



Pow is a robust, modular, and extendable authentication and user management solution for Phoenix and Plug-based apps.

Features

- User registration
- Session based authorization
- Per Endpoint/Plug configuration
- API token authorization
- Mnesia cache with automatic cluster healing
- Multitenancy
- User roles
- Extendable
- I18n
- And more

Installation

Add Pow to your list of dependencies in `mix.exs`:

```
1 defp deps do
2   [
3     # ...
4     {:pow, "~> 1.0.38"}
5   ]
6 end
```

Run `mix deps.get` to install it.

Getting started

Phoenix app

Umbrella project: Check out the umbrella project guide.

Install the necessary files:

```
1 mix pow.install
```

This will add the following files to your app:

```
1 LIB_PATH/users/user.ex
2 PRIV_PATH/repo/migrations/TIMESTAMP_create_users.ex
```

And also update the following files:

```
1 config/config.exs
2 WEB_PATH/endpoint.ex
3 WEB_PATH/router.ex
```

Run migrations with `mix setup`, start the server with `mix phx.server`, and you can now visit <http://localhost:4000/registration/new> to create a user.

Modify templates

By default, Pow exposes as few files as possible.

If you wish to modify the templates, you can generate them using:

```
1 mix pow.phoenix.gen.templates
```

This will also add `web_module: MyAppWeb` to the configuration in `config/config.exs`.

Extensions

Pow is made so it's easy to extend the functionality with your own complimentary library. The following extensions are included in this library:

- PowResetPassword
- PowEmailConfirmation
- PowPersistentSession
- PowInvitation

Check out the “Other libraries” section for other extensions.

Add extensions support

To keep it easy to understand and configure Pow, you'll have to enable the extensions yourself.

Let's install the `PowResetPassword` and `PowEmailConfirmation` extensions.

First, install extension migrations by running:

```
1 mix pow.extension.ecto.gen.migrations --extension PowResetPassword --  
  extension PowEmailConfirmation
```

Then run the migrations with `mix ecto.migrate`. Now, update `config/config.ex` with the `:extensions` and `:controller_callbacks` key:

```
1 config :my_app, :pow,  
2   user: MyApp.Users.User,  
3   repo: MyApp.Repo,  
4   extensions: [PowResetPassword, PowEmailConfirmation],  
5   controller_callbacks: Pow.Extension.Phoenix.ControllerCallbacks
```

Update `LIB_PATH/users/user.ex` with the extensions:

```
1 defmodule MyApp.Users.User do  
2   use Ecto.Schema  
3   use Pow.Ecto.Schema  
4   use Pow.Extension.Ecto.Schema,  
5     extensions: [PowResetPassword, PowEmailConfirmation]  
6  
7   # ...  
8  
9   def changeset(user_or_changeset, attrs) do  
10     user_or_changeset  
11     |> pow_changeset(attrs)  
12     |> pow_extension_changeset(attrs)  
13   end  
14 end
```

Add Pow extension routes to `WEB_PATH/router.ex`:

```
1 defmodule MyAppWeb.Router do  
2   use MyAppWeb, :router  
3   use Pow.Phoenix.Router  
4   use Pow.Extension.Phoenix.Router,  
5     extensions: [PowResetPassword, PowEmailConfirmation]  
6  
7   # ...  
8  
9   scope "/" do  
10     pipe_through :browser  
11   end
```

```
12     pow_routes()
13     pow_extension_routes()
14 end
15
16 # ...
17 end
```

Modify extension templates Templates for extensions can be generated with:

```
1 mix pow.extension.phoenix.gen.templates --extension PowResetPassword --
  extension PowEmailConfirmation
```

Please follow the instructions in “Modify templates” to ensure that your custom templates will be used.

Mailer support

Many extensions require a mailer to have been set up. Let’s create a mailer mock module in `WEB_PATH/mails/pow/mailer.ex`:

```
1 defmodule MyAppWeb.Pow.Mailer do
2   use Pow.Phoenix.Mailer
3   require Logger
4
5   def cast(%{user: user, subject: subject, text: text, html: html,
6     assigns: _assigns}) do
7     # Build email struct to be used in `process/1`
8     %{to: user.email, subject: subject, text: text, html: html}
9   end
10
11   def process(email) do
12     # Send email
13
14     Logger.debug("E-mail sent: #{inspect email}")
15   end
16 end
```

Update `config/config.ex` with `:mailer_backend` key:

```
1 config :my_app, :pow,
2   # ...
3   mailer_backend: MyAppWeb.Pow.Mailer
```

This mailer module will only output the mail to your log, so you can e.g. try out the reset password and email confirmation links. You should integrate the Pow mailer with your actual mailer system. For

Swoosh or Bamboo integration, check out the [Configuring mailer guide](#).

Modify mailer templates Generate the template files:

```
1 mix pow.extension.phoenix.mailer.gen.templates --extension  
   PowResetPassword --extension PowEmailConfirmation
```

This will generate template files in the `WEB_PATH/mails/` directory. This will also add the necessary `mail/0` macro to `WEB_PATH/my_app_web.ex` and update the pow config with `web_mailer_module: MyAppWeb`.

Configuration

Pow is built to be modular, and easy to configure. The configuration is passed to function calls as well as plug options, and they will take priority over any environment configuration. It's ideal in case you got an umbrella app with multiple separate user domains.

The easiest way to use Pow with Phoenix is to use a `:otp_app` in function calls and set the app environment configuration. It will keep a persistent fallback configuration that you configure in one place.

Module groups

Pow has three main groups of modules that each can be used individually, or in conjunction with each other:

Pow.Plug This group will handle the plug connection. The configuration will be assigned to `conn.private[:pow_config]` and passed through the controller to the users' context module. The Plug module has functions to authenticate, create, update, and delete users, and will generate/renew the session automatically.

Pow.Ecto This group contains all modules related to the Ecto based user schema and context. By default, Pow will use the `Pow.Ecto.Context` module to authenticate, create, update and delete users with lookups to the database. However, it's straightforward to extend or write your custom user context. You can do this by setting the `:users_context` configuration key.

Pow.Phoenix This group contains the controllers and templates for Phoenix. You only need to set the (session) plug in `endpoint.ex` and add the routes to `router.ex`. Templates are not generated by default, instead, the compiled templates in Pow are used. You can generate the templates used by running `mix pow.phoenix.gen.templates`. You can also customize flash messages and callback routes by creating your own using `:messages_backend` and `:routes_backend`.

The registration and session controllers can be changed with your customized versions too, but since the routes are built on compile time, you'll have to set them up in `router.ex` with `:pow` namespace. For minor pre/post-processing of requests, you can use the `:controller_callbacks` option. It exists to make it easier to modify flow with extensions (e.g., send a confirmation email upon user registration).

Pow.Extension

This module helps build extensions for Pow. There're three extension mix tasks to generate Ecto migrations and phoenix templates.

```
1 mix pow.extension.ecto.gen.migrations
```

```
1 mix pow.extension.phoenix.gen.templates
```

```
1 mix pow.extension.phoenix.mailer.gen.templates
```

Authorization plug

Pow ships with a session plug module. You can easily switch it out with a different one. As an example, here's how you do that with `Phoenix.Token`:

```
1 defmodule MyAppWeb.Pow.Plug do
2   use Pow.Plug.Base
3
4   @session_key :pow_user_token
5   @salt "user salt"
6   @max_age 86400
7
8   def fetch(conn, config) do
9     conn = Plug.Conn.fetch_session(conn)
10    token = Plug.Conn.get_session(conn, @session_key)
11
12    MyAppWeb.Endpoint
13    |> Phoenix.Token.verify(@salt, token, max_age: @max_age)
14    |> maybe_load_user(conn)
15  end
```

```
16
17   defp maybe_load_user({:ok, user_id}, conn), do: {conn, MyApp.Repo.get
      (User, user_id)}
18   defp maybe_load_user({:error, _any}, conn), do: {conn, nil}
19
20   def create(conn, user, config) do
21     token = Phoenix.Token.sign(MyAppWeb.Endpoint, @salt, user.id)
22     conn =
23       conn
24       |> Plug.Conn.fetch_session()
25       |> Plug.Conn.put_session(@session_key, token)
26
27     {conn, user}
28   end
29
30   def delete(conn, config) do
31     conn
32     |> Plug.Conn.fetch_session()
33     |> Plug.Conn.delete_session(@session_key)
34   end
35 end
36
37 defmodule MyAppWeb.Endpoint do
38   # ...
39
40   plug MyAppWeb.Pow.Plug, otp_app: :my_app
41 end
```

Ecto changeset

The user module has a fallback `changeset/2` function. If you want to add custom validations, you can use the `pow_changeset/2` function like so:

```
1 defmodule MyApp.Users.User do
2   use Ecto.Schema
3   use Pow.Ecto.Schema
4
5   schema "users" do
6     field :custom, :string
7
8     pow_user_fields()
9
10    timestamps()
11  end
12
13  def changeset(user_or_changeset, attrs) do
14    user_or_changeset
15    |> pow_changeset(attrs)
16    |> Ecto.Changeset.cast(attrs, [:custom])
```

```
17     |> Ecto.Changeset.validate_required([:custom])
18   end
19 end
```

Phoenix controllers

Controllers in Pow are very slim and consists of just one `Pow.Plug` function call with response functions. If you wish to change the flow of the `Pow.Phoenix.RegistrationController` and `Pow.Phoenix.SessionController`, the best way is to create your own and modify `router.ex`.

However, to make it easier to integrate extension, you can add callbacks to the controllers that do some light pre/post-processing of the request:

```
1 defmodule MyCustomExtension.Phoenix.ControllerCallbacks do
2   use Pow.Extension.Phoenix.ControllerCallbacks.Base
3
4   def before_respond(Pow.Phoenix.RegistrationController, :create, {:ok,
5     user, conn}, _config) do
6     # send email
7     {:ok, user, conn}
8   end
9 end
```

You can add functions for `before_process` / 4 (before the action happens) and `before_respond` / 4 (before parsing the results from the action).

Testing with authenticated users To test with authenticated users in your controller tests, you just have to assign the user to the conn in your setup callback:

```
1 setup %{conn: conn} do
2   user = %User{email: "test@example.com"}
3   conn = Pow.Plug.assign_current_user(conn, user, otp_app: :my_app)
4
5   {:ok, conn: conn}
6 end
```

l18n

All templates can be generated and modified to use your Gettext module.

For flash messages, you can create the following module:

```
1 defmodule MyAppWeb.Pow.Messages do
```

```
2 use Pow.Phoenix.Messages
3 use Pow.Extension.Phoenix.Messages,
4   extensions: [PowResetPassword]
5
6 import MyAppWeb.Gettext
7
8 def user_not_authenticated(_conn), do: gettext("You need to sign in
9   to see this page.")
10
11 # Message functions for extensions has to be prepended with the snake
12 # cased
13 # extension name. So the `email_has_been_sent/1` function from
14 # `PowResetPassword` is written as `
15 #   pow_reset_password_email_has_been_sent/1`
16 # in your messages module.
17 def pow_reset_password_email_has_been_sent(_conn), do: gettext("An
18   email with reset instructions has been sent to you. Please check
19   your inbox.")
20 end
```

Add `messages_backend: MyAppWeb.Pow.Messages` to your configuration. You can find all the messages in `Pow.Phoenix.Messages` and `[Pow Extension].Phoenix.Messages`.

Callback routes

You can customize callback routes by creating the following module:

```
1 defmodule MyAppWeb.Pow.Routes do
2   use Pow.Phoenix.Routes
3   use MyAppWeb, :verified_routes
4
5   def after_sign_in_path(conn), do: ~p"/home"
6 end
```

Add `routes_backend: MyAppWeb.Pow.Routes` to your configuration. You can find all the routes in `Pow.Phoenix.Routes`.

Password hashing function

`Pow.Ecto.Schema.Password` is used to hash and verify with Pbkdf2. It's highly recommended to lower the iterations count in your test environment to speed up your tests:

```
1 config :pow, Pow.Ecto.Schema.Password, iterations: 1
```

You can change the password hashing function easily. For example, this is how you use `comeonin` with `Argon2`:

```
1 defmodule MyApp.Users.User do
2   use Ecto.Schema
3   use Pow.Ecto.Schema,
4     password_hash_verify: {&Argon2.hash_pwd_salt/1,
5                           &Argon2.verify_pass/2}
6
7   # ...
8 end
```

Current user and sign out link

You can use `Pow.Plug.current_user/1` to fetch the current user from the connection.

This can be used to show the sign in or sign out links in your Phoenix template:

```
1 <.link :if={Pow.Plug.current_user(@conn)} href={~p"/session"} method="
  delete">Sign out</.link>
2 <.link :if={is_nil Pow.Plug.current_user(@conn)} navigate={~p"/
  registration/new">Registration</.link>
3 <.link :if={is_nil Pow.Plug.current_user(@conn)} navigate={~p"/session/
  new">Sign In</.link>
```

The current user can also be fetched by using the template assigns set in the configuration with `:current_user_assigns_key` (defaults to `@current_user`).

Plugs

Pow.Plug.Session

Enables session-based authorization. The user struct will be collected from a cache store through a GenServer using a unique token generated for the session. The token will be reset every time the authorization level changes (handled by `Pow.Plug`) or after a certain interval (default 15 minutes).

The user struct fetched can be out of sync with the database if the row in the database is updated by actions outside Pow. In this case, it's recommended to add a plug that reloads the user struct and reassigns it to the connection.

Custom metadata can be set for the session by assigning a private `:pow_session_metadata` key in the conn. Read the `Pow.Plug.Session` module docs for more details.

Cache store By default `Pow.Store.Backend.EtsCache` is started automatically and can be used in development and test environment.

For a production environment, you should use a distributed, persistent cache store. Pow makes this easy with `Pow.Store.Backend.MnesiaCache`. To start MnesiaCache in your Phoenix app, add it to your `application.ex` supervisor:

```
1 defmodule MyApp.Application do
2   use Application
3
4   def start(_type, _args) do
5     children = [
6       MyApp.Repo,
7       MyAppWeb.Endpoint,
8       Pow.Store.Backend.MnesiaCache
9       # # Or in a distributed system:
10      # {Pow.Store.Backend.MnesiaCache, extra_db_nodes: {Node, :list,
11        []}},
12      # Pow.Store.Backend.MnesiaCache.Unsplit # Recover from netsplit
13    ]
14
15    opts = [strategy: :one_for_one, name: MyAppWeb.Supervisor]
16    Supervisor.start_link(children, opts)
17  end
18
19  # ...
20 end
```

Update `config/config.ex` with `:cache_store_backend` key:

```
1 config :my_app, :pow,
2   # ...
3   cache_store_backend: Pow.Store.Backend.MnesiaCache
```

Remember to add `:mnesia` to your `:extra_applications` so it'll be available for your release build. Mnesia will write files to the current working directory. The path can be changed with `config :mnesia, dir: '/path/to/dir'`.

The MnesiaCache requires write access. If you've got a read-only file system you should take a look at the Redis cache backend store guide.

Pow.Plug.RequireAuthenticated

Will halt connection if no current user is not present in assigns. Expects an `:error_handler` option.

Pow.Plug.RequireNotAuthenticated

Will halt connection if a current user is present in assigns. Expects an `:error_handler` option.

Pow security practices

See security practices.

Other libraries

PowAssent - Multi-provider support for Pow with strategies for Twitter, Github, Google, Facebook and more

Contributing

Please read CONTRIBUTING.md.

LICENSE

(The MIT License)

Copyright (c) 2018-2019 Dan Schultzer & the Contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.