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 13: Cleaning up

In the previous chapter we completed the modeling of the Northwind Traders ERD with Elixir and Ecto. We now have at our disposal the Northwind Elixir Traders database with all its tables, including the many-to-many association between the `orders` and `products` tables through the `order_details` join table. Throughout all the chapters so far we have also implemented numerous improvements to the database, primarily in order to import the original data reliably, and with as little ambiguity as possible, such as for the phone numbers.

By now, the SQLite database managed by `NorthwindElixirTraders.Repo` is filled with the data from the original Northwind Traders database, and with the `countries` table populated with data imported from the CSV file:

```
iex> DataImporter.count_all_both
[
  {"Categories", 8, 8}, {"Customers", 91, 91}, {"Employees", 10, 10}, {"OrderDetails", 518, 518},
  {"Orders", 196, 196}, {"Products", 77, 77}, {"Shippers", 3, 3}, {"Suppliers", 29, 29}
]
iex> Repo.aggregate(Country, :count)
249
```

We are in a great position to continue improving our codebase and to run various queries to explore the original dataset.

Discovering some loose ends of our table prioritization

In case you have experimented on this database, now would be a good time to use the mix tasks to drop the database, create it, run all migrations, and then execute `DataImporter.import_all_modeled/0`, which *should* bring the database contents to the state shown above. I write *should* and not *shall*, because it's time to address the bug I alluded to in **Chapter 11** regarding the prioritization algorithm of the tables to be imported from Northwind Traders.

So, let's first do this:

```
$ (mix ecto.drop && mix ecto.create && mix ecto.migrate) > /dev/null && iex -S
mix
```

Then, from within IEx, let's try to import everything again:

```
iex> DataImporter.import_all_modeled
iex> DataImporter.check_all_imported_ok
:warning
```

Oops—what went wrong?

```
iex> DataImporter.count_all_both |> Enum.filter(& elem(&1, 2) == 0)
iex> [{"OrderDetails", 518, 0}]
```

The data of the **OrderDetails** table failed to import correctly. Let's look at the order in which the tables were imported:

```
iex> DataImporter.prioritize |> Enum.map(fn m -> Module.split(m) |> List.last end) |> Enum.with_index
[
  {"Category", 0}, {"Employee", 1}, {"Supplier", 2}, {"Shipper", 3}, {"Customer", 4}, {"Product", 5},
  {"OrderDetail", 6}, # <-- here's the problem
  {"Order", 7}
]
```

The **OrderDetails** table was prioritized *before* the **Orders** table; however, the former depends on the latter. Why is the order wrong? We'll see soon. For now: with the **Orders** table now already imported, we should be able to import the **OrderDetails** table after the fact, since now the `%Order{}` records referred to with the `:order_id` foreign-key field values in every order detail already exist. Indeed:

```
iex> DataImporter.insert_all_from("OrderDetails")
iex> DataImporter.count_all_both |> Enum.filter(& elem(&1, 0) == "OrderDetails")
[{"OrderDetails", 518, 518}]
```

Thus, our prioritization is off, for some reason. This also means that reversing the order and using `DataImporter.teardown/0` should also not work, when the database is properly populated:

```
iex> DataImporter.reset
iex> DataImporter.teardown
…
22:22:07.103 [debug] QUERY ERROR source="orders" db=7.6ms queue=0.2ms idle=1511.0ms
DELETE FROM "orders" AS o0 []
** (Exqlite.Error) FOREIGN KEY constraint failed
```

The `teardown/0` function tried to remove all records from the `orders` table, and failed. As we saw in **Chapter 7**, in "Handling foreign key constraints in changesets" of the "Limitations and caveats" section of the Ecto SQLite3 documentation we find the following information:

> *"[...] unlike other databases, SQLite3 does not provide the precise name of the constraint violated [...]"*

We know already that the reason is that we attempt to remove `%Order{}` records that are referred to from `%OrderDetails{}` that haven't yet been removed, due to our buggy table prioritization. In the section "Tearing down the database and re-importing data" of **Chapter 10**, we found out that it was not possible to delete records that are referred to from other records. Let's review **Chapter 10**, where we find this:

> *"In general, we need an algorithm that will automatically order the list of modeled tables in order of increasing dependence for subsequent data insertion."*

Further into this chapter, we find this thought:

> *"[...] if we sort this list in ascending order of the length of the list in `elem(1)`, we automatically get the order in which we should automatically import the tables, so that the importing of the dependent tables is deferred until after the importing of the tables they depend upon has taken place."*

There, the *"length of the list in `elem(1)`"* referred to the length of the list of "outbound connections" of each module that is a list containing the foreign keys of that schema. We got this list by `Enum.mapping` the `DataImporter.outbound/1` function across all modules. Here's the problem, however: it just so happens that the length of the list of foreign keys of the schema in `Order` is longer than the equivalent list for `OrderDetail`:

```
iex> [Order, OrderDetail] |> Enum.map(&{&1, DataImporter.outbound(&1)}) |> Map.new
%{
  NorthwindElixirTraders.Order => [:customer_id, :employee_id, :shipper_id],
  NorthwindElixirTraders.OrderDetail => [:order_id, :product_id]
}
```

Clearly, our algorithm in `prioritize/0` is not as robust as we thought… The crucial error is the simplification we undertook in avoiding to create an actual *graph* of dependencies, by relying on the criterion of the length of the list of dependencies; concretely, this line in `prioritize/0`:

```
|> Enum.sort_by(fn {_, dependencies} -> length(dependencies) end)
```

Oh well, live and learn, *and re-learn!* With our simplification we went fast initially, and we will now need to go slow. In **Chapter 10** we discussed (as an aside) the right way to solve the issue of prioritization; remember this part?

> *"A universally applicable approach would be to use `Enum.reduce_while/3` with a cleverly-defined reducer function that will repeatedly "gather" the items of the "connectivity map" and collapse them into a nested map representing a tree of the dependencies between the tables, with the most-dependent table at the tree's "root". […] For now, let's keep it simple […]"*

We judged too early by solving the problem of correctly prioritizing the `Product` module before the `OrderDetail` module. This was a situation in which the length comparison between the two lists of "outbound connections" gave us the misleading impression that we had found a robust simplification in avoiding to properly create a graph.

Oh, but it gets even worse!

```
iex> [Product, OrderDetail] |> Enum.map(&{&1, DataImporter.outbound(&1)}) |> Map.new
%{
  NorthwindElixirTraders.Product => [:category_id, :supplier_id],
  NorthwindElixirTraders.OrderDetail => [:order_id, :product_id]
}
```

The two lists have the same length; so, *what exactly* determined that the records of the `products` table should be imported *before* the records of the `order_details` table…? It's not the `DataImporter.gather/1` function, as it preserves the order of key-value pairs of the dependency map that `DataImporter.make_dependency_map/0` generates. If we "walk through" the function pipeline in `prioritize/0`, it turns out that it's *sheer coincidence* that `products` gets (correctly) prioritized before `order_details`!

Our wishful thinking simplified algorithm was anyway broken, a realization worthy of a Picard facepalm.

As is often the case, our intent to simplify without taking into account the entire problem (and instead cherry-picking what seemed to work at the time as a sign of success, then unconsciously crossing our fingers and YOLOing it), resulted not in a simplification, but in an *oversimplification.* We fell for one of the three wastes of Lean Product and Process Development according to Dr. Allen C. Ward: **wishful thinking**. The other two are: **scatter** and **hand off**. I strongly recommend reading about them in this review or in his excellent, seminal book.

Fret not, however! This can and does happen often in Research & Development—which is exactly what we have been doing so far. We didn't want to "boil the ocean" (as consultants call it), and our oversimplification brought us *this* far. Fine. We got far enough, but have some loose ends to tie. It's now time to pay off the technical debt incurred by this oversimplification.

Building a proper table dependency graph

Let's work on a *proper* implementation of `prioritize/0` that doesn't naively rely on the number of foreign keys per table to (allegedly) determine the degree of table dependency. What we want to do instead is have an algorithm that (here come the requirements):

- *Shall* work for any database *without cyclical dependencies* between tables. This is the case for the Northwind Traders ERD.
- *Shall* be insensitive to "happy little accidents", such as the fact that our current `prioritize/0` works correctly (by chance) for prioritizing `Product` and `OrderDetail`, but not when prioritizing `OrderDetail` ahead of `Order`.
- *Shall* be insensitive to the order in which we "touch" the tables of the original database while developing our Elixir version. In other words, it should have worked even if we would have begun our Northwind Elixir Traders journey by modeling any table other than the "outermost" tables of the ERD.
- *Should* be as simple as we can make it.

Let's think this through. Here are the modules drawn as the nodes of an inverted tree. Every branch of that tree represents the association between two modules with the foreign key shown on the arrow.

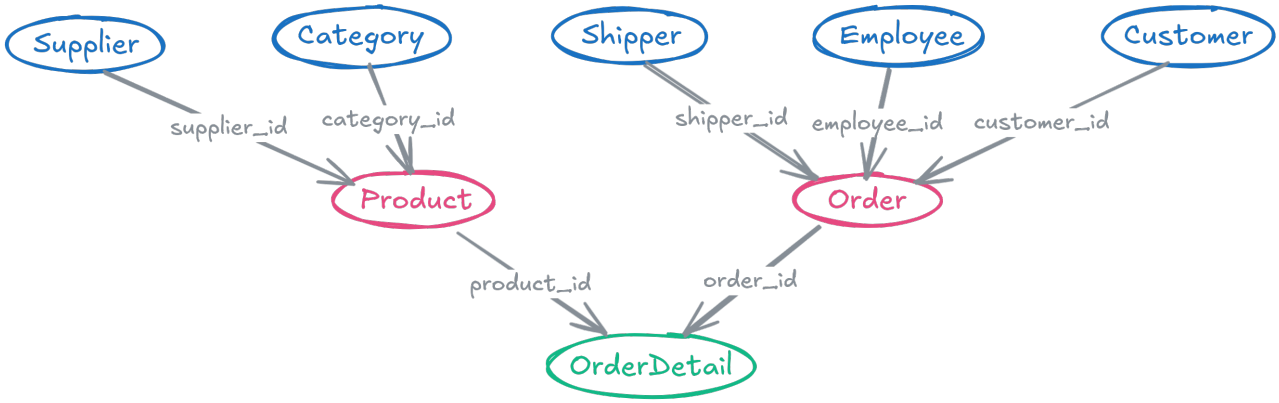

Figure 16.1. The Northwind Traders ERD drawn as an inverted tree

We want to end up with a list of the modules ordered correctly this time. We can still use the output of `make_dependency_map/0` that gives us a map with the module names (the nodes), and for each node the list of foreign keys that "point at it". We previously called the function that generates these notional arrows "`DataImporter.outbound/1`", but in our "inverted tree" diagram the list of foreign keys is actually a list of *inbound* connections to the node.

```
iex> DataImporter.make_dependency_map
%{
  NorthwindElixirTraders.Category => [],
  NorthwindElixirTraders.Employee => [],
  NorthwindElixirTraders.Supplier => [],
  NorthwindElixirTraders.Product => [:category_id, :supplier_id],
  NorthwindElixirTraders.Shipper => [],
  NorthwindElixirTraders.Customer => [],
  NorthwindElixirTraders.Order => [:customer_id, :employee_id, :shipper_id],
  NorthwindElixirTraders.OrderDetail => [:order_id, :product_id]
}
```

Here's an idea: instead of this representation, where each module (key of the map above) has a list of foreign keys (a list of values), let's generate a list of *edges* of this graph; something that will look like this:

```
Order → Customer
Order → Employee
…
OrderDetail → Order
OrderDetail → Product
```

Then, we can implement an algorithm that "walks" over the arrows in this list. First, we need to convert each foreign key name to a module name, e.g. convert `:customer_id` to `NorthwindElixirTraders.Customer`.

```
iex> :customer_id |> Atom.to_string |> String.replace("_id", "") |> String.capitalize
"Customer"
```

Next, we need to go from this string to the module name `NorthwindElixirTraders.Customer` module name. The application can be extracted from `__MODULE__`, as we did in the first part of the function pipeline within our original `DataImporter.get_application/0` (the one that we replaced with `NorthwindElixirTraders.get_application/0` in the previous chapter). Note that since we're running this within IEx, `__MODULE__` is not available, but we can use the `:string` key's value of our new `get_application/0` function:

```
iex> __MODULE__
nil
iex> NorthwindElixirTraders.get_application().string
"NorthwindElixirTraders"
```

We then use `Module.concat/1` to generate the module name. Let's add the completed function to `data_importer.ex`:

Figure 16.2. A helper function that converts a foreign key to the associated module name

```
defmodule NorthwindElixirTraders.DataImporter do
  …
  def fk_to_module(foreign_key) when is_atom(foreign_key) do
    app = NorthwindElixirTraders.get_application() |> Map.get(:string)
    module_name = foreign_key |> Atom.to_string() |> String.replace("_id", "") |> String.capitalize()
    Module.concat(app, module_name)
  end
  …
end
```

Recompile and try it out:

```
iex> DataImporter.fk_to_module(:customer_id)
NorthwindElixirTraders.Customer
```

We can ignore the key-value pairs with an empty list of foreign keys from the output of `make_dependency_map/0`:

```
iex> DataImporter.make_dependency_map |> Enum.filter(&!Enum.empty?(elem(&1, 1)))
…
[
  {NorthwindElixirTraders.Product, [:category_id, :supplier_id]},
  {NorthwindElixirTraders.Order, [:customer_id, :employee_id, :shipper_id]},
  {NorthwindElixirTraders.OrderDetail, [:order_id, :product_id]}
]
```

We also want to invert the direction of the "arrows", so instead of `Order → Customer`, we want `Customer → Order`:

```
iex> fk_modules = DataImporter.make_dependency_map |> Enum.filter(&!Enum.empty?(elem(&1, 1))) |>
Enum.map(fn {k, vv} -> Enum.map(vv, fn v -> {v, k} end) end) |> List.flatten
[
  category_id: NorthwindElixirTraders.Product,
  supplier_id: NorthwindElixirTraders.Product,
  customer_id: NorthwindElixirTraders.Order,
  employee_id: NorthwindElixirTraders.Order,
  shipper_id: NorthwindElixirTraders.Order,
  order_id: NorthwindElixirTraders.OrderDetail,
  product_id: NorthwindElixirTraders.OrderDetail
]
```

Great! Now all we need to do is `Enum.map/2` the `fk_to_module/1` function across the tuples in the list:

```
iex> edges = fk_modules |> Enum.map(fn {k, v} ->
{DataImporter.fk_to_module(k), v} end) |> Map.new
%{
  NorthwindElixirTraders.Category => NorthwindElixirTraders.Product,
  NorthwindElixirTraders.Employee => NorthwindElixirTraders.Order,
  NorthwindElixirTraders.Supplier => NorthwindElixirTraders.Product,
  NorthwindElixirTraders.Product => NorthwindElixirTraders.OrderDetail,
  NorthwindElixirTraders.Shipper => NorthwindElixirTraders.Order,
  NorthwindElixirTraders.Customer => NorthwindElixirTraders.Order,
  NorthwindElixirTraders.Order => NorthwindElixirTraders.OrderDetail
}
```

This is, essentially, our "inverted" tree diagram, in Elixir map form; in other words, the edges of our graph. Written differently, and ordered to look more like our diagram, from left to right, and in an order that more closely resembles the flow of goods and information in the business:

```
Supplier → Product
Category → Product
Product  → OrderDetail
Order    → OrderDetail
Employee → Order
Shipper  → Order
Customer → Order
```

How do we go from this to an ordered list? Again, let's think about it. We want to convert the de-duplicated list of module names from the latest map to a list that's ordered in order of increasing degree of dependency. If a key of the map is not part of the list of values of the map, then that key (that module) can be imported first.

I'll hide the `NorthwindElixirTraders.` prefix to make the result more readable.

```
iex> edges |> Map.keys() |> Enum.filter(&!Enum.member?(Map.values(edges), &1))
[Category, Employee, Supplier, Shipper, Customer]
```

Indeed—the modules in this list are the ones on the top-most row of our tree diagram. We can import these first. What if we invert the condition in `Enum.member?/2`?

```
iex> edges |> Map.keys() |> Enum.filter(&Enum.member?(Map.values(edges), &1))
[Order, Product]
```

These are the modules that are "pointed at" from upstream nodes, i.e. from the earlier list. These need to be imported second in sequence. Finally, we are left with the `OrderDetail` module. This module does not appear in the list of keys of the map; therefore, it has no outbound connections to other modules. It only appears in the list of values of the map, so it only has "inbound" connections from some other modules; concretely, from the `Order` and `Product` modules above it on the diagram. Therefore, as we already know, `%OrderDetail{}` records must be imported last.

How do we turn this into an algorithm, so that we don't presuppose what we already know about the ERD? After all, the Northwind Traders ERD has a small number of tables, but what if we wanted to import data from a database with a tree diagram with more than 3 "layers"?

We can think about this recursively:

1. We have a list of nodes that we want to prioritize. This is our initial state of the "candidate" nodes to be sorted.
2. If a node has no inbound connections, but has outbound connections, then it must take priority. So, we strike it from the list of candidates and place it in the list of "sorted" nodes. Effectively, we remove all entries from the map with a key equal to the node name.
3. We are left with a pruned tree, i.e. a pruned list of nodes to prioritize.
4. Go to step 1 and apply the same process on the remaining nodes that needs to prioritized.

The nodes that are left over in our list of candidates are the last ones to be imported, so we can finally append them to the sorted list. In other words, we process a list of candidates and *accumulate* them based on certain criteria, recursively, into a list of sorted nodes. Where does an *accumulator* appear in Elixir? In the `Enum.reduce/3` function, which we have already used a couple of times so far in this book. Our accumulator `acc` will start as an empty list. We'll somehow *reduce* the list of keys of the `edges` map recursively and accumulate them into `acc`, but sorted.

Implementing a Depth-First Search algorithm

But wait—do we need to implement *everything* from scratch? The following formulation of the problem seems like something for which algorithms must already exist:

> *Given a list of edges of a graph, sort the nodes in order of degree of dependency.*

In fact, this problem is named *topological sorting*, and an algorithm described on the Wikipedia page to solve the problem is called *Kahn's algorithm*.

> *"First, find a list of "start nodes" that have no incoming edges and insert them into a set S; at least one such node must exist in a non-empty (finite) acyclic graph."*

The ERD of Northwind Traders is *acyclic*, i.e. contains no loops. It is also, obviously, *non-empty*. We already know how to find the list of nodes that have no incoming edges; this is the set S that we saw earlier:

```
iex> s = edges |> Map.keys() |> Enum.filter(&!Enum.member?(Map.values(edges), &1))
[Category, Employee, Supplier, Shipper, Customer]
```

Then, we would have to "draw the rest of the owl" by applying Kahn's algorithm. The other algorithm described on the Wikipedia search is *Depth-first Search* and looks simpler (emphasis mine):

> *"The algorithm loops through each node of the graph, **in an arbitrary order**, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort or the node has no outgoing edges (i.e., a leaf node)."*

I particularly like the arbitrary order, and the fact that—in contrast to Kahn's algorithm—we don't need to keep track of the number of inbound connections to each node. Additionally, the fact that it's depth-first search ("DFS") means that it's a great fit for recursion, and recursion is how we often do things in functional programming, instead of using `for` loops.

Our `DataImporter` module has already grown to include many functions, but a few of those could become deprecated after we implement DFS, so let's implement DFS within the same module. We can prune any unused functions later. This is the pseudo-code, straight from Wikipedia:

```
L ← Empty list that will contain the sorted nodes
while exists nodes without a permanent mark do
    select an unmarked node n
    visit(n)

function visit(node n)
    if n has a permanent mark then
        return
    if n has a temporary mark then
        stop   (graph has at least one cycle)

    mark n with a temporary mark

    for each node m with an edge from n to m do
        visit(m)

    mark n with a permanent mark
    add n to head of L
```