Henricus Louwhoff Follow
I build stuff
Oct 28 · 7 min read

Roll your own Email & Password Authentication with Guardian, Comeonin & Phoenix

In this post I'm going to explain that it is relatively easy to implement your own simple email & password authentication with the help of Guardian & Comeonin in a fresh silky-smooth new Guardian app.

At the time of writing I'm using the following stack:

- Elixir v1.3.4
- Phoenix v1.2
- Guardian v0.13
- Comeonin v2.6

I'll try to update the post when a new breaking version is released of either library. (Like the upcoming Phoenix v1.3)

Installing Guardian

Let's start with a new Phoenix app, I believe that you should always start with a fresh new app to test stuff like this and not potentially mess up an existing app. For the sake of this post I'm calling it 'Unicorn'

```
$ mix phoenix.new unicorn
```

The following step would be to configure your database and the rest of your app. I'm not going to cover that in this post because you're probably familiar with it and if not you can alway read the official Up and running guide.

The first thing we're going to do is create an user migration so we can store the newly created user in our database. Run the following command to let Ecto create a new migration file.

```
$ mix ecto.gen.migration create_user
```

The above command will create a file similar to this:

```
priv/repo/migrations/20161107100239_create_user.exs
```

Open that file and add the following code to it:

```
1   defmodule Unicorn.Repo.Migrations.CreateUser do
2     use Ecto.Migration
3
4     def change do
5       create table(:users) do
6         add :email, :string
7         add :password_hash, :string
8
9         timestamps()
10      end
```

Make sure to add line #12 which will create an unique index based upon the email address so we can't have two or more users with the same email address.

Installing Guardian

We're going to use Guardian which will create the JWT's for us to use. Add Guardian to your project's mix file:

```
defp deps do
  [
    # ...
    {:guardian, "~> 0.13.0"}
    # ...
  ]
```

Now install the library by running:

```
$ mix deps.get
```

We’re almost done, we need to add some config for Guardian to work and then we can start coding! Add the following to your `config/config.exs`

```
config :guardian, Guardian,
  allowed_algos: ["HS512"], # optional
  verify_module: Guardian.JWT,  # optional
  issuer: "Unicorn",
  ttl: { 30, :days },
  verify_issuer: true, # optional
```

To create a new `secret_key` you run the following mix command:

```
$ mix phoenix.gen.secret
```

Let’s start coding!

We have an app, ‘Unicorn’, and we have successfully installed Guardian. Great! Let’s roll!

In the Guardian config, we referenced a serializer, but we didn’t write the implementation, so let’s do that now. This module is described on the Guardian site as “The serializer that serializes the ‘sub’ (Subject) field into and out of the token.”

Create a new file `guardian_serializer.ex` in the `lib/unicorn` folder with the following contents:

```elixir
defmodule Unicorn.GuardianSerializer do
  @moduledoc """
  """
  @behaviour Guardian.Serializer

  alias Unicorn.Repo
  alias Unicorn.User

  def for_token(user = %User{}), do: { :ok, "User:#{user.id
  def for_token(_), do: { :error, "Unknown resource type" }
```

As you can see in the code above we need a `User` struct so let's create one.

Run the following mix command to have Phoenix create a new User struct (model) for you:

```
$ mix phoenix.gen.model User users email:string
password_hash:string
```

Phoenix will create three files for you, the model (struct), the migration and the test.

Let's migrate our changes to the database with the mix command:

```
$ mix ecto.migrate
```

When we generated the User struct we created the field `password_hash`. Because it's bad practice to store a plaintext password, we'll only store the password as a hash.

Installing Comeonin

The library `comeonin` will handle the password hashing for us.

Let's add it to our mix file:

```
defp deps do
  [ {:comeonin, "~> 2.6"} ]
end

def application do
  [applications: [:comeonin]]
```

And install it:

```
$ mix deps.get
```

Now that `comeonin` is installed we can start using it in our `User` struct.

Setting up the User struct

Open up `web/models/user.ex` and change it to:

```
1   defmodule Unicorn.User do
2     @moduledoc """
3     """
4     use Unicorn.Web, :model
5
6     alias Comeonin.Bcrypt
7
8     schema "users" do
9       field :email, :string
10      field :password_hash, :string
11
12      field :password, :string, virtual: true
13      field :password_confirmation, :string, virtual: true
14
15      timestamps()
16    end
17
18    @doc "Builds a changeset based on the `struct` and `param
19    def register_changeset(struct, params \\ %{}) do
20      struct
21      |> cast(params, [:email, :password, :password_confirmat
22      |> validate_required([:email, :password, :password_conf
23      |> validate_format(:email, ~r/@/)
24      |> validate_length(:password, min: 8)
25      |> validate_confirmation(:password)
26      |> hash_password()
```

Line #12–13
Add 2 new **virtual** fields here, `password` and `password_confirmation`

Line #21–22
Remove the `password_hash` and added the new virtual fields.

Line #26
Add the function `hash_password`

Line #30–39
Create the `hash_password` function

Let's go over the code.

The first thing we do is add two virtual fields. This is needed because we want the password fields to be used in the changeset but we don't want to persist them to the database. These fields will contain the plaintext version of our password.

The second thing we add is the function `hash_password`. This function will hash the password for us, using Bcrypt, so we can store that hash in the database.

The last thing we added is the actual function to hash the password. This is a straightforward process and I think you'll be able to figure it out by looking at the code.

. . .

Time to make a template and controller to handle your user registration.

First we open the `web/router.ex` file and create a new route for our registration. Make it look something like this:

```elixir
defmodule Unicorn.Router do
  use Unicorn.Web, :router

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

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

Next we create the `UserController` to handle our `new` and `create` functions:

```elixir
defmodule Unicorn.UserController do
  @moduledoc """
  """
  use Unicorn.Web, :controller

  def new(conn, _params) do
  end

```

Let’s start with the `new` function, it’s going to be an easy one because all we want it to do is render a registration form:

```elixir
defmodule Unicorn.UserController do
  @moduledoc """
  """
  use Unicorn.Web, :controller

  alias Unicorn.User

  def new(conn, _params) do
    render conn, "new.html", changeset: User.register_chang
  end
```

When the form is posted, the `create` function is called so let’s create it:

```elixir
defmodule Unicorn.UserController do
  @moduledoc """
  """
  use Unicorn.Web, :controller

  alias Unicorn.User

  def new(conn, _params) do
    render conn, "new.html", changeset: User.register_chang
  end

  def create(conn, %{"user" => user_params}) do
    result=
      %User{}
      |> User.register_changeset(user_params)
      |> Repo.insert()

    case result do
```

We’re half way there, we only need a view and a template to make it all work!

Create `web/views/user_view.ex` with the following contents:

```elixir
defmodule Unicorn.UserView do
  use Unicorn.Web, :view
end
```

And the template in `web/templates/user/new.html.eex`

```
<%= form_for @changeset, user_path(@conn, :create), fn f ->
  <%= if @changeset.action do %>
    <div class="alert alert-danger">
      <p>Oops, something went wrong! Please check the error
    </div>
  <% end %>

  <div class="form-group">
    <%= label f, :email, class: "control-label" %>
    <%= text_input f, :email, class: "form-control" %>
    <%= error_tag f, :email %>
  </div>

  <div class="form-group">
    <%= label f, :password, class: "control-label" %>
    <%= password_input f, :password, class: "form-control"
    <%= error_tag f, :password %>
  </div>
```

Now if you try to start the app you will notice that a warning is displayed:

```
== Compilation error on file
web/controllers/user_controller.ex ==
** (CompileError) web/controllers/user_controller.ex:22:
undefined function session_path/2
```

This is correct because after a successful registration we want to redirect the user to the login form but we haven’t built it yet.

Open up the `web/router.ex` file again and add the session rule to it:

```elixir
defmodule Unicorn.Router do
  use Unicorn.Web, :router

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

  scope "/", Unicorn do
    pipe_through :browser

```

Logging into our app

Now lets create the controller to handle our sessions, create a file called `web/controllers/session_controller.ex`

```elixir
defmodule Unicorn.SessionController do
  @moduledoc """
  """
  use Unicorn.Web, :controller

  def new(conn, _params) do
  end

  def create(conn, params) do
  end
```

At this point we're going to start our app and register a new user.

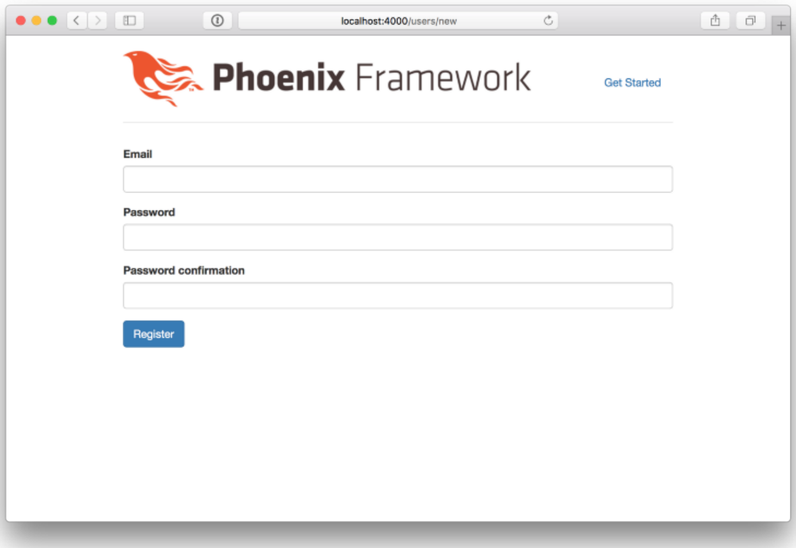
Start your server with the following mix command

```
$ iex -S mix phoenix.server
```

Once your server has been started you can browse to:

http://localhost:4000/users/new

and you should be able to see your brand new shiny registration form!



Enter your details and press ‘Register’. You should receive an error saying:

```
RuntimeError at GET /sessions/new

expected action/2 to return a Plug.Conn, all plugs must
receive a connection (conn) and return a connection
```

That is ok, we made a dummy `SessionController` just to be able to start our app so this is totally expected.

You can check the console to see if an `User` has been created. Go to your console and type:

```
iex(1)> Unicorn.User |> Unicorn.Repo.get(1)
```

This should return an User struct like:

```
%Unicorn.User{__meta__: #Ecto.Schema.Metadata<:loaded,
"users">,
 email: "henry@postb.us", id: 1,
 inserted_at: #Ecto.DateTime<2016-10-28 15:06:59>, password:
```

```
nil,
 password_confirmation: nil,
 password_hash:
"$2b$12$yLfoNSuH9zj8/TwBRQ6qOeR6VO1Wz923oyf.9yREHm/b8SeaenFj
q",
 updated_at: #Ecto.DateTime<2016-10-28 15:06:59>}
```

Let's fix this by opening our `SessionController` again and start adding the login form.

```
defmodule Unicorn.SessionController do
  @moduledoc """
  """
  use Unicorn.Web, :controller

  def new(conn, _params) do
    render conn, "new.html"
  end

  def create(conn, params) do
  end
```

The `new` function is an easy one, the only thing it needs to do is display the login form.

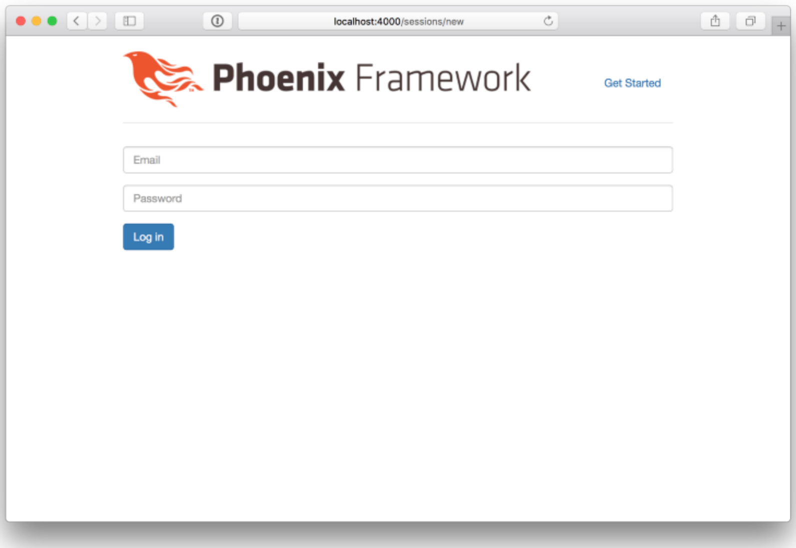
Next thing we need to do it create the view and template.

```
defmodule Unicorn.SessionView do
  use Unicorn.Web, :view
end
```

```
<%= form_for @conn, session_path(@conn, :create), [as: :ses
  <div class="form-group">
    <%= text_input f, :email, placeholder: "Email", class:
  </div>

  <div class="form-group">
    <%= password_input f, :password, placeholder: "Password
  </div>
```

Go back to your browser and when you refresh you should be able to see the login form.



On to the actual logging in part. Open your `SessionController` and let's start adding the logic to handle the login.

```
defmodule Unicorn.SessionController do
  @moduledoc """
  """
  use Unicorn.Web, :controller

  import Comeonin.Bcrypt, only: [checkpw: 2, dummy_checkpw:

  alias Unicorn.User

  def new(conn, _params) do
    render conn, "new.html"
  end

  def create(conn, %{"session" => %{"email" => "", "passwor
    conn
    |> put_flash(:error, "Please fill in an email address a
    |> render("new.html")
  end

  def create(conn, %{"session" => %{"email" => email, "pass
    case verify_credentials(email, password) do
      {:ok, user} ->
        conn
        |> put_flash(:info, "Successfully signed in")
        |> Guardian.Plug.sign_in(user)
        |> redirect(to: admin_page_path(conn, :index))
      {:error, _reason} ->
        conn
        |> put_flash(:error, "Invalid email address or pass
        |> render("new.html")
    end
  end

  def delete(conn, _params) do
    conn
    |> Guardian.Plug.sign_out()
    |> put_flash(:info, "Successfully signed out")
    |> redirect(to: "/")
  end

  defp verify_credentials(email, password) when is_binary(e
```

There is a lot going on in this module but you should take a moment to follow the code flow and see the following steps:

Line #14–18
If email and password are empty we're going to render the login form and notify the user.

Line #20–32
Once an email and password are posted we're going to see if we can find a user based on the email address and if the given password matches the password for that user.

Line #22–26
If an user was found and the password was correct we use Guardian to sign in the user. Guardian creates a session and inserts the user token into the session.

Line #49
If an user was not found we execute the function `dummy_checkpw` so that when someone tries a bunch of email addresses with dummy passwords the timing doesn't change indicating that the email address does not exist in our database.

The next thing we need to do is add Guardian to our `web/router.ex` so we can check if someone has logged in and what his/her user account is.

```elixir
defmodule Unicorn.Router do
  use Unicorn.Web, :router

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

  pipeline :browser_session do
    plug Guardian.Plug.VerifySession
    plug Guardian.Plug.LoadResource
  end

  pipeline :auth do
    plug Guardian.Plug.EnsureAuthenticated, handler: Unicor
  end

  scope "/admin", Unicorn.Admin do
    pipe_through [:browser, :browser_session, :auth]

```

So we added a couple of things.

First a `:browser_session` pipeline with 2 plugs, `VerifySession` which looks for a token in the session and `LoadResources` to look for the sub field of the token, fetches the resource from the Serializer and makes it available via `Guardian.Plug.current_resource(conn)` .

The second pipeline `:auth` contains a plug that looks for a previously verified token. If one is found, continues, otherwise it will call the `:unauthenticated` function of your handler.

Let's create a new file `web/controllers/auth_handler.ex` with the following contents so when an user is not logged in it will redirect to the login page:

```
defmodule Unicorn.AuthHandler do
  @moduledoc """
  """
  use Unicorn.Web, :controller

  def unauthenticated(conn, _params) do
    conn
    |> put_flash(:error, "Authentication required")
```

As you might have noticed in the router file there is an extra `PageController` so lets create a dummy one.

```
defmodule Unicorn.Admin.PageController do
  use Unicorn.Web, :controller

  def index(conn, _params) do
    current_user = Guardian.Plug.current_resource(conn)

    render conn, "index.html"
```

That should be it! I didn't cover every single thing but I'm sure that you will figure it out by looking at the code. Drop me a message if something is wrong or you still have questions.

Thanks

Big thanks go out to Dragica Mandaric for proofreading my first ever blog post and to Jeroen Visser for getting me hooked on Elixir and for squashing many typos and code style!