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
```

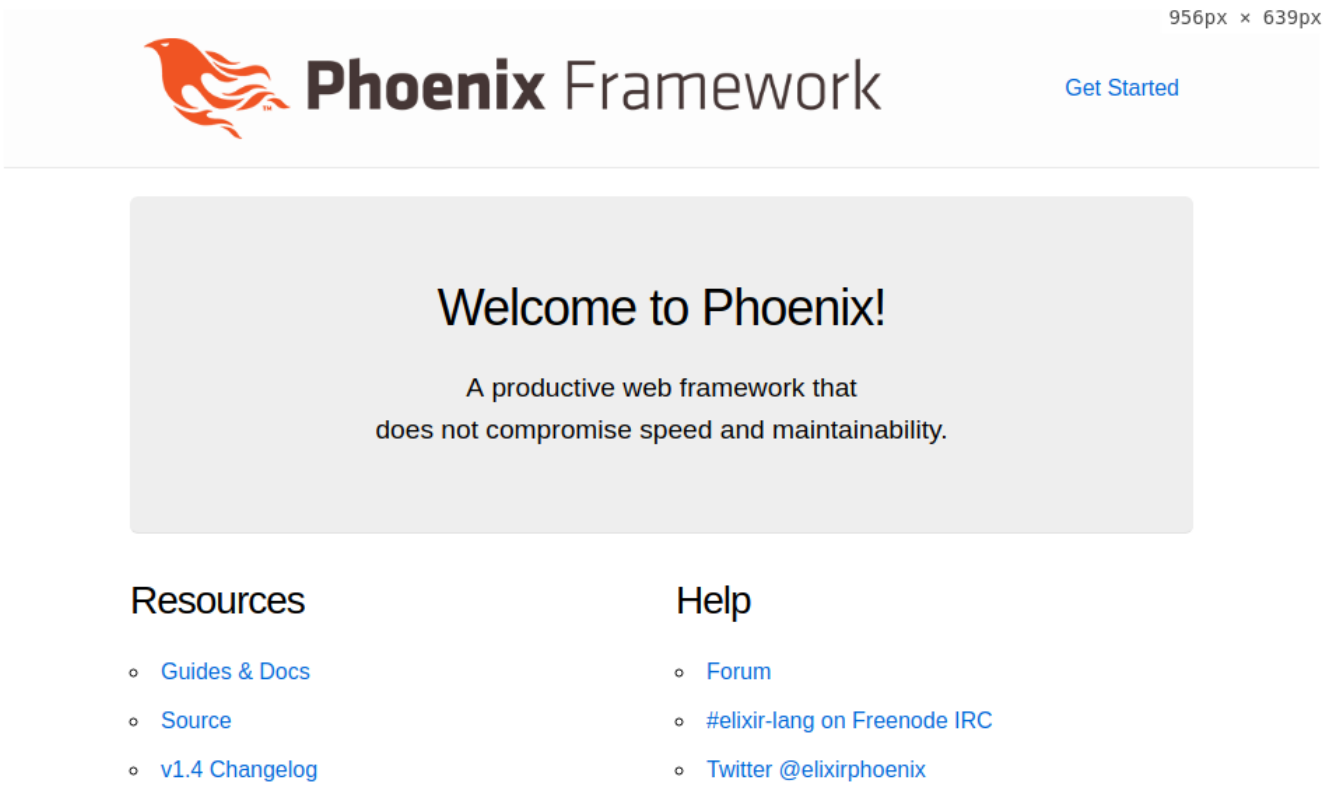
Not too much spoiler, I will get in lengthy details :) Ask me for more during the workshop!

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 Elixirian :)

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

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 Elixirian !

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

```

```

```

```

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

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

And remember, Keep Elixiring :)