Frameworks and Libraries

Programming Scalable Systems

Ángel Herranz

2022-2023

Universidad Politécnica de Madrid

One framework/library every year

- Interesting libraries and frameworks has a steep learning curve, in general you need months or even years to master
- These slides are just meant to point you in the direction of some interesting libraries
- We keep previous years slides on DETS, ExGram, Ecto or Phoenix
- This year just some names and links are given

2023: a betting platform

Scalability

- Our proposal this year is focussed on scalability

  concurrency, atomicity/serializability, benchmarking, fault-tolerance, process pool, publisher/subscriber, sharding, distribution, etc.

- **Recommendation**: start with a very simple implementation
- **Then** try to think about potential problems (eg. race conditions, bottlenecks, recovering from crashes, etc.)

Resources

- You wil need persistency (DETs and Ecto) and, if you want to UI-interface the platform see ExGram or Phoenix
- Try to solve the problems with your own knowledge:

  concurrency, OTP, distribution, etc.
- Explore built-ins like `Task`, `DynamicSupervisor`, or `Agent`; and some "almost-built-ins" like `Phoenix.PubSub`
- Explore curated lists of awesome libraries:

  `https://elixir.libhunt.com/`

  `https://github.com/h4cc/awesome-elixir`

2022: a weather bot

Activities and assesments i

No exams

Activities and assesments i

No exams

4 voluntary exercise sheets, one per topic

expected personal work every week

(some) submissions will be reviewed in class

Activities and assesments i

No exams

4 voluntary exercise sheets, one per topic
expected personal work every week
(some) submissions will be reviewed in class

Mandatory final exercise sheet, a subset of exercises (20%)

Activities and assesments i

No exams

4 voluntary exercise sheets, one per topic
expected personal work every week
(some) submissions will be reviewed in class

Mandatory final exercise sheet, a subset of exercises (20%)

1 final project (team of two?)
(50% git review + 30% presentation)

Activities and assesments ii

Deadline[1]	Exercises
Week 4	Exercise sheet 1
Week 7	Exercises sheet 2
Week 10	Exercises sheet 3
Week 13	Exercises sheet 4
Week 14	Mandatory exercise sheet[2] (**used to grade**)
	to be published on May 6th, deadline May 20th

[1] To be confirmed.

[2] Most exercises from exercises sheets 1, 2, 3 and 4.

A Project Proposal

- The Telegram Weather Bot Proposal
- Have a look at the statement in Moodle
- But you are encouraged to choose other theme
- Email us your own project ideas for approval – difficulty at least as hard as our proposal
- In this lecture we will explore some convenient libraries (some of the frameworks) needed to tackle any interesting project
- But, to start, you will need to talk to @BotFather:

`https://core.telegram.org/bots`

Libraries for the Bot

ExGram: A Telegram Bot Framework

- ExGram is a powerful framework implemented by a former UPM student `:)`
- The documentation of the library can be found at `https://hexdocs.pm/ex_gram`
- Let's follow the documentation and start a bot

Start a Bot

- Follow instructions

  `https://hexdocs.pm/ex_gram`

- To explore and understand data, change `handle/2`:

```elixir
def handle(message, context) do
  context
  |> answer("CONTEXT: #{inspect(context)}")
  |> answer("MESSAGE: #{inspect(message)}")
end
```

AEMET Framework

- The AEMET has a REST API
- Documentation and credentials: `https://opendata.aemet.es/`
- Main resources to implement the proposal are

  ```
  /api/prediccion/especifica/municipio/horaria/{municipio}
  /api/prediccion/provincia/hoy/{provincia}
  ```

- API doc: `https://opendata.aemet.es/dist/index.html?`
- Let's use Elixir libraries to access the service: Tesla + Jason

Explore the AEMET REST API i

- Ask for your API Key from `https://opendata.aemet.es/centrodedescargas/inicio`
- Use *Tesla*[3] library to access AEMET information, e.g.

```
key = "eyJhbGci..."
municipio = "28026"
endpoint = "https://opendata.aemet.es/opendata"
url = "#{endpoint}/api/prediccion/especifica/municipio/horaria/#{municipio}"
{:ok, response} = Tesla.get url, query: [api_key: key]
json = Jason.decode! response.body
```

[3] Documentation at `https://hexdocs.pm/tesla/readme.html#direct-usage`.

Explore the AEMET REST API ii

- Explore the `json`, you will see that forecasting is not there but in other URL, the URL indicated by `json["datos"]`:

```
url2 = json["datos"]
{:ok, response2} = Tesla.get url, query: [api_key: key]
json2 = Jason.decode! response2.body
```

- And now it is your turn to explore the prediction in `json2`
- **Note:** the body of the response from AEMET is ISO-8859-15 encoded, if you have any problem decoding the body, maybe you will need to transform it to UTF8 (see function in next slide)

Explore the AEMET REST API iii

```elixir
def latin1_to_utf8(latin1) do
  latin1
  |> :binary.bin_to_list()
  |> :unicode.characters_to_binary(:latin1)
end
```

DETS: Persistent ETS

DETS i

- Based on a memory built-in storage named ETS

  Erlang Term Storage

  It is a very quick key-value storage, very convenient for scalability (e.g. cache implementation)
- DETS: *Disk Erlang Term Storage*
- Large API[4] but we just need four *functions*: `:dets.open_file`, `:dets.insert`, `:dets.lookup`, and `:dets.delete`, and

[4] `https://erlang.org/doc/man/dets.html`

DETS ii

- `:dets.open_file`: Opens a table. An empty Dets table is created if no file exists.

```
table = :dets.open_file(:users, file: '/tmp/users')
```

- `:dets.insert`: Inserts one or more objects into the table. If there already exists an object with a key matching the key of some of the given objects and the table type is set, the old object will be replaced.

```
:dets.insert(table, {username, password})
```

- `:dets.lookup`: Returns a list of all objects with key stored in table.

```
:dets.lookup(table, username)
```

- `:dets.delete`: Deletes all objects with key from table.

```
:dets.delete(table, username)
```

DETS iii

limited support for

concurrent access

(use a server to serialize access)

2021: a twitter clone

ECTO: a Database Toolkit

Ecto

- Ecto is *the* database library of Elixir
- Exposes a relational model (SQL databases)
- The name ORM[5] is used in other languages
- But should not be used in Elixir, why?
- Inspired by *history*: mainly LINQ in .NET (queries) and ActiveRecord in Rails (migrations and schemas)
- Extensive use of metaprogramming: a very convenient DSL[6]

[5] Object-relational mapping

[6] Domain Specific Language

Ecto is actually big

- Let's just introduce the bureaucratic part and some of the main concepts (bellow), you will need to explore documentation
- Schema: mapping structs and database tables
- Repo: adoption of the repository pattern (get, insert, update, delete)
- Changeset: representation of changing data
- Query: database queries can be incrementally created
- Try to follow me! If you cannot, ask me to explore!

Getting Started i

- We need a database engine installed: PostgreSQL
- Ready for reading: `https://hexdocs.pm/ecto/Ecto.html`
- Add to the dependencies in `mix.exs`

```
defp deps do
  [
    {:ecto_sql, "~> 3.0"},
    {:postgrex, ">= 0.0.0"}
  ]
end
```

Getting Started ii

- `mix deps.get`
- Connect your project to the database in `config/config.exs`

```
config :twitter, Twitter.Repo,
  database: "twitter",
  username: "twitteradm",
  password: "verysecret",
  hostname: "localhost"
```

Getting Started iii

- Create your repo in `lib/twitter/repo.ex`

```
defmodule Twitter.Repo do
  use Ecto.Repo,
    otp_app: :twitter,
    adapter: Ecto.Adapters.Postgres
end
```

- And add it to the supervisor tree in `lib/twitter/application.ex`

Getting Started iv

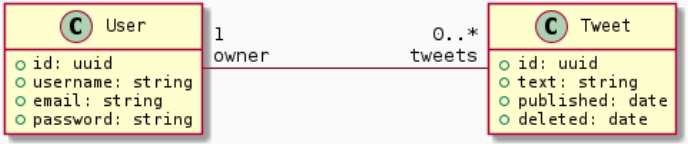
- Check/create user and password in the database:

```
sudo -u postgres psql
```

```
postgres=# create user twitteradm with encrypted
   password 'verysecret';
CREATE ROLE
postgres=# alter user twitteradm createdb;
ALTER ROLE
```

- Create the database: `mix ecto.create`

Tweets and Users: our model



Create the schema for *User* i

```
defmodule Twitter.User do
  use Ecto.Schema

  schema "users" do
    field :username, :string
    field :email, :string
    field :password, :string
  end
end
```

Create the schema for *User* ii

```
iex> alias Twitter.User
iex> %User{}
iex> alias Twitter.Repo
iex> Repo.insert(%User{})
```

Create the schema for *User* iii

```
$ mix ecto.gen.migration create_users
```

And make the migration to create the table:

Ecto.Migration.{create, table, add}

```
def change do
  create table(:users) do
    add(:username, :string)
    add(:email, :string)
    add(:password, :string)
  end
end
```

Create the schema for *User* iv

```
iex> Repo.insert(%User{})

02:06:28.991 [debug] QUERY OK db=220.6ms queue=0.8ms idle=1161.9ms
INSERT INTO "users" VALUES (DEFAULT) RETURNING "id" []

{:ok,
 %Twitter.User{
   __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
   email: nil,
   id: 1,
   password: nil,
   username: nil
 }}
```

Finetuning the migration

- `username`, `email` and likely `password` should not be null.
- Let's have a look at

  `hexdocs.pm/ecto_sql/Ecto.Migration.html#add/3`

  `hexdocs.pm/ecto_sql/Ecto.Migration.html#index/3`
- In the migration:

```
add(:username, :string, null: false)
create index(:users, :username, unique: true)
```

- Try inserting an empty *`User`* again

Seeding data

- Convention: `priv/repo/seeds.exs`

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

- Let's fill the database with some users
- In the future, you an explore library ExMachina

Querying the repo

- Explore some callbacks in *Ecto*:

```
all(User)
get(User, id)
get!(User, id)
get_by(User, username: un)
delete(User, id)
```

Associations 1:N i

- We expect a very convenient way to access *properties*:

```
user.tweets
tweet.owner
```

Associations 1:N ii

```elixir
defmodule Twitter.Tweet do
  use Ecto.Schema

  schema "tweets" do
    field :text, :string
    field :published, :date
    field :deleted, :date

    belongs_to :owner, Twitter.User
  end
end
```

```elixir
defmodule Twitter.User do
  use Ecto.Schema

  schema "users" do
    field :username, :string
    field :email, :string
    field :password, :string

    has_many :tweets, Twitter.Tweet,
      foreign_key: :owner_id
  end
end
```

Associations 1:N iii

```elixir
defmodule Twitter.Repo.Migrations.CreateTweets do
  use Ecto.Migration

  def change do
    create table(:tweets) do
      add(:text, :string, null: false)
      add(:published, :naive_datetime)
      add(:deleted, :naive_datetime)
      add(:owner_id, references(:users), null: false)
    end
  end
end
```

Seeding data + reset

- Add tweets to the seeds: `priv/repo/seeds.exs`
- Reset the database:

```
$ mix do ecto.drop, ecto.create, ecto.mgirate, run
   priv/repo/seeds.exs
```

- Let's play a bit with the properties

Phoenix: a Web Framework

Phoenix

- Phoenix is a library[7] that enables you to write Web applications and Web services (e.g. REST)

  Web apps and Web services are based on HTTP

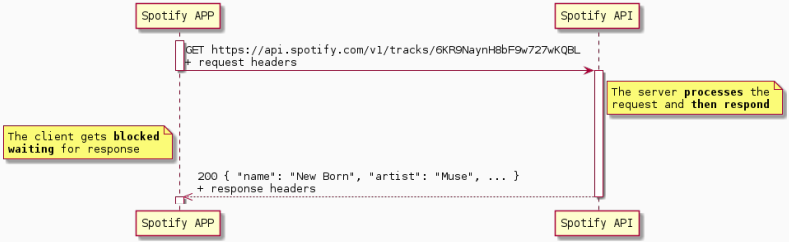
- Each language has its own Web framework, to name some:

  Ruby (on Rails), *Spring/Java*, *Django/Python*, *Laravel/PHP*, *Next/Javascript*, etc.

[7] In general referred as a **framework**, because it also includes a lot of conventions.

HTTP

- HTTP is the protocol of the Web
- HTTP is big (infinite details!)
- HTTP is synchronous[8]:



[8]This has an important impact in how web applications are created.

How does a Web App Work?

- Open your favourite Web and try F12 in your browser[9]
- The starting point is a HTML document,
- the browser traverses the HTML document,
- creates a structure to represent the document (the DOM tree),
- downloads other documents (CSS, JS, images, etc.) according to information in HTML,
- renders the DOM using the CSS files,
- run Javascript code in the original HTML

[9] `www.upm.es` ;)

Let's explore!

- Thousands of hours to learn Phoenix (any framework)
- Good documentation: `www.phoenixframework.org`
- Good books, e.g. *Programming Elixir*

Let's explore!

- Thousands of hours to learn Phoenix (any framework)
- Good documentation: `www.phoenixframework.org`
- Good books, e.g. *Programming Elixir*
- We will just explore

Let's explore!

Recommendations

- You can try to follow me
- If you can't, stop trying it and just pay attention
- If you want me to experiment, ask me for

Our first Web app in 1 min i

- Install the Phoenix Application Generator:

```
$ mix archive.install hex phx_new 1.5.8
$
```

- Install Node.js and npm[10]
- Install `inotify-tools` for *live reloading* (automatic reload of the page when the Elixir source code changes!)
- PostgreSQL no needed for the moment

[10] Try a *modern* version of Node.js, Phoenix will use it to compile CSS and Javascript code that will run in the browser.

Our first Web app in 1 min ii

- Create a Phoenix project: our own *Twitter*

```
$ mix phx.new twitter --no-ecto
$
```

- A project has been created, follow last instruction:

```
$ cd twitter
$ iex -S mix phx.server
...
iex(1)>
```

- And then visit `http://localhost:4000`

Look around: `cd twitter; tree`

```
.
|-- config
|   |-- config.exs
|   |-- dev.exs
|   |-- prod.exs
|   |-- prod.secret.exs
|   `-- test.exs
|-- mix.exs
|-- priv
|   `-- gettext
|       |-- en
|       |   `-- LC_MESSAGES
|       |       `-- errors.po
|       `-- errors.pot
|-- README.md
```

```
.
|-- lib
|   |-- twitter
|   |   `-- application.ex
|   |-- twitter.ex
|   |-- twitter_web
|   |   |-- channels
|   |   |   `-- user_socket.ex
|   |   |-- controllers
|   |   |   `-- page_controller.ex
|   |   |-- endpoint.ex
|   |   |-- gettext.ex
|   |   |-- router.ex
|   |   |-- telemetry.ex
|   |   |-- templates
|   |   |   |-- layout
|   |   |   |   `-- app.html.eex
|   |   |   `-- page
|   |   |       `-- index.html.eex
|   |   `-- views
|   |       |-- error_helpers.ex
|   |       |-- error_view.ex
|   |       |-- layout_view.ex
|   |       `-- page_view.ex
|   `-- twitter_web.ex
```

```
.
|-- assets
|   |-- css
|   |   |-- app.scss
|   |   `-- phoenix.css
|   |-- js
|   |   |-- app.js
|   |   `-- socket.js
|   |-- package.json
|   |-- static
|   |   |-- favicon.ico
|   |   |-- images
|   |   |   `-- phoenix.png
|   |   `-- robots.txt
|   |-- vendor
    `-- webpack.config.js
```

Configuration of the app

- `README.exs`: instructions to start
- `mix.exs`: version, dependencies, ...
- `config/config.exs`: common configuration
- `config/dev.exs`: config for development environment (port, live reloading, secrets, ...)
- `lib/twitter/application.ex`: the Erlang application (OTP application and supervisor)

Why `localhost:4000/`?

- config/dev.exs

```
config :twitter, TwitterWeb.Endpoint,
  http: [port: 4000],
```

- lib/twitter_web/router.ex

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

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

Router: path + controller + action

- `router.ex` says

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

When the client visit URL `/`,
content will be rendered
with action `:index`
in controller `PageController`,
i.e. function `PageController.index/2`

Action in controller

- lib/twitter_web/controllers/page_controller.ex

```
defmodule TwitterWeb.PageController do
  use TwitterWeb, :controller

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

Assume `@spec index(Plug.Conn.t(), map())::` `Plug.Conn.t()` where `Plug.Conn.t()` values represents HTTP connections: *request + response*

The *magic*: views and templates

- Because of the controller's name "page" (`page_controller.ex`),
- and because of the call to `render(conn, "index.html")` in `PageController.index/2`,
- Phoenix considers two files must exist:
  1. `lib/twitter_web/views/page_view.ex`
  2. `lib/twitter_web/templates/page/index.html.eex`

The *view*

- `lib/twitter_web/views/page_view.ex` is just a module:

```
defmodule TwitterWeb.PageView do
  use TwitterWeb, :view
end
```

- Functions in this module are accessible from the template

The *template*

- Templates are written in a template language[11]:

  Embedded Elixir (*EEx*)

- You will need to learn *EEx*
- You will need to learn HTML too
- Both are intuitive enough to understand the file, so let's explore: `lib/twitter_web/templates/page/index.html.eex`

[11] Every framework has its own template language.

Layouts i

- Go to `localhost:4000` in your browser
- Open *Web Developer Tools*, in general `Ctr-Shift-I` or F12
- Let's discover what the server actually sent to the client
  1. Tab network > clear > reload
  2. Explore the response to the first HTTP request (File: `/`)
- Try to identify content with respect to `index.html.eex`
- What about the HTML around the content?

Layouts ii

- HTML content is rendered inside a layout
- A layout is just a template (+ its view)
- The default layout is

  `lib/twitter_web/views/layout_view.ex`

  `lib/twitter_web/templates/layout/app.html.eex`
- Let's explore and find the EEx construct

```
<%= @inner_content %>
```

The *front-end*

- The browser receives and processes a lot of documents:

  HTML + CSS + JS

  1. It creates a DOM from the HTML,
  2. executes the JS code (might change the DOM!),
  3. renders the DOM by using the CSS,
  4. in a loop: listens to UI events and runs code (predefined behaviour like links or submits or specific registered JS callbacks)
- This is the front-end

The *front-end* in our app i

- In `lib/twitter_web/templates/layout/app.html.eex`:

```
<link rel="stylesheet" href="<%= Routes.static_path(@conn, "/css/app.css") %>"/>
<script defer type="text/javascript" src="<%= Routes.static_path(@conn, "/js/app.js") %>"></script>
```

- Syntax `<%= E %>` is EEx: replace with result of expression *E*
- *`Routes`*`.static_path/2` returns the path to *static* content[12]
- Static content is extracted from directory `assets`, let's explore!

[12] Non dynamically generated content such as CSS, JS, images.

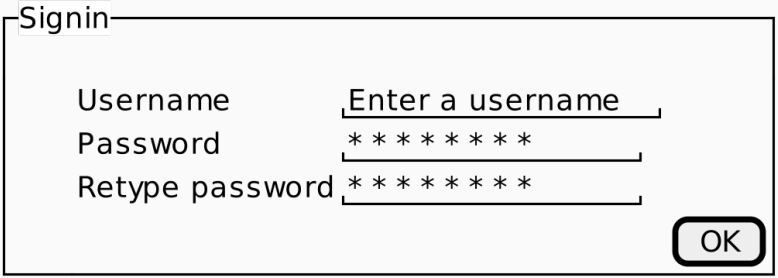
The *front-end* in our app ii

- We will not teach HTML, CSS or JS in this course
- We will not use JS :)
- We will use Milligram, the default *CSS framework* in the project:

  `https://milligram.io/`
- We will use HTML as Milligram examples indicates

Signup (aka registration)

localhost:4000/signup



New controller for signup and login

- Create a new controller *access* (to control signup and login)
- Create its view and template
- Find inspiration in `https://milligram.io/#forms`
- Finaly, let's try a different layout (`access.html.eex`):

```
defmodule TwitterWeb.AccessController do
  use TwitterWeb, :controller
  plug :put_layout, "landing.html" # Affects every action!
  ...
end
```

conn and params

- Let's try to understand conn and params:

```
defmodule TwitterWeb.AccessController do
  require Logger
  ...
  def signup(conn, params) do
    Logger.debug("CONN: #{inspect conn}")
    Logger.debug("PARAMS: #{inspect params}")

    conn |> render("signup.html")
  end
  ...
end
```

The signup web page

- Find inspiration in `https://milligram.io/#forms`
- Copy, Paste & adapt it to username and passwords
- Try the click and look at the server console!

The signup web page

- Find inspiration in `https://milligram.io/#forms`
- Copy, Paste & adapt it to username and passwords
- Try the click and look at the server console!
- Ignore attribute `for` of `<label>`
- Ignore attribute `id` of `<input>`
- Add attribute `name` to `<input>`
- Try again: reload then fill the form then click!

POST /signup

- Difficult to know if the client wants to load the page or submit a registration request (same route!)
- HTTP has two main operations named *verbs* that help
- GET "to request data from a specified resource."
- POST "to send data to a server to create/update a resource."
- So let's try to *POST* instead of to *GET*:

```
<form method="post">...
```

Route to POST /signup

no route found for POST /signup (TwitterWeb.Router)

- Function **get** in router means HTTP verb GET:

```
get "/signup", AccessController, :signup
```

- Let's route the POST request to the same *action*
- Add the route

```
post "/signup", AccessController, :signup
```

Library Phoenix.HTML i

invalid CSRF (Cross Site Request Forgery) token, . . .

- The error is related to the defense of a security attack[13]
- Phoenix expects you to write HTML code with `Phoenix.HTML`:

```
Phoenix.HTML.{Form, Link, Tag}
```

- A lot of security aspects are covered by `Phoenix.HTML`
- Let's write HTML using this library

[13] More information in CSRF.

Library Phoenix.HTML i

```
<%= form_for @conn, Routes.access_path(@conn, :signup), fn f -> %>
  <fieldset>
    <%= label f, :username, "Username" %>
    <%= text_input f, :username, placeholder: "Choose your username!" %>
    <%= label f, :password, "Password" %>
    <%= password_input f, :password, placeholder: "* * * * * * * *" %>
    <%= label f, :repassword, "Retype password" %>
    <%= password_input f, :repassword, placeholder: "* * * * * * * *" %>
    <%= submit "OK", class: "button button-primary float-right" %>
  </fieldset>
<% end %>
```

- Every entry "`<%= ... %>`" will be expanded by EEx
- Explore the resulting HTML sent to client!

An action for user creation

- Much better introducing a different action for GET and POST

```
post "/signup", AccessController, :register
```

- Let's write the logic for POST (`AccessController.register`)
  1. Check password is *correct*
  2. Check username is not registered
  3. Register the username (with its password)
- We need a persistency layer! (insert, update and lookup)
- We will use a key-value database: DETS

Context *Users* i

- Term "context" from Eric Evans DDD
- Module to isolate controllers from the concrete database
- `Twitter.Users` in `lib/twitter/users.ex` (no `twitter_web`)
- We will use `use GenServer` to avoid concurrent access to DETS[14]
- API: `insert`, `update`, `get`, and `delete`

[14] This is a design decision, the context could be implemented in a different way.

Context *Users* ii

```elixir
defmodule Twitter.Users do
  use GenServer

  # API
  def start_link(_) do
    GenServer.start_link(__MODULE__, :ok, name: __MODULE__)
  end

  def insert(username, password) do
    GenServer.call(__MODULE__, {:insert, username, password})
  end

  def update(username, password) do
    GenServer.call(__MODULE__, {:update, username, password})
  end

  def get(username) do
    GenServer.call(__MODULE__, {:get, username})
  end

  def delete(username) do
    GenServer.call(__MODULE__, {:delete, username})
  end

  # GenServer callbacks

  def init(_) do
    :dets.open_file(__MODULE__, file: '/tmp/users')
  end

  def handle_call({:insert, username, password}, _from, table) do
    reply = :dets.insert_new(table, {username, password})
    {:reply, reply, table}
  end

  def handle_call({:update, username, password}, _from, table) do
    case :dets.lookup(table, username) do
      [] ->
        {:reply, {:error, :not_found, nil}, table}

      _ ->
        reply = :dets.insert(table, {username, password})
        {:reply, reply, table}
    end
  end

  ...
end
```

Context *Users* iii

- Start the *GenServer supervised* by the *Application* by adding *Twitter.Users* as a child of the Supervisor at `lib/twitter/application.ex`

```
children = [
  # Start the Telemetry supervisor
  TwitterWeb.Telemetry,
  # Start the PubSub system
  {Phoenix.PubSub, name: Twitter.PubSub},
  # Start the Endpoint (http/https)
  TwitterWeb.Endpoint,
  # Start a worker by calling: Twitter.Worker.start_link(arg)
  Twitter.Users
]
```

Redirect

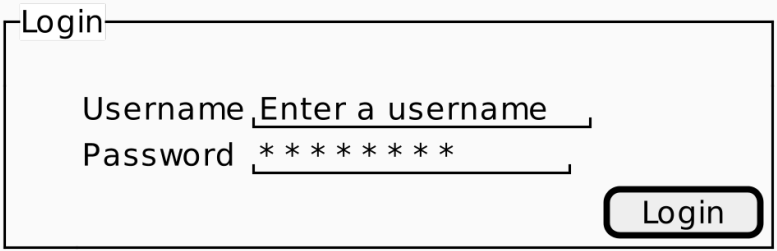
- When registration successes, the user must be redirected to a proper page, maybe the `/login` page:

```
def register(conn, params) do
  ## TODO: validate username, password and repassword
  Twitter.Users.insert(params["username"], params["password"])
  redirect(conn, to: "/login")
end
```

- As `render`, function redirect is accessible in every controller
- Have a look at documention of *Phoenix.Controller*

  `https://hexdocs.pm/phoenix/Phoenix.Controller.html`

localhost:4000/login



localhost:4000/login

- Once the use has successfuly logged-in main page must present the main page of out application
- *Profile* information and *followed's* posts at least!
- You are to explore the most relevant elements in `conn`:

  Assigns: used to pass information to templates

  Sessions: used to keep *session* information (e.g. knowing whether a user is logged-in and which user is logged-in)

  Flashes: used to *flash error/info UI messages*

Assigns, Sessions and Flashes

- Explore functions in *Plug.Conn* for *assigns* and *sessions*:

  `https://hexdocs.pm/phoenix/Plug.Conn.html`

- `assign`: adds a values to the `:assigns` key that will be accessible from the templates
- `put_session` and `delete_session`: adds/remove keys to the session (passed in the cokies to the client)
- *Phoenix.Controller*.`put_flash`: adds a value in *flash* (available in template with `get_flash`)