.html.eex*

```
<html lang="en">
  <head>
    <meta charset="utf-8"/>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>Slang · Phoenix Framework</title>
    <link rel="stylesheet" href="<%= Routes.static_path(@conn, "/css/app.css") %>"/>
    <%= if assigns[:current_user] do %>
        <script>window.userId = "<%= assigns.current_user.id %>";</script>
    <% end %>
  </head>
```

We just need to make the userId available to javascript… that's not the most elegant, but probably the most effective ! :D

Since the channel seems to work, let's finish this Chat !

put some html element to enable the chat

*lib/slang_web/templates/pages/index.html.eex*

```
<div>Awesome Chat :D</div>
<div id="messages"></div>
<input id="chat-input" type="text"></input>
```

remove everything and replace by those 3 lines

now onto to javascript code

*assets/js/socket.js*

```
socket.connect()

// Now that you are connected, you can join channels with a topic:
let channel           = socket.channel("room:lobby", {})
let chatInput         = document.querySelector("#chat-input")
let messagesContainer = document.querySelector("#messages")

function addMessage(payload){
  let messageItem = document.createElement("li")
  messageItem.innerText = ${payload.user_email} : ${payload.msg}
  messagesContainer.appendChild(messageItem)
}

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

channel.on("new_msg", payload => {
  addMessage(payload)
})

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

export default socket
```

Replace everything by this
It's pretty strait forward

On enter in the input box push the message to the channel

Listen for new_msg broadcast and add it to the messagesContainer

And we are done !

startup 2 different browser login with 2 differents user and start chat with yourself !

you can checkout the finished version of the code here on tag `simplistic_chat_finished`

# JSON (Bonus content :D)

We want some Json API of course.
There's quite a lot of way to do that
Easiest is that - route - controller that's it

here is how it goes :

*lib/router.ex*

```
  scope "/api", SlangWeb do
    pipe_through :api

    get "/rooms/:room_id", RoomController, :json_index
  end
```

uncomment this scope that has been lying at the bottom of your router
and add a get route
Notice the :room_id, it's how you parameterize your URLs, it will be passed as a map to your controller :)

*lib/controllers/room_controller.ex*

```
def json_index(conn, %{"room_id" => id}) do
  try do
    %{id: id, description: description, name: name}= Chat.get_room!(id)
    json(conn, %{id: id, description: description, name: name})
  rescue
    _ -> json(conn, %{error: "room not found"})
  end
end
```

We just query the room by its id and render that via the json helper function instead dof the ussual render function
I have to decompose / recompose the Room manually because the json helper won't do it by itself, it's a security hazard to just encode your schema !

# Stimululs - Turbolinks

```
cd assets
npm install --save stimulus turbolinks
```

And remember, Keep Elixiring :)