# Northwind Elixir Traders

Learn Elixir and database modeling with Ecto and SQLite, all in one project

Isaak Tsalicoglou

OVERBRING LABS

# Northwind Elixir Traders

Learn Elixir and database modeling with Ecto and SQLite, all in one project

Isaak Tsalicoglou

# Chapter 1: Our objective, the application and the repo

Hello there, welcome to this “Elixir and Ecto gym” in book form! Together, over almost 500 pages we will explore how to achieve our main objective:

> **We will know we are done, when**: we will have fully modeled the classic (and ancient) Northwind Traders database with Ecto, implemented improvements, and brought our Elixir app to a sufficiently enhanced (though not perfect) working state, in which we will be able to explore the toy dataset of the original database with Ecto queries to generate insights.

Every endeavor nowadays is being described as a journey, and we shall not refrain from using this tired cliché either. Every journey begins with a first step, which I hope you have already taken some time ago: your zero-th step should have been installing Elixir on your operating system of choice, and learning how to run IEx in a terminal window.

However, *this* journey right here begins with creating our “Northwind Elixir Traders” application, adding Ecto as a single direct dependency, and setting up the Ecto repository, so that we can start modeling the database. Our Elixir application will be composed by a set of modules, across which we will be splitting our code, so that we can make it easier to find everything we are looking for. Nothing is stopping you from putting all modules within a single `.ex` file—except common sense.

Let’s get started! We will use `mix new PATH`, where `PATH` is the name of the base directory of the application. `PATH` must contain lowercase ASCII letters, numbers, or underscores. It will be converted by Mix into the module name by capitalizing the first letter of every sequence of characters before, between and after any underscores, and by removing the underscores in the end. This means that if we want our application to be called `NorthwindElixirTraders`, then `PATH` must be defined as `northwind_elixir_traders`.

We will also provide the `--sup` option, so that the application includes a supervision tree. This will be necessary later on, so that the Ecto repository process can interact with the database in a fault-tolerant manner.

```
$ mix new northwind_elixir_traders --sup
* creating README.md
* creating .formatter.exs
…
* creating test/northwind_elixir_traders_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd northwind_elixir_traders
    mix test

Run "mix help" for more commands.
```

From now on all shell commands we’ll be running will use the `northwind_elixir_traders` directory as the base path, so go ahead and `cd` into it. Also, the ellipsis symbol “…” will indicate omitted lines that are not the focus of the code block. Wherever empty lines are not part of syntax, they will not be shown (e.g. in the output of console commands).

## Adding dependencies

The Mix task to create the application generated the bare minimum code that's required, so it should come as no surprise that our new application also has a bare-bones `mix.exs` file. There, among other information of our Mix project, we will be adding dependencies that enable us to create and manage the database of a fictional business called "Northwind Elixir Traders".

In Elixir projects we use Ecto to:

1. define data structures (table schemas) and where they are stored (a database; in Ecto terms, a "repository"),
2. process data structures and generate changes to their contained data, and
3. define and run either SQL-like or functionally-composable queries.

It is also possible to use Ecto *without* a database; for example, to validate data structures. However, this project is all about modeling and managing a database. Therefore, we will need to use a "database adapter" sooner, rather than later. You can think of this database adapter as the interface between Ecto and a specific type of database, such as PostgreSQL or SQLite.

To make things more conducive towards exploratory learning we will start with SQLite, and we will therefore need to use the `ecto_sqlite3` adapter. At the time of this writing, the package's page on hex.pm indicates that 0.18.1 is the most recent version. This is the only dependency that we need to include right now, as Mix will automatically fetch all dependencies of `ecto_sqlite3`. These will most importantly include:

- Ecto itself,
- Ecto SQL, to be able to work with SQL in Ecto, and
- Exqlite, an Elixir library for interfacing with SQLite version 3.x.

The hex.pm page also features a "Config" side-pane that indicates how we should include this dependency in `mix.exs`, so let's go ahead and do that by adding the corresponding line in the list that the `deps/0` private function returns. Also, let's remove the indicative example placeholders.

**Figure 4.1. Adding Ecto SQLite3 as a dependency in mix.exs**

```elixir
defmodule NorthwindElixirTraders.MixProject do
…
  defp deps do
    [{:ecto_sqlite3, "~> 0.18.1"}]
  end
…
end
```

Before we proceed, we need to fetch the dependencies we implicitly added by including `ecto_sqlite3` in `mix.exs`. With the Mix command `deps.get` you will see that *far* more than just `ecto_sqlite3` or the three other dependencies mentioned above will indeed be fetched, as the dependencies themselves depend on other Hex packages:

```
$ mix deps.get
Resolving Hex dependencies...
Resolution completed in 0.029s
Unchanged:
  …
* Getting ecto_sqlite3 (Hex package)
* Getting ecto (Hex package)
* Getting ecto_sql (Hex package)
* Getting exqlite (Hex package)
…
* Getting telemetry (Hex package)
```

If you now examine the generated `mix.lock` file, you'll find a list of packages that were pulled, alongside their versions. We can also see dependencies with names that include "ecto" or "qlite", without having to wade through `mix.lock`:

```
$ mix deps | grep -E "ecto|qlite" | grep package
* ecto 3.12.5 (Hex package) (mix)
* ecto_sql 3.12.1 (Hex package) (mix)
* ecto_sqlite3 0.18.1 (Hex package) (mix)
* exqlite 0.29.0 (Hex package) (mix)
```

## Creating the repository (the "repo")

We will use the Mix task `ecto.gen.repo` to generate the repository. Note the name we use with the `-r` parameter: the repo will be a module of our `NorthwindElixirTraders` application.

```
$ mix ecto.gen.repo -r NorthwindElixirTraders.Repo
* creating lib/northwind_elixir_traders
* creating lib/northwind_elixir_traders/repo.ex
* creating config/config.exs
Don't forget to add your new repo to your supervision tree
(typically in lib/northwind_elixir_traders/application.ex):

    def start(_type, _args) do
      children = [
        NorthwindElixirTraders.Repo,
      ]

And to add it to the list of Ecto repositories in your
configuration files (so Ecto tasks work as expected):

    config :northwind_elixir_traders,
      ecto_repos: [NorthwindElixirTraders.Repo]
```

Make the modifications as instructed before proceeding. To be more specific, you need to:

1. Do as instructed in `lib/northwind_elixir_traders/application.ex`.
2. Add the final two code lines in `config/config.exs`.

Now, there's a catch with the Mix `ecto.gen.repo` task we used: if you look into the generated `lib/northwind_elixir_traders/repo.ex` file you will see that the adapter is set to PostgreSQL by default (`adapter: Ecto.Adapters.Postgres`). Change that line accordingly, to be able to use SQLite instead of PostgreSQL:

**Figure 4.2. repo.ex with the Ecto SQLite3 database adapter**

```elixir
defmodule NorthwindElixirTraders.Repo do
  use Ecto.Repo,
    otp_app: :northwind_elixir_traders,
    adapter: Ecto.Adapters.SQLite3
end
```

There's the same catch for the generated `config/config.exs` file, which contains connection and authentication settings that are not applicable to an SQLite database, since the database is a file on the disk, instead of a server application. We do not need to define `username`, `password`, or `hostname`. Modify `config.exs` in two ways:

1. By adding the second snippet as suggested by the `mix ecto.gen.repo` command above the existing block that starts with `config :northwind_elixir_traders, NorthwindElixirTraders.Repo,`.
2. By modifying the existing block that starts with "`config :northwind_elixir_traders, NorthwindElixirTraders.Repo,`" to remove the non-applicable parameters, keeping only the line beginning with "`database:`" and adding the filename of the SQLite database; this will be relative to your application's base path.

You should end up with the following, which includes the two lines you added earlier as instructed by the Mix task, and the final two lines that result from switching to PostgreSQL parameters to the single parameter we need for an SQLite database:

**Figure 4.3. config.exs configured for an SQLite database**

```elixir
import Config

config :northwind_elixir_traders,
  ecto_repos: [NorthwindElixirTraders.Repo]

config :northwind_elixir_traders, NorthwindElixirTraders.Repo,
  database: "northwind_elixir_traders_repo.db"
```

With these modifications:

1. We define the list of Ecto repositories to be used; in this case, we will only use a single repository/database ("repo"), but in other cases you might have more than one, each with different settings and even a different Ecto database adapter. In fact, later on in the book we will be using a second repo, from which we'll import data.
2. We configure the single repo our app will use with the filename that will contain the SQLite database.

Next, we use Mix and the `ecto.create` task to create the database:

```
$ mix ecto.create
Compiling 1 file (.ex)
The database for NorthwindElixirTraders.Repo has been created
```

In case you are using git, add this line to `.gitignore`, so that the database files are ignored: `*.db*`

You should now see a `northwind_elixir_traders_repo.db` file in the directory. It is entirely empty, as we have not yet defined any tables in our database.

## The need for a supervision tree

When we created our new application earlier, we provided the `--sup` option to the Mix task, thus requesting the creation of minimal, boilerplate code for the creation of a supervision tree. Later, we added our repo as a child process of that tree. But what is a *supervision tree*, really?

A supervision tree is a way for structuring an application so that it can become fault-tolerant, i.e. so that it can continue operating uninterrupted, even when some of its components fail. The Erlang documentation on supervision principles provides a nice visual explanation.

When we provided the `--sup` option, code for a new supervision tree was generated by creating a *supervisor*, i.e. a process that monitors the behaviors of other processes: either other supervisors or *workers*, processes that get things done (other than supervision). We can examine the effects of this option by looking closer at the contents of the `lib/northwind_elixir_traders/application.ex` file.

Though this is an excellent way to get "nerd-sniped" and start diving deeper and deeper into a topic, it's also an excellent way to understand better what exactly is going on with these seemingly "magic" invocations. This book is built upon this premise. We will go as far as necessary to understand what's happening, without going so far as to lose sight of our end-goal.

There, we will find the following:

```
use Application
```

This line means that the `NorthwindElixirTraders.Application` module will include the functions, macros and *behaviors* of Elixir's `Application` behavior module. This module is used to implement the life-cycle of an Elixir application; from the application getting started, to its graceful or ungraceful stopping. In other words, the `Application` module makes it possible to execute some actions on startup, before, and after the termination of our application.

To fulfill this requirement (i.e., *life-cycle management*), an application must implement specific behaviors: functions and *callbacks* that are defined in the `Application` module. With `use Application` we "announce" that we will be doing things (life-cycle management, in this case) according to the way (the behaviors) that the `Application` module has implemented. You can think of this as a "contract" that establishes how to do certain things in accordance to the behaviors expected by the `Application` module.

In this app we will be using Ecto's functionality for interacting with a database. This relies on:

- functions and macros provided by the `Ecto.Repo` module of Ecto, and
- a *repository process* that will communicate with our database of choice.

Therefore:

- in the file `lib/northwind_elixir_traders/repo.ex` you will find a `use Ecto.Repo` line, and
- further down in `application.ex` you fill find code that starts the repository process.

Indeed, `application.ex` continues with the following lines:

```
@impl true
def start(_type, _args) do
```

The first line, `@impl true`, is an *attribute* providing optional metadata to the Elixir compiler and optional documentation to a human reader of the code. `@impl true` declares that the `start/2` function that follows is implementing a behavior that is required by another module. It makes the Elixir compiler check that the function indeed complies to what's expected by the `Application` module. This way, if our `NorthwindElixirTraders.Application` module's functions don't conform to the behavior "contract" of Elixir's `Application` module, we will see errors at compile time.

The function definition that follows implements one of the callback functions that Elixir's `Application` module expects. Specifically, it fulfills the "contractual obligation" of defining a `start/2` function that will be called when the application is started (hence the "callback" name).

There could be *any* code we wanted in this function. However, since we used the `--sup` option when we created the application, we requested that Mix defines a supervision tree. For such a tree to exist, it must contain a supervisor process. That's why if you look at the content of the `start/2` callback function you'll see that it uses the `start_link/2` function of Elixir's `Supervisor` module to create a supervisor and, therefore, the root of the supervision tree we requested.

The arguments passed to the `start_link/2` function here define:

1. the name of the supervisor (the `:name` option),
2. the children processes of this supervisor (the `children` list), and
3. the *supervision strategy* (the `:strategy` option).

The name of the supervisor seems pretty self-explanatory: `NorthwindElixirTraders.Supervisor`.

An exploration of the kinds of names we could provide here goes beyond the scope of this book. If you want to find out more, look into the "Name registration" section of Elixir's `GenServer` module.

The Mix task instructed us to include `NorthwindElixirTraders.Repo` as a child process of the supervision tree by including it in the `children` list. `NorthwindElixirTraders.Repo` is a *worker process*, i.e. a process that will not supervise other processes, but will "get things done" under the supervision of `NorthwindElixirTraders.Supervisor`. "Getting things done" in this case means managing database connections and database operations.

Finally, we come to the supervision strategy, i.e. how the supervisor should handle crashes of its child processes. The Mix task defined it as `:one_for_one`. The effect of this strategy is the following: if the `NorthwindElixirTraders.Repo` child process crashes, then `NorthwindElixirTraders.Supervisor` will restart only that specific process. This is what provides the fault tolerance that we mentioned earlier; a crash of the repository process should not impact other parts of our application. The process would be restarted by its supervisor while other child processes (that don't depend on the crashed-and-restarted `Repo` process) would continue working. If you've heard of the motto "let it crash" mentioned in relation to Elixir or Erlang, this is what is being referred to. Let it crash; the supervisor will deal with it according to the strategy we defined.

The Mix task to create the application defined a bare-bones, minimal `application.ex` file with a minimal `start_link/3` callback that we amended. Given that we `use Application`, in the future we might want to define other parts of the application's life-cycle too, such as any cleanup tasks that should happen after the application is stopped, or even what should happen *before* the application is stopped. For example, we could use the `prep_stop/1` callback:

- to send messages to other processes that the application is about to be stopped, so that they can execute their own cleanup functions preparing for the impending termination of the application,
- to save the application state so that operations can resume from the same state as before, when the application is eventually restarted,

- to clean up any temporary files that were created while the application was running, or
- to send information about the application's impending unavailability to logging and monitoring systems.

We could use the `stop/1` callback for similar cleanup tasks, but these will be executed *after* the supervision tree of the application has been stopped.

For example, in one of the Phoenix LiveView I've been developing I'm using Req calls within the various callback functions to make `POST` requests to a self-hosted instance of ntfy.sh, so that my phone receives notifications about the state of the application's Ecto repo process.

## Summary and outlook

In this chapter we used Elixir's Mix to set up our "Northwind Elixir Traders" application and its dependencies, and defined the repository to use SQLite and an SQLite database file on the disk, instead of a database on a PostgreSQL server. We also learned the basics of what a supervision tree is, and how it will be used to provide fault tolerance to our use of `Ecto.Repo` in our applications repo module.

In the next chapter we will take a look at the original Northwind Traders database. We will take our first steps in defining one of its simplest tables and running simple commands on its records using Ecto, while taking into account Ecto's conventions.