Test project experience in Elixir - live coding

Machine setup

Before creating a Elixir/Phoenix app we need a few prerequisites installed on our machine. Luckily there's a simple guide for us on how to get those here: https://hexdocs.pm/phoenix/installation.html

Starting a project

Let's start small by bootstraping a simple REST API backed by postgres. Generators to the rescue: https://hexdocs.pm/phoenix/Mix.Tasks.Phx.New.html

First we need to get the generator

```
mix archive.install hex phx_new
```

And then use it to generate our project

```
mix phx.new addressbook --database postgres --no-assets --no-html --no-gettext --no-live --no-mailer --verbose
```

Go through - configs - code architecture - tests - migrations + seeds - mix.exs

Create initial commit

```
git init
git add .
git commit -m "feat: initial project template"
```

Add docker-compose.yaml file with postgres

```
version: '3.9'

services:
  database:
    container_name: addressbook_db
    restart: always
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: addressbook_dev
    volumes:
      - ./.db/init:/docker-entrypoint-initdb.d
    healthcheck:
      test: pg_isready -U postgres -D addressbook_dev -h 127.0.0.1
      interval: 5s
    image: postgres:11.4-alpine
    ports:
      - 5432:5432
    logging:
      driver: none
```

Configure it for our two databases in `.db/init/init.sql`

```
CREATE DATABASE "addressbook_test";
GRANT ALL PRIVILEGES ON DATABASE "addressbook_test" TO postgres;
```

Bring up our database

```
docker-compose up --detached
```

Show how tests are run

```
mix test
```

Show how the local development instance is run

```
mix phx.server
```

Show how the homepage currently looks

```
http://localhost:4000/
```

Show the live dashboard

```
http://localhost:4000/dashboard/home
```

Commit the database changes

```
git add .
git commit -m "feat: dockerized postgres instance"
```

Users & Authentication

It's a good idea to write an authentication system once from scratch, get it peer-reviewed from someone more experienced to learn how things are supposed to work. Mostly all applications require some sort of authentication and there's no need to reinvent the wheel everytime we need one. So let's invite `phx.gen.auth` to the rescue: https://hexdocs.pm/phoenix/Mix.Tasks.Phx.Gen.Auth.html#content

Unfortunately `phx.gen.auth` generates cookie based authentication for web only. Therefore we'll need to do a bit more work, but we can follow the tutorial here: https://njwest.medium.com/jwt-auth-with-an-elixir-on-phoenix-1-3-guardian-api-and-react-native-mobile-app-1bd00559ea51

Add new dependencies

```
      {:comeonin, "~> 5.3"},
      {:bcrypt_elixir, "~> 2.3"},
      {:guardian, "~> 2.2"}
```

Get them

```
mix deps.get
```

Create user JSON CRUD routes

```
mix phx.gen.json Accounts User users email:unique password_hash:string
```

Add new user routes to router

```
resources "/users", UserController, except: [:new, :edit]
```

Apply ecto migrations locally

```
mix ecto.migrate
```

Password Encryption and Field Validation

Add password virtual properties to `lib/addressbook/accounts/user.ex`

```

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

And update the changeset in the same location

```
  def changeset(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)
  end
```

Make sure we hash the provided password before save

```
  import Bcrypt, only: [hash_pwd_salt: 1]

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

Introducing JWT tokens

We can use the guardian library to handle JWT token generation and validation for us. Let's see how that would be done.

Configure guardian `config/config.exs`

```
config :addressbook, Addressbook.Guardian,
       issuer: "Addressbook API",
       secret_key: "Secret key. Use `mix guardian.gen.secret` to generate one"
```

Define basic guardian behavior, create `lib/guardian.ex`

```
defmodule Addressbook.Guardian do
  use Guardian, otp_app: :addressbook

  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 = Addressbook.Accounts.get_user!(id)
    {:ok,  resource}
  end

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

Serve token after registration

Add jwt token JSON renderer into `lib/addressbook_web/views/user_view.ex`

```
 def render("jwt.json", %{jwt: jwt}) do
   %{jwt: jwt}
 end
```

Adjust the create method of user controller to

```
 def create(conn, %{"user" => user_params}) do
   with {:ok, %User{} = user} <- Accounts.create_user(user_params),
   {:ok, token, _claims} <- Guardian.encode_and_sign(user) do
     conn
     |> put_status(:created)
     |> render("jwt.json", jwt: token)
   end
 end
```

Serve token on sign-in

Add a few helper functions to `accounts.ex`

```
 import Bcrypt, only: [verify_pass: 2, no_user_verify: 0]
 alias Addressbook.Guardian

 def token_sign_in(email, password) do
   case email_password_auth(email, password) do
     {:ok, user} ->
       Guardian.encode_and_sign(user)
     _ ->
       {:error, :unauthorized}
   end
 end

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

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

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

And call them appropriately from `user_controller.ex`

```
  def sign_in(conn, %{"email" => email, "password" => password}) do
    case Accounts.token_sign_in(email, password) do
      {:ok, token, _claims} ->
        conn |> render("jwt.json", jwt: token)
      _ ->
        {:error, :unauthorized}
    end
  end
```

Add a route to router

And add a unauthorized error handler to `fallback_controller.ex`

```
  def call(conn, {:error, :unauthorized}) do
    conn
    |> put_status(:unauthorized)
    |> put_view(AddressbookWeb.ErrorView)
    |> render(:"401")
  end
```

Adding authenticated routes

So we've got our JWT tokens ready, let's use them to protect some resources.

We start by creating an authentication pipeline using the Guardian library: addressbook_web/auth_pipeline.ex

```
defmodule Addressbook.Guardian.AuthPipeline do
  use Guardian.Plug.Pipeline, otp_app: :addressbook,
  module: Addressbook.Guardian,
  error_handler: Addressbook.AuthErrorHandler

  plug Guardian.Plug.VerifyHeader, scheme: "Bearer"
  plug Guardian.Plug.EnsureAuthenticated
  plug Guardian.Plug.LoadResource
end
```

And follow-up with the auth error handler: addressbook_web/auth_error_handler.ex

```
defmodule Addressbook.AuthErrorHandler do
  import Plug.Conn

  def auth_error(conn, {type, _reason}, _opts) do
    body = Jason.encode!(%{error: to_string(type)})
    send_resp(conn, 401, body)
  end

end
```

Now it's time to protect some routes in our router:

```
  alias Addressbook.Guardian

  pipeline :jwt_authenticated do
    plug Guardian.AuthPipeline
  end

  scope "/api/v1", AddressbookWeb do
    pipe_through [:api, :jwt_authenticated]

    get "/users/me", UserController, :show
  end
```

And let's demonstrate how we get the user from the request context now: user_controller.ex

```
 def show(conn, _params) do
   user = Guardian.Plug.current_resource(conn)
   conn |> render("user.json", user: user)
end
```