Ecto is not an ORM

Ecto is a domain specific language for writing queries and interacting with databases in Elixir

4 Main Ecto Components

- Repository (Repo)
- Schema
- Changeset
- Query

Generate an OTP application including a supervision tree

```
mix new blog –sup
```

Add Ecto and Postgrex adapter dependencies

mix.exs

```
defp deps do
     [{:ecto, "~> 2.1.0-rc.1"},
     {:postgrex, "~> 0.12.1"}]
end
```

```
mix deps.get
```

Add Ecto and Postgrex to our applications list

mix.exs

```
def application do
     [applications: [:logger, :ecto, :postgrex],
     mod: {Blog, []}]
end
```

Repo

A repository maps to an underlying data store, controlled by the adapter.

Setup repository configuration

```
mix ecto.gen.repo -r Blog.Repo
```

config\config.exs

```
config :blog, Blog.Repo, adapter:
     Ecto.Adapters.Postgres,
     database: "blog_repo",
     username: "postgres",
     password: "postgres",
     hostname: "localhost"
```

Ecto 2.0 requires an **:ecto_repos** configuration for running **ecto.migrate** and others tasks

config\config.exs

```
config :blog, ecto_repos: [Blog.Repo]
```

Repo module definition

lib\blog\repo.ex

```
defmodule Blog.Repo do
     use Ecto.Repo, otp_app: :blog
end
```

otp_app: :blog *tells Ecto which Elixir application it can look for database configuration in config\config.exs*

Setup the **Blog.Repo** as a worker within the application's supervision tree

lib\blog.ex

```
# Define workers and child supervisors to be supervised
children = [
    worker(Blog.Repo, []),
]
```

This will start the Ecto process which receives and executes our application's queries

Creating database

```
mix ecto.create
```

The database for Blog.Repo has been created

Generating migrations to modify your DB schema in a DB independent way

```
mix ecto.gen.migration create_posts
```

priv\repo\migrations\20161001033650_create_posts.exs

```
create table(:posts) do
      add :title, :string
      add :body, :text
      add :pinned, :Boolean
      add :user_id, references(:users)

      timestamps  #inserted_at, updated_at
end
```

Run migration and create the posts database

```
mix ecto.migrate
```

*You can run **mix ecto.rollback** to undo the changes made by the migration*

Schema

An Ecto schema is used to map data coming from a repository, usually a table, into **Elixir structs**.

Creating the schema

lib\blog\post.ex

```
schema "posts" do
    field :title, :string
    field :body, :string
    field :pinned, :boolean, default: false
    belongs_to :user, Blog.User
    has_many :comments, Blog.Comment
    timestamps
end
```

Schema: primary & foreign keys

By default, a schema will generate a primary key, named **id** and of type **:integer**, and **belongs_to** associations in the schema will generate foreign keys of type **:integer**.

Inserting data into Repo

priv\repo\seeds.exs

```
user_alex = Repo.insert! %User{
     name: "Alex",
     reputation: 13 }

alex_post = Repo.insert! Ecto.build_assoc(alex,
:posts, title: "Hello", body: "World", pinned: true)

{:ok, _} = Repo.insert %Comment{body: "First!",
user_id: alex.id, post_id: alex_post.id}
```

Changeset

Changesets allow filtering, casting, validation and definition of constraints when manipulating structs.

Validating **Post**

lib\blog\post.ex

```
def changeset(post, params \\ %{}) do
    post
    |> cast(params, [:title, :body, :pinned, :user_id])
    |> validate_required([:title, :body, :user_id])
end
```

Valid changeset

```
valid_changeset = Post.changeset(%Post{}, %{
     title: "Correct",
     body: "Post",
     user_id: user.id
})

{:ok, _} = Repo.insert(valid_changeset)
```

Invalid changeset

```
invalid_changeset = Post.changeset(%Post{}, %{
     title: "Incorrect",
     body: 123
})

false = invalid_changeset.valid?

{:error, err_changeset} =
     Repo.insert(invalid_changeset)
```

Invalid changeset

```
#Ecto.Changeset<action: :insert, changes: %{title:
"Incorrect"}, errors: [user_id: {"can't be blank",
[]}, body: {"is invalid", [type: :string]}], data:
#Blog.Post<>, valid?: false>
```

Query

Written in Elixir syntax, queries are used to retrieve information from a given repository.

Fetching a single record

```
user_query = Ecto.Query.first(User)

user = Repo.one(user_query)
```

User struct

```
%Blog.User{__meta__: #Ecto.Schema.Metadata<:loaded,
"users">, id: 1, inserted_at: ~N[2016-10-01
08:09:06.201000], name: "Alex", posts:
#Ecto.Association.NotLoaded<association :posts is
not loaded>, reputation: 13, updated_at: ~N[2016-10-
01 08:09:06.207000]}
```

Fetching a single record

```
another_user_query =
      from u in User,
      order_by: [asc: u.id],
      limit: 1
```

```
#Ecto.Query<from u in Blog.User, order_by: [asc: u.id], limit: 1>
#Ecto.Query<from u in Blog.User, order_by: [asc: u.id], limit: 1>
```

Fetching all records

```
all_users = Repo.all(User)
```

Fetch a single record based on ID

```
singe_user = Repo.get(User, user.id)
```

Fetch a single record based on a specific attribute

```
alex = Repo.get_by(User, name: "Alex")
```

Complex query

```
low_rep =
     from u in User,
     where: u.reputation < 30,
     preload: [:posts],
     select: u

alex_with_posts = Repo.all(low_rep)
```

Composing Ecto queries

```
alex_post_q =
     from p in Post,
     where: p.user_id == ^alex.id

pinned_post_q =
     from ap in alex_post_q,
     where: ap.pinned == true

pinned_post = Repo.all(pinned_post_q)
```

Updating records

```
alex_changeset = Ecto.Changeset.change(alex, reputation: 144)
```

```
{:ok, new_alex} = Repo.update(alex_changeset)
```

```
%Blog.User{__meta__: #Ecto.Schema.Metadata<:loaded, "users">, id: 1,
 inserted_at: ~N[2016-10-01 08:09:06.201000], name: "Alex",
 posts: #Ecto.Association.NotLoaded<association :posts is not loaded>,
 reputation: 144, updated_at: ~N[2016-10-01 08:24:27.720000]}
```

Deleting records

```
molly = Repo.get_by(User, name: "Molly")

{:ok, _deleted_rec} = Repo.delete(molly)
```

Resources

https://hexdocs.pm/ecto/Ecto.html

https://github.com/yuriibodarev/Ecto_not_ORM