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 10: Importing data from a dynamic repository

In the previous chapter we learned the basics of creating and modifying associations using `put_assoc/4` and `cast_-assoc/3`. This can be a confusing topic, even when using fictional examples. If we want to learn more easily through practice, we would benefit from an extensive dataset to play with. The toy dataset of the original Northwind Traders database can fulfill that role, but we must import the data into whatever tables we have modeled thus far.

Seeding the database

To make sure that your database and mine are at the same state, let's seed the database with a fresh set of demo data. First, exit IEx and rebuild the database:

```
mix ecto.drop && mix ecto.create && mix ecto.migrate && iex -S mix
```

Then, copy and paste the following code line-by-line into your IEx session to "seed" the database with a clean set of data for demonstration purposes in the first half of this chapter, before we proceed with actual data from the Northwind Traders database:

```
alias NorthwindElixirTraders.{Repo, Category, Product, Supplier}
[[1, "Post-Apocalyptic Provisions"], [2, "Arcane Artifacts"]] |> Enum.map(&(Enum.zip([:id, :name], &1) |> \
Map.new)) |> Enum.map(&Category.changeset(%Category{}, &1) |> Repo.insert)
[[1, "Radroach & Sons"], [2, "TranStar Disposals & Acquisitions"]] |> Enum.map(&(Enum.zip([:id, :name], &1\
) |> Map.new)) |> Enum.map(&Supplier.changeset(%Supplier{}, &1) |> Repo.insert)
[[1, "BlamCo Not-Cheese Spread", "1 tin - 250 g", 4.99, 1, 1], [2, "Elixir of Maybe-Healing", "1 bottle - \
330 mL", 19.99, 2, 1], [3, "Wabbajug (™ pending)", "1 piece", 2999.99, 2, 2], [4, "Mimic Repellent Spray",
 "1 can - 50 g", 1337, 1, 2], [5, "Junk Jet Fuel", "1 bottle", 45, 1, 1]] |> Enum.map(&(Enum.zip([:id, :na
me, :unit, :price, :category_id, :supplier_id], &1) |> Map.new)) |> Enum.map(&Product.changeset(%Product{}
, &1) |> Repo.insert)
```

Alternatively, copy and paste the lines of code from this URL: https://pastebin.com/raw/U0H36ZaX

Fetching the original Northwind Traders data

Wikiversity provides a text file (also archived here) that can be used to recreate the original database, including the records of its various tables. There are a few ways we could turn this text file into a source of data for our own database. Here they are, in order of decreasing hassle.

Option 1: convert the original file to our conventions

We could use `sed` on the terminal to search and replace the column names in the file so that they reflect the naming scheme we used; for example, `id` instead of `CategoryID`, `name` instead of `SupplierName`, and so on. We would also need to do the same with all foreign key names. Note that we wouldn't have to do this to address the names of all tables beginning with a capital letter, since SQLite is case-insensitive with regards to table and column names. However, we would have to address the difference in naming between the CamelCase of "**OrderDetails**" and the "snake case" of our future "`order_details`" table (which we haven't yet dealt with). Then, we would have two options:

- **Option 1a**: After that, we could write some Elixir code that parses all the `INSERT INTO` SQL statements of the text file and converts them into `INSERT INTO` SQL statements that we can execute using `Repo.query/3`.
- **Option 1b**: Alternatively, we could write some Elixir code that parses all the `INSERT INTO` SQL statements of the text file and converts them into maps, which we can then persist using `Repo.insert/2`.

In either case, we would need to determine the order of columns in the original `CREATE TABLE` statement of each table, either by hard-coding or parsing the column order. With this information, we would then be able to correctly parse and interpret the order of values from the original `INSERT INTO` statements. Next, we would need to convert these values to match the order of columns in our Elixir/Ecto implementation of the database, or to construct a map that properly associates the values from the original `INSERT INTO` statements with the correct fields in the schemas we defined for `Product`, `Category`, `Supplier`, and so on.

As for executing SQL from within Ecto, here is an example of how this works:

```
iex> {:ok, r} = Repo.query("SELECT * FROM Products;")
{:ok,
 %Exqlite.Result{
   command: :execute,
   columns: ["id", "name", "unit", "price", "inserted_at", "updated_at",
    "category_id", "supplier_id"],
   rows: [
     [1, "BlamCo Not-Cheese Spread", "1 tin - 250 g", 4.99,
      "2025-02-23T16:49:49Z", "2025-02-23T16:49:49Z", 1, 1],
     …
     [5, "Junk Jet Fuel", "1 bottle", 45, "2025-02-23T16:49:49Z",
      "2025-02-23T16:49:49Z", 1, 1]
   ],
   num_rows: 5
 }}
```

Note how the struct of the result `r` of the query returned by Exqlite contains a `:columns` field with a list of the columns in the order in which the values are reported for each row in `:rows`. Therefore, if you can do this:

```
iex> Enum.zip(r.columns, hd(r.rows))
[
  {"id", 1},
  {"name", "BlamCo Not-Cheese Spread"},
  {"unit", "1 tin - 250 g"},
  {"price", 4.99},
  {"inserted_at", "2025-02-23T16:49:49Z"},
  {"updated_at", "2025-02-23T16:49:49Z"},
  {"category_id", 1},
  {"supplier_id", 1}
]
```

…then you can also do the following, to convert the results of the query into a list of maps:

```
iex> Enum.map(r.rows, &(Map.new(Enum.zip(Enum.map(r.columns, fn x -> String.to_atom(x) end), &1))))
[
  %{
    id: 1, name: "BlamCo Not-Cheese Spread", unit: "1 tin - 250 g", category_id: 1, price: 4.99,
    supplier_id: 1, inserted_at: "2025-02-23T16:49:49Z", updated_at: "2025-02-23T16:49:49Z"
  }, …
]
```

Does this one-liner work? Yes. Will you scratch your head when you return to such code in the future? Also yes. Should you write such code in `.ex` files? It depends on how much you hate your future self!

Note that the `r` struct contains no information about the types of the columns; thus, the values of any datetime fields, such as `:updated_at` and `:inserted_at`, will not be automatically converted to a `DateTime` struct! Though the original database doesn't contain such timestamps, the **Orders** and **Employees** tables include the `OrderDate` and `BirthDate` fields, respectively, that we would need to deal with.

Option 2: do as the web page instructs, and then use a second Ecto repo

Alternatively, we can follow the instructions within the SQL file shown on the web page—and that's in fact what we will do. By executing all SQL statements, we will recreate the Northwind Traders database in a new SQLite `.db` file. This will result in the `NorthwindTraders-original.db` database file that you've seen me refer to a few times thus far.

First, you could copy and paste the SQL text from the Wikiversity page (also archived here) into a text file, e.g. named `nt.sql`. Then you can feed it into `sqlite3` to execute all statements and create the database:

```
$ sqlite3 -batch -echo NorthwindTraders-original.db < nt.sql
```

You should now have the original Northwind Traders database as the `NorthwindTraders-original.db` SQLite file.

So far we've dealt with our own database, which is managed by Ecto using `Repo` and the `northwind_elixir_-traders_repo.db` database file, the name of which we defined in `config.exs` back in **Chapter 1**. We can write an Elixir module that selectively imports some of the data from the original database file into our Repo. "Selectively" and "some", because we have not yet implemented all tables.

Using a dynamic Ecto repo

Our Elixir application has been configured to use what we have so far called `Repo`, defined in `repo.ex`, backed by the SQLite database file defined in `config.exs`. How can we open a second database to pull data from it?

The answer is: dynamic repositories, and the `put_dynamic_repo/1` function in particular. Remember that when we configured our repository back in **Chapter 1**, we also added it to the supervision tree of our application with the one-for-one strategy (`strategy: :one_for_one` in `application.ex`). This means that `NorthwindElixir-Traders.Repo` is a process that accesses the database file using the Ecto SQLite3 database adapter, is a child process of the `NorthwindElixirTraders` application. `Repo` will be restarted if it crashes, without affecting other child processes of our application.

Since what we alias as `Repo` is a process, we can find out its PID (process identifier) within the BEAM VM by using Elixir's `Process` module. Note that the PID value doesn't need to be the same on your machine:

```
iex> Process.whereis(Repo)
#PID<0.298.0>
```

What we want to do is to start *another* process, so that we can connect to the original Northwind Traders database file. We achieve this by using the `start_link/1` function of `Repo` and providing the necessary options (`opts`) that configure the connection to the database *dynamically*, i.e. *not* by relying on configuration values in `config.exs`.

Following the documentation on dynamic repositories, we need to start another repository process by providing a `:name`. Since we are not using PostgreSQL, we again ignore the `:hostname`, `:username` and `:password` options and only define the `:database` option along side a name. Note that `Repo.start_link/1` returns one of the following tuples:

- `{:ok, pid()}` in case the call was successful, with the PID of the new repository process as its `elem(1)`.
- `{:error, {:already_started, pid()}}` in case the repository with that name has already been started. By default, "that name" is what's already defined in `config.exs`.
- `{:error, term()}` if anything else went wrong.

For example, we can find out the PID of the repo that connects to our Northwind Elixir Traders SQLite database like so:

```
iex> Repo.start_link
{:error, {:already_started, #PID<0.298.0>}}
```

We can also get its configuration using the `config/0` function. This reflects the settings we have provided in `config.exs`:

```
iex> Repo.config
[
  telemetry_prefix: [:northwind_elixir_traders, :repo],
  otp_app: :northwind_elixir_traders, timeout: 15000, pool_size: 10,
  database: "northwind_elixir_traders_repo.db"
]
```

Thus, we can also start a new repository, which we will be calling `:nt`:

```
iex> {:ok, nt_pid} = Repo.start_link(name: :nt, database: "NorthwindTraders-original.db")
{:ok, #PID<0.449.0>}
```

In case you didn't pattern-match on the tuple to have the PID in `nt_pid`, you can get the PID value as before, by looking for the process named `:nt`:

```
iex> Process.whereis(:nt)
#PID<0.449.0>
```

We can also see which repositories are running:

```
iex> Ecto.Repo.all_running
[:nt, NorthwindElixirTraders.Repo]
```

Now that the new repo has been dynamically started, we need to switch to it by using the `put_dynamic_repo/1` function. We can also verify that the switch has happened by using `get_dynamic_repo/0`:

```
iex> Repo.put_dynamic_repo(:nt)
iex> Repo.get_dynamic_repo
:nt
```

Now, any interaction with Repo hits the `NorthwindTraders-original.db` database. Thus, it's no surprise that the following function call that uses the `NorthwindElixirTraders.Product` schema will fail, since the schema doesn't match the schema of the **Products** table in the original database:

```
iex> Repo.all(Product)
** (Exqlite.Error) no such column: p0.id
SELECT p0."id", p0."name", p0."unit", p0."price", p0."category_id", p0."supplier_id", p0."inserted_at", p0\
."updated_at" FROM "products" AS p0
    (ecto_sql 3.12.1) lib/ecto/adapters/sql.ex:1096: Ecto.Adapters.SQL.raise_sql_call_error/1
    (ecto_sql 3.12.1) lib/ecto/adapters/sql.ex:994: Ecto.Adapters.SQL.execute/6
    (ecto 3.12.5) lib/ecto/repo/queryable.ex:232: Ecto.Repo.Queryable.execute/4
    (ecto 3.12.5) lib/ecto/repo/queryable.ex:19: Ecto.Repo.Queryable.all/3
```

Meanwhile, we can pull data using SQL queries just fine, like we did before; only, this time, the queries hit the original Northwind Traders database, since the active repo is not `NorthwindElixirTraders.Repo`, but `:nt`:

```
iex> Repo.query("SELECT * FROM Categories;")

05:59:00.593 [debug] QUERY OK db=0.1ms idle=1382.4ms
SELECT COUNT(*) FROM Categories; []
{:ok, %Exqlite.Result{command: :execute, columns: ["COUNT(*)"], rows: [[8]], num_rows: 1}}
```

Implementing a data importer module

To import the data from the original Northwind Traders database into Northwind Elixir Traders, we need to map the data from those SQL `SELECT` queries onto the existing schemas. Thus, we can implement a module that performs the following tasks that match the one to the other:

- Primary keys named "SomethingID" turn into the `:id` primary key of the `Somethings` table.
- Columns named in CamelCase are converted to "snake case", i.e. `FirstName` turns into `first_name`.
- Foreign-key names are only converted to snake case.
- By our own convention, a column of the "Somethings" table that's named "SomethingName" becomes the field `:name`.
- Any column with a name that contains "Date" represents either a date or a datetime field, depending on the original Northwind Traders schemas.

This module should also include functions that execute a `SELECT *` on a table of Northwind Traders and process the query results into maps that can then be cast using a changeset function, since this is data coming from outside our application.

Let's start by creating a new file named `data_importer.ex` under `lib/northwind_elixir_traders/`. In the new `NorthwindElixirTraders.DataImporter` module we will be adding all the helper functions that will perform these tasks. We can start by adding:

1. a `@name` module attribute for the internal name we'll give to the dynamic repo, and one (`@database`) indicating the SQLite file for this dynamic repo,

2. a function that wraps `Repo.start_link/1`, but with a name that's easier to remember, and
3. a "preparatory/cleanup" function that switches the repository from anything else to the `:nt` repository before any SQL is executed, and then back to the previous repository (here, `NorthwindElixirTraders.Repo`) after a function is run.

We'll also require the `Logger` module, so that we get access to its macros for logging the switch to the `:nt` dynamic repository and back to our `Repo`. We can also make this module detect whether the dynamic repository process has already been started, and start if it, if it hasn't.

Figure 13.1. The DataImporter module's data_importer.ex with switch/0 and switch/1 to switch between Repo and :nt

```elixir
defmodule NorthwindElixirTraders.DataImporter do
  require Logger
  alias NorthwindElixirTraders.Repo

  @name :nt
  @database "NorthwindTraders-original.db"

  def start() do
    if is_nil(Process.whereis(@name)),
      do: Repo.start_link(name: @name, database: @database)
  end

  def switch(name) when name in [@name, Repo] do
    try do
      if name == @name, do: start()
      Repo.put_dynamic_repo(name)
      Logger.debug("Switched to #{name}")
      {:ok, Repo.get_dynamic_repo()}
    catch
      _ ->
        Logger.debug("Error: could not switch to #{name}")
        {:error, Repo.get_dynamic_repo()}
    end
  end

  def switch() do
    case Repo.get_dynamic_repo() do
      @name -> switch(Repo)
      _ -> switch(@name)
    end
  end
end
```

This way, we'll be calling `switch(:nt)` at the beginning of every function that gets data from the Northwind Traders database, and call `switch()` again before its return, to reactivate the `Repo`, which handles the Northwind Elixir Traders database. Thus, functions that pull data from the Northwind Traders database briefly "pop out" of our standard interaction mode with `Repo` (the one configured in `config.exs`) and back to it after our queries are done. We can wrap `Repo.query/1` within our own `nt_query/1` function that takes the SQL statement as its sole argument:

A different way of going about this is to avoid switching back and forth with the `switch/0` functions above. Instead, we can use `Ecto.Adapters.SQL.query/4`, which takes the target repo name (`:nt`), the SQL query, and the query parameters as its third arguments.

Figure 13.2. Wrapper function to temporarily run queries on the original Northwind Traders database

```elixir
defmodule NorthwindElixirTraders.DataImporter do
  …
  def nt_query(sql) when is_bitstring(sql) do
    switch(:nt)
    result = Repo.query(sql)
    switch()
    result
  end
end
```

We'll need to execute SQL on tables within the original database, so it would be good to have the list of table names available. Since I don't like to hard-code data in my code, I'll add:

1. a function that converts a singular row name to plural, to map a row name to a table name,
2. a function that does the opposite, i.e. converts a plural table name to the singular name of its row,
3. a function that returns the list of table names in the Northwind Traders database.

Let's implement (1) and (2):

Figure 13.3. Helper functions that singularize and pluralize rows' and tables' names, respectively

```elixir
defmodule NorthwindElixirTraders.DataImporter do
  …
  def singularize(plural) when is_bitstring(plural) do
    ending = String.slice(plural, -3..-1)

    if ending == "ies" do
      String.replace(plural, ending, "y")
    else
      String.trim(plural, "s")
    end
  end

  def pluralize(singular) when is_bitstring(singular) do
    last_char = String.last(singular)

    case last_char do
      "y" -> String.trim(singular, last_char) <> "ies"
      _ -> singular <> "s"
    end
  end
end
```

With these two functions we can get from the table name to the corresponding row name that we use in the Northwind Elixir traders database, and vice versa. Note that we don't need to explicitly start the dynamic `:nt` repo, as it is done for us automatically, if it's not started yet.