What´s new in Ecto 2.1

Table of Contents

Foreword

In January 2017, we will celebrate 5 years since we decided to invest in Elixir. Back in 2012, José Valim, our co-founder and partner, presented us the idea of a programming language that would be expressive, embrace productivity in its tooling, and leverage the Erlang VM to not only tackle the problems in writing concurrent software but also to build fault-tolerant and distributed systems.

Elixir continued, in some sense, to be a risky project for months. We were certainly excited about spreading functional, concurrent and distributed programming concepts to more and more developers, hoping it would lead to a positive impact on the software development industry, but developing a language is a long-term effort that may never become concrete.

During the summer of 2013, other companies and developers started to show interest on Elixir. We heard about companies using it in production, more developers began to contribute and create their own projects, different publishers were writing books on the language, and so on. Such events gave us the confidence to invest more in Elixir and bring the language to version 1.0.

Once Elixir 1.0 was launched in September 2014, we turned our focus to the web platform. We tidied up Plug, the building block for writing web applications in Elixir. We also focused intensively on Ecto, bringing it to version 1.0 together with the Ecto team, and then worked alongside Chris McCord and team to get the first major Phoenix release out. During this time we also started other community centric initiatives, such as Elixir Radar, and began our first commercial Elixir projects.

Today, both the community and our open source projects are showing steady and healthy growth. Elixir is a stable language with continuous improvements landed in minor versions. Plug continues to be a solid foundation for frameworks such as Phoenix. Ecto, however, required more than a small nudge in the right direction. We realized that we needed to let go of old, harmful habits and make Ecto less of an abstraction layer and more of a tool you control and apply to different problems.

This book is the final effort behind Ecto 2.0. It showcases the new direction we have planned for Ecto, the structural improvements made by the Ecto team and many of its new features. We hope you will enjoy it. After all, it is time to let go of past habits.

Have fun,

- The Plataformatec team

Introduction

Ecto 2.0 is a substantial departure from earlier versions. Instead of thinking about models, Ecto 2.0 aims to provide developers a wide range of data-centric tools. Therefore, in order to use Ecto 2.0 effectively, we must learn how to wield those tools properly. That's the goal of this book.

This book, however, is not an introduction to Ecto. If you have never used Ecto before, we recommend you to get started with Ecto's documentation and learn more about repositories, queries, schemas and changesets. We assume the reader is familiar with these building blocks and how they relate to each other.

The first chapters of the book will cover the biggest conceptual changes in Ecto 2.0. We will talk about relational mappers in "Ecto is not your ORM" and then explore Schemaless Queries and the relationship between Schemas and Changesets.

After we will take a deeper look into queries, discussing how to target different databases via query prefixes, as well as the new aggregate and subquery features. Then we will go back to schemas and discuss the schema-related enhancements that are now part of Ecto, such as `many_to_many` associations.

Finally, we will explore brand new topics, like the new Ecto SQL Sandbox, that allows developers to run tests against the database concurrently, as well as `Ecto.Multi`, which makes working with transactions simpler than ever.

This book was also updated to describe features that were introduced in Ecto 2.1 throughout the chapters, such as the dynamic macro, upsert support and improved subqueries.

Acknowledgments

We want to thank the Ecto team for their fantastic work behind Ecto: Eric Meadows-Jönsson, James Fish, José Valim and Michał Muskała. We also thank everyone who has contributed to Ecto, be it with code, documentation, by writing articles, giving presentations, organizing workshops, etc.

Finally we appreciate everyone who has reviewed our beta editions and sent us feedback: Adam Rutkowski, Alkis Tsamis, Christian von Roques, Curtis Ekstrom, Eric Meadows-Jönsson, Jeremy Miranda, John Joseph Sweeney, Kevin Baird, Kevin Rankin, Michael Madrid, Michał Muskała, Po Chen, Raphael Vidal, Steve Pallen, Tobias Pfeiffer, Victoria Wagman and Wojtek Mach.

Ecto is not your ORM

Depending on your perspective, this is a rather bold or obvious statement to start this book. After all, Elixir is not an object-oriented language, so Ecto can't be an Object-relational Mapper. However, this statement is slightly more nuanced than it looks and there are important lessons to be learned here.

O is for Objects

At its core, objects couple state and behaviour together. In the same `user` object, you can have data, like the `user.name`, as well as behaviour, like confirming a particular user account via `user.confirm()`. While some languages enforce different syntaxes between accessing data (`user.name` without parentheses) and behaviour (`user.confirm()` with parentheses), other languages follow the Uniform Access Principle in which an object should not make a distinction between the two syntaxes. Eiffel and Ruby are languages that follow such principle.

Elixir fails the "coupling of state and behaviour" test. In Elixir, we work with different data structures such as tuples, lists, maps and others. Behaviour cannot be attached to data structures. Behaviour is always added to modules via functions.

When there is a need to work with structured data, Elixir provides structs. Structs define a set of fields. A struct will be referenced by the name of the module where it is defined:

```
defmodule User do
  defstruct [:name, :email]
end

user = %User{name: "John Doe", email: "john@example.com"}
```

Once a user struct is created, we can access its email via `user.email`. However, structs are only data. It is impossible to invoke `user.confirm()` on a particular struct in a way it will execute code related to e-mail confirmation.

Although we cannot attach behaviour to structs, it is possible to add functions to the same module that defines the struct:

```
defmodule User do
  defstruct [:name, :email]

  def confirm(user) do
    # Confirm the user email
  end
end
```

Even with the definition above, it is impossible in Elixir to confirm a given user by calling `user.confirm()`. Instead, the `User` prefix is required and the `user` struct must be explicitly given as argument, as in `User.confirm(user)`. At the end of the day, there is no structural coupling between the `user` struct and the functions in the `User` module. Hence Elixir does not have *methods*, it has *functions*.

Without having objects, Ecto certainly can't be an ORM. However, if we let go of the letter "O" for a second, can Ecto still be a relational mapper?

Relational mappers

An Object-Relational Mapper is a technique for converting data between incompatible type systems, commonly databases, to objects, and back.

Similarly, Ecto provides schemas that maps *any* data source into an Elixir struct. When applied to your database, Ecto schemas are relational mappers. Therefore, while Ecto is not a relational mapper, it *contains* a relational mapper as part of the many different tools it offers.

For example, the schema below ties the fields `name`, `email`, `inserted_at` and `updated_at` to fields similarly named in the `users` table:

```
defmodule MyApp.User do
  use Ecto.Schema

  schema "users" do
    field :name
    field :email
    timestamps()
  end
end
```

The appeal behind schemas is that you define the shape of the data once and you can use this shape to retrieve data from the database as well as coordinate changes happening on the data:

```
MyApp.User
|> MyApp.Repo.get!(13)
|> Ecto.Changeset.cast([name: "new name"], [:name, :email])
|> MyApp.Repo.update!
```

By relying on the schema information, Ecto knows how to read and write data without extra input from the developer. In small applications, this coupling between the data and its representation is desired. However, when used wrongly, it leads to complex codebases and sub par solutions.

It is important to understand the relationship between Ecto and relational mappers because saying "Ecto is not your ORM" does not automatically save Ecto schemas from some of the downsides many developers associate ORMs with.

Here are some examples of issues often associated with ORMs that Ecto developers may run into when using schemas:

- Projects using Ecto may end-up with "God Schemas", commonly referred as "God Models", "Fat Models" or "Canonical Models" in some languages and frameworks. Such schemas could contain hundreds of fields, often reflecting bad decisions done at the data layer. Instead of providing one single schema with fields that span multiple concerns, it is better to break the schema across multiple contexts. For example, instead of a single `MyApp.User` schema with dozens of fields, consider breaking it into `MyApp.Accounts.User`, `MyApp.Purchases.User` and so on. Each struct with fields exclusive to its enclosing context
- Developers may excessively rely on schemas when sometimes the best way to retrieve data from the database is into regular data structures (like maps and tuples) and not pre-defined shapes of data like structs. For example, when doing searches, generating

reports and others, there is no reason to rely or return schemas from such queries, as it often relies on data coming from multiple tables with different requirements

- Developers may try to use the same schema for operations that may be quite different structurally. Many applications would bolt features such as registration, account login, into a single User schema, while handling each operation individually, possibly using different schemas, would lead to simpler and clearer solutions

In the next two chapters, we want to break those "bad practices" apart by exploring how to use Ecto with no or multiple schemas per context. By learning how to insert, delete, manipulate and validate data with and without schemas, we hope developers will feel comfortable with building complex applications without relying on one-size-fits-all schemas.

Schemaless queries

Most queries in Ecto are written using schemas. For example, to retrieve all posts in a database, one may write:

```
MyApp.Repo.all(Post)
```

In the construct above, Ecto knows all fields and their types in the schema, rewriting the query above to:

```
MyApp.Repo.all(from p in Post, select: %Post{title: p.title, body: p.body, ...})
```

Interestingly, back in Ecto's early days, there was no such thing as schemas. Queries could only be written directly against a database table by passing the table name as a string:

```
MyApp.Repo.all(from p in "posts", select: {p.title, p.body})
```

When writing schemaless queries, the select expression must be explicitly written with all the desired fields.

While the above syntax made it into Ecto 1.0, by the time Ecto 1.0 was launched, most of the development focus in Ecto had changed towards schemas. This means while developers were able to read data without schemas, they were often too verbose. Not only that, if you wanted to insert entries to your database without schemas, you were out of luck.

Ecto 2.0 levels up the game by adding many improvements to schemaless queries, not only by improving the syntax for reading and updating data, but also by allowing all database operations to be expressed without a schema.

insert_all

One of the functions added to Ecto 2.0 is `Ecto.Repo.insert_all/3`. With `insert_all`, developers can insert multiple entries at once into a repository:

```
MyApp.Repo.insert_all(Post, [[title: "hello", body: "world"],
                             [title: "another", body: "post"]])
```

Although `insert_all` is just a regular Elixir function, it plays an important role in Ecto 2.0 as it allows developers to read, create, update and delete entries without a schema. `insert_all` was the last piece of the puzzle. Let's see some examples.

If you are writing a reporting view, it may be counter-productive to think how your existing application schemas relate to the report being generated. It is often simpler to write a query that returns only the data you need, without trying to fit the data into existing schemas:

```
import Ecto.Query

def running_activities(start_at, end_at)
  MyApp.Repo.all(
    from u in "users",
      join: a in "activities",
      on: a.user_id == u.id,
      where: a.start_at > type(^start_at, :naive_datetime) and
             a.end_at < type(^end_at, :naive_datetime),
      group_by: a.user_id,
      select: %{user_id: a.user_id, interval: a.end_at - a.start_at, count: count(u.id
)}
  )
end
```

The function above does not rely on schemas. It returns only the data that matters for building the report. Notice how we use the `type/2` function to specify what is the expected type of the argument we are interpolating, benefiting from the same type casting guarantees a schema would give.

Inserts, updates and deletes can also be done without schemas via `insert_all`, `update_all` and `delete_all` respectively:

```
# Insert data into posts and return its ID
[%{id: id}] =
  MyApp.Repo.insert_all "posts", [[title: "hello"]], returning: [:id]

# Use the ID to trigger updates
post = from p in "posts", where: [id: ^id]
{1, _} = MyApp.Repo.update_all post, set: [title: "new title"]

# As well as for deletes
{1, _} = MyApp.Repo.delete_all post
```

It is not hard to see how these operations directly map to their SQL variants, keeping the database at your fingertips without the need to intermediate all operations through schemas.

Simpler queries

Besides supporting schemaless inserts, updates and deletes queries, with varying degrees of complexity, Ecto 2.0 also makes regular schemaless queries more expressive.

One example is the ability to select all desired fields without duplication. In early versions, you would have to write verbose select expressions such as:

```
from p in "posts", select: %{title: p.title, body: p.body}
```

With Ecto 2.0 you can simply pass the desired list of fields directly:

```
from "posts", select: [:title, :body]
```

The two queries above are equivalent. When a list of fields is given, Ecto will automatically convert the list of fields to a map or a struct.

Support for passing a list of fields or keyword lists has been added to almost all query constructs in Ecto 2.0. For example, we can use an update query to change the title of a given post without a schema:

```
def update_title(post, new_title) do
  query = from "posts", where: [id: ^post.id], update: [set: [title: ^new_title]]
  MyApp.Repo.update_all(query)
end
```

The `update` construct supports four commands:

- `:set` - sets the given column to the given values

- `:inc` - increments the given column by the given value
- `:push` - pushes (appends) the given value to the end of an array column
- `:pull` - pulls (removes) the given value from an array column

For example, we can increment a column atomically by using the `:inc` command, with or without schemas:

```
def increment_page_views(post) do
  query = from "posts", where: [id: ^post.id], update: [inc: [page_views: 1]]
  MyApp.Repo.update_all(query)
end
```

By allowing regular data structures to be given to most query operations, Ecto 2.0 makes queries with and without schemas more accessible. Not only that, it also enables developers to write dynamic queries, where fields, filters, ordering cannot be specified upfront. We will explore such with more details in upcoming chapters. For now, let's continue exploring schemas in the context of changesets.

Schemas and changesets

In the last chapter we learned how to perform all database operations, from insertion to deletion, without using a schema. While we have been exploring the ability to write constructs without schemas, we haven't discussed what schemas actually are. In this chapter, we will rectify that.

In this chapter we will take a look at the role schemas play when validating and casting data through changesets. As we will see, sometimes the best solution is not to completely avoid schemas, but break a large schema into smaller ones. Maybe one for reading data, another for writing. Maybe one for your database, another for your forms.

Schemas are mappers

The Ecto documentation says:

> An Ecto schema is used to map *any* data source into an Elixir struct.

We put emphasis on *any* because it is a common misconception to think Ecto schemas map only to your database tables.

For instance, when you write a web application using Phoenix and you use Ecto to receive external changes and apply such changes to your database, we have this mapping:

```
Database <-> Ecto schema <-> Forms / API
```

Although there is a single Ecto schema mapping to both your database and your API, in many situations it is better to break this mapping in two. Let's see some practical examples.

Imagine you are working with a client that wants the "Sign Up" form to contain the fields "First name", "Last name" along side "E-mail" and other information. You know there are a couple problems with this approach.

First of all, not everyone has a first and last name. Although your client is decided on presenting both fields, they are a UI concern, and you don't want the UI to dictate the shape of your data. Furthermore, you know it would be useful to break the "Sign Up" information across two tables, the "accounts" and "profiles" tables.

Given the requirements above, how would we implement the Sign Up feature in the backend?

One approach would be to have two schemas, Account and Profile, with virtual fields such as `first_name` and `last_name`, and use associations along side nested forms to tie the schemas to your UI. One of such schemas would be:

```
defmodule Profile do
  use Ecto.Schema

  schema "profiles" do
    field :name
    field :first_name, :string, virtual: true
    field :last_name, :string, virtual: true
    ...
  end
end
```

It is not hard to see how we are polluting our Profile schema with UI requirements by adding fields such `first_name` and `last_name`. If the Profile schema is used for both reading and writing data, it may end-up in an awkward place where it is not useful for any, as it contains fields that map just to one or the other operation.

One alternative solution is to break the "Database <-> Ecto schema <-> Forms / API" mapping in two parts. The first will cast and validate the external data with its own structure which you then transform and write to the database. For such, let's define a schema named `Registration` that will take care of casting and validating the form data exclusively, mapping directly to the UI fields:

```
defmodule Registration do
  use Ecto.Schema

  embedded_schema do
    field :first_name
    field :last_name
    field :email
  end
end
```

We used `embedded_schema` because it is not our intent to persist it anywhere. With the schema in hand, we can use Ecto changesets and validations to process the data:

```
fields = [:first_name, :last_name, :email]

changeset =
  %Registration{}
  |> Ecto.Changeset.cast(params["sign_up"], fields)
  |> validate_required(...)
  |> validate_length(...)
```

Now that the registration changes are mapped and validated, we can check if the resulting changeset is valid and act accordingly:

```
if changeset.valid? do
  # Get the modified registration struct out of the changeset
  registration = Ecto.Changeset.apply_changes(changeset)

  MyApp.Repo.transaction fn ->
    MyApp.Repo.insert_all "accounts", [Registration.to_account(registration)]
    MyApp.Repo.insert_all "profiles", [Registration.to_profile(registration)]
  end

  {:ok, registration}
else
  # Annotate the action we tried to perform so the UI shows errors
  changeset = %{changeset | action: :registration}
  {:error, changeset}
end
```

The `to_account/1` and `to_profile/1` functions in `Registration` would receive the registration struct and split the attributes apart accordingly:

```
def to_account(registration) do
  Map.take(registration, [:email])
end

def to_profile(%{first_name: first, last_name: last}) do
  %{name: "#{first} #{last}"}
end
```

In the example above, by breaking apart the mapping between the database and Elixir and between Elixir and the UI, our code becomes clearer and our data structures simpler.

Note we have used `MyApp.Repo.insert_all/2` to add data to both "accounts" and "profiles" tables directly. We have chosen to bypass schemas altogether. However, there is nothing stopping you from also defining both `Account` and `Profile` schemas and changing `to_account/1` and `to_profile/1` to respectively return `%Account{}` and `%Profile{}` structs. Once structs are returned, they could be inserted through the usual `MyApp.Repo.insert/2` operation. Doing so can be especially useful if there are uniqueness or other constraints that you want to check during insertion.

Schemaless changesets

Although we chose to define a `Registration` schema to use in the changeset, Ecto 2.0 also allows developers to use changesets without schemas. We can dynamically define the data and their types. Let's rewrite the registration changeset above to bypass schemas:

```
data  = %{}
types = %{first_name: :string, last_name: :string, email: :string}

changeset =
  {data, types} # The data+types tuple is equivalent to %Registration{}
  |> Ecto.Changeset.cast(params["sign_up"], Map.keys(types))
  |> validate_required(...)
  |> validate_length(...)
```

You can use this technique to validate API endpoints, search forms, and other sources of data. The choice of using schemas depends mostly if you want to use the same mapping in different places or if you desire the compile-time guarantees Elixir structs gives you. Otherwise, you can bypass schemas altogether, be it when using changesets or interacting with the repository.

However, the most important lesson in this chapter is not when to use or not to use schemas, but rather understand when a big problem can be broken into smaller problems that can be solved independently leading to an overall cleaner solution. The choice of using

schemas or not above didn't affect the solution as much as the choice of breaking the registration problem apart.

Dynamic queries

Ecto was designed from the ground up to have an expressive query API that leverages Elixir syntax to write queries that are pre-compiled for performance and safety. When building queries, we may use the keywords syntax

```
import Ecto.Query

from p in Post,
  where: p.author == "José" and p.category == "Elixir",
  where: p.published_at > ^minimum_date,
  order_by: [desc: p.published_at]
```

or the pipe-based one

```
import Ecto.Query

Post
|> where([p], p.author == "José" and p.category == "Elixir")
|> where([p], p.published_at > ^minimum_date)
|> order_by([p], desc: p.published_at)
```

While many developers prefer the pipe-based syntax, having to repeat the binding `p` made it quite verbose compared to the keyword one. Furthermore, the compile-time aspect of Ecto queries was at odds with building queries dynamically.

Imagine for example a web application that provides search functionality on top of existing posts. The user should be able to specify multiple criteria, such as the author name, the post category, publishing interval, etc.

In Ecto 1.0, the only way to write such functionality would be via `Enum.reduce/3`:

```
def filter(params) do
  Enum.reduce(params, Post, &filter/2)
end

defp filter({"author", author}, query) do
  where(query, [p], p.author == ^author)
end
defp filter({"category", category}, query) do
  where(query, [p], p.category == ^category)
end
defp filter({"published_at", minimum_date}, query) do
  where(query, [p], p.published_at > ^minimum_date)
end
defp filter({"order_by", "published_at_asc"}, query) do
  order_by(query, [p], asc: p.published_at)
end
defp filter({"order_by", "published_at_desc"}, query) do
  order_by(query, [p], desc: p.published_at)
end
defp filter(_ignore_unknown, query) do
  query
end
```

While the code above works fine, it couples the processing of the parameters with the query generation. It is a verbose implementation that is also hard to test since the result of filtering and handling of parameters are stored directly in the query struct.

A better approach would be to process the parameters into regular data structures and then build the query as late as possible. That's exactly what Ecto 2.0 allows us to do.

Focusing on data structures

Ecto 2.0 provides a simpler API for both keyword and pipe based queries by making data structures first-class. Let's rewrite the original queries to use data structures when possible:

```
from p in Post,
  where: [author: "José", category: "Elixir"],
  where: p.published_at > ^minimum_date,
  order_by: [desc: :published_at]
```

and

```
Post
|> where(author: "José", category: "Elixir")
|> where([p], p.published_at > ^minimum_date)
|> order_by(desc: :published_at)
```

Notice how we were able to ditch the `p` selector in most expressions. In Ecto 2.0, all constructs, from `select` and `order_by` to `where` and `group_by`, accept data structures as input. The data structure can be specified at compile-time, as above, and also dynamically at runtime, shown below:

```
where = [author: "José", category: "Elixir"]
order_by = [desc: :published_at]
Post
|> where(^where)
|> where([p], p.published_at > ^minimum_date)
|> order_by(^order_by)
```

The advantage of interpolating data structures is that we can decouple the processing of parameters from the query generation. Note however not all expressions can be converted to data structures. Since `where` converts a key-value to a `key == value` comparison, order-based comparisons such as `p.published_at > ^minimum_date` still need to be written as part of the query as before.

Luckily, Ecto 2.1 solves this issue.

The dynamic macro

For cases where we cannot rely on data structures but still desire to build queries dynamically, Ecto 2.1 includes the `Ecto.Query.dynamic/2` macro.

In order to understand how the `dynamic` macro works let's rewrite the `filter/1` function from the beginning of this chapter using both data structures and the `dynamic` macro. The example below requires Ecto 2.1:

```
def filter(params) do
  Post
  |> order_by(^filter_order_by(params["order_by"]))
  |> where(^filter_where(params))
  |> where(^filter_published_at(params["published_at"]))
end

def filter_order_by("published_at_desc"), do: [desc: :published_at]
def filter_order_by("published_at"),      do: [asc:  :published_at]
def filter_order_by(_),                   do: []

def filter_where(params) do
  for key <- [:author, :category],
      value = params[Atom.to_string(key)],
      do: {key, value}
end

def filter_published_at(date) when is_binary(date),
  do: dynamic([p], p.published_at > ^date)
def filter_published_at(_date),
  do: true
```

The `dynamic` macro allows us to build dynamic expressions that are later interpolated into the query. `dynamic` expressions can also be interpolated into dynamic expressions, allowing developers to build complex expressions dynamically without hassle.

Because we were able to break our problem into smaller functions that receive regular data structures, we can use all the tools available in Elixir to work with data. For handling the `order_by` parameter, it may be best to simply pattern match on the `order_by` parameter. For building the `where` clause, we can traverse the list of known keys and convert them to the format expected by Ecto. For complex conditions, we use the `dynamic` macro.

Testing also becomes simpler as we can test each function in isolation, even when using dynamic queries:

```
test "filter published at based on the given date" do
  assert inspect(filter_published_at("2010-04-17")) ==
         "dynamic([p], p.published_at > ^\"2010-04-17\")"
  assert inspect(filter_published_at(nil)) ==
         "true"
end
```

While at the end of the day some developers may feel more comfortable with using the `Enum.reduce/3` approach, Ecto 2.0 and later gives us the option to choose which approach works best.

Thanks to Michał Muskała for suggestions and feedback on this chapter.

Multi tenancy with query prefixes

Ecto 2.0 introduces the ability to run queries in different prefixes using a single pool of database connections. For databases engines such as Postgres, Ecto's prefix maps to Postgres' DDL schemas. For MySQL, each prefix is a different database on its own.

Query prefixes may be useful in different scenarios. For example, multi tenant apps running on Postgres would define multiple prefixes, usually one per client, under a single database. The idea is that prefixes will provide data isolation between the different users of the application, guaranteeing either globally or at the data level that queries and commands act on a specific prefix.

Prefixes may also be useful on high-traffic applications where data is partitioned upfront. For example, a gaming platform may break game data into isolated partitions, each named after a different prefix. A partition for a given player is either chosen at random or calculated based on the player information.

While query prefixes were designed with the two scenarios above in mind, they may also be used in other circumstances, which we will explore throughout this chapter. All the examples below assume you are using Postgres. Other databases engines may require slightly different solutions.

Global prefixes

As a starting point, let's start with a simple scenario: your application must connect to a particular prefix when running in production. This may be due to infrastructure conditions, database administration rules or others.

Let's define a repository and a schema to get started:

```
# lib/repo.ex
defmodule MyApp.Repo do
  use Ecto.Repo, otp_app: :my_app
end

# lib/sample.ex
defmodule MyApp.Sample do
  use Ecto.Schema

  schema "samples" do
    field :name
    timestamps
  end
end
```

Now let's configure the repository:

```
# config/config.exs
config :my_app, MyApp.Repo,
  adapter: Ecto.Adapters.Postgres,
  username: "postgres",
  password: "postgres",
  database: "demo",
  hostname: "localhost",
  pool_size: 10
```

And define a migration:

```
# priv/repo/migrations/20160101000000_create_sample.exs
defmodule MyApp.Repo.Migrations.CreateSample do
  use Ecto.Migration

  def change do
    create table(:samples) do
      add :name, :string
      timestamps()
    end
  end
end
```

Now let's create the database, migrate it and then start an IEx session:

```
$ mix ecto.create
$ mix ecto.migrate
$ iex -S mix
Interactive Elixir (1.4.0-dev) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> MyApp.Repo.all MyApp.Sample
[]
```

We haven't done anything unusual so far. We created our database instance, made it up to date by running migrations and then successfully made a query against the "samples" table, which returned an empty list.

By default, connections to Postgres' databases run on the "public" prefix. When we run migrations and queries, they are all running against the "public" prefix. However imagine your application has a requirement to run on a particular prefix in production, let's call it "global_prefix".

Luckily Postgres allows us to change the prefix our database connections run on by setting the "schema search path". The best moment to change the search path is right after we setup the database connection, ensuring all of our queries will run on that particular prefix, throughout the connection life-cycle.

To do so, let's change our database configuration in "config/config.exs" and specify an `:after_connect` option. `:after_connect` expects a tuple with module, function and arguments it will invoke with the connection process, as soon as a database connection is established:

```
config :my_app, MyApp.Repo,
  adapter: Ecto.Adapters.Postgres,
  username: "postgres",
  password: "postgres",
  database: "demo_dev",
  hostname: "localhost",
  pool_size: 10,
  after_connect: {Postgrex, :query!, ["SET search_path TO global_prefix", []]}
```

Now let's try to run the same query as before:

```
$ iex -S mix
Interactive Elixir (1.4.0-dev) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> MyApp.Repo.all MyApp.Sample
** (Postgrex.Error) ERROR (undefined_table): relation "samples" does not exist
```

Our previously successful query now fails because there is no table "samples" under the new prefix. Let's try to fix that by running migrations:

```
$ mix ecto.migrate
** (Postgrex.Error) ERROR (invalid_schema_name): no schema has been selected to create
in
```

Oops. Now migration says there is no such schema name. That's because Postgres automatically creates the "public" prefix every time we create a new database. If we want to use a different prefix, we must explicitly create it on the database we are running on:

```
$ psql -d demo_dev -c "CREATE SCHEMA global_prefix"
```

Now we are ready to migrate and run our queries:

```
$ mix ecto.migrate
$ iex -S mix
Interactive Elixir (1.4.0-dev) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> MyApp.Repo.all MyApp.Sample
[]
```

Data in different prefixes are isolated. Writing to the "samples" table in one prefix cannot be accessed by the other unless we change the prefix in the connection or use the Ecto conveniences we will discuss next.

Per-query and per-struct prefixes

While still configured to connect to the "global_prefix" on `:after_connect`, let's run some queries:

```
iex(1) MyApp.Repo.all MyApp.Sample
[]
iex(2) MyApp.Repo.insert %MyApp.Sample{name: "mary"}
{:ok, %MyApp.Sample{...}}
iex(3) MyApp.Repo.all MyApp.Sample
[%MyApp.Sample{...}]
```

The operations above ran on the "global_prefix". Now what happens if we try to run the sample query on the "public" prefix? To do so, let's build a query struct and set the prefix field manually:

```
iex(4)> query = Ecto.Queryable.to_query MyApp.Sample
#Ecto.Query<from s in MyApp.Sample>
iex(5)>  MyApp.Repo.all %{query | prefix: "public"}
[]
```

Notice how we were able to change the prefix the query runs on. Back in the default "public" prefix, there is no data.

Ecto 2.1 also supports the `:prefix` option on all relevant repository operations:

```
iex(6)> MyApp.Repo.all MyApp.Sample
[%MyApp.Sample{...}]
iex(7)> MyApp.Repo.all MyApp.Sample, prefix: "public"
[]
```

One interesting aspect of prefixes in Ecto is that the prefix information is carried along each struct returned by a query:

```
iex(8) [sample] = MyApp.Repo.all MyApp.Sample
[%MyApp.Sample{}]
iex(9)> Ecto.get_meta(sample, :prefix)
nil
```

The example above returned nil, which means no prefix was specified by Ecto, and therefore the database connection default will be used. In this case, "global_prefix" will be used because of the `:after_connect` callback we added at the beginning of this chapter.

Since the prefix data is carried in the struct, we can use such to copy data from one prefix to the other. Let's copy the sample above from the "global_prefix" to the "public" one:

```
iex(10)> public_sample = Ecto.put_meta(sample, prefix: "public")
%MyApp.Sample{}
iex(11)> MyApp.Repo.insert public_sample
{:ok, %MyApp.Sample{}}
iex(12)> [sample] = MyApp.Repo.all MyApp.Sample, prefix: "public"
[%MyApp.Sample{}]
iex(13)> Ecto.get_meta(sample, :prefix)
"public"
```

Now we have data inserted in both prefixes.

Prefixes in queries and structs always cascade. For example, if you run `MyApp.Repo.preload(sample, [:some_association])`, the association will be queried for and loaded in the same prefix as the `sample` struct. If `sample` has associations and you call

`MyApp.Repo.insert(sample)` or `MyApp.Repo.update(sample)`, the associated data will also be inserted/updated in the same prefix as `sample`. That's by design to facilitate working with groups of data in the same prefix, and especially because **data in different prefixes must be kept isolated**.

Migration prefixes

So far we have explored how to set a global prefix using Postgres' and how to set the prefix at the query or struct level. When the global prefix is set, it also changes the prefix migrations run on. However it is also possible to set the prefix through the command line or per table in the migration itself.

For example, imagine you are a gaming company where the game is broken in 128 partitions, named "prefix_1", "prefix_2", "prefix_3" up to "prefix_128". Now, whenever you need to migrate data, you need to migrate data on all different 128 prefixes. There are two ways of achieve that.

The first mechanism is to invoke `mix ecto.migrate` multiple times, once per prefix, passing the `--prefix` option:

```
$ mix ecto.migrate --prefix "prefix_1"
$ mix ecto.migrate --prefix "prefix_2"
$ mix ecto.migrate --prefix "prefix_3"
...
$ mix ecto.migrate --prefix "prefix_128"
```

The other approach is by changing each desired migration to run across multiple prefixes. For example:

```
defmodule MyApp.Repo.Migrations.CreateSample do
  use Ecto.Migration

  def change do
    for i <- 1..128 do
      prefix = "prefix_#{i}"
      create table(:samples, prefix: prefix) do
        add :name, :string
        timestamps()
      end

      # Execute the commands on the current prefix
      # before moving on to the next prefix
      flush()
    end
  end
end
```

Schema prefixes

Finally, Ecto 2.1 adds the ability to set a particular schema to run on a specific prefix. Imagine you are building a multi-tenant application. Each client data belongs to a particular prefix, such as "client_foo", "client_bar" and so forth. Yet your application may still rely on a set of tables that is shared across all clients. One of such tables may be exactly the table that maps the Client ID to its database prefix. Let's assume we want to store this data in a prefix named "main":

```
defmodule MyApp.Mapping do
  use Ecto.Schema

  @schema_prefix "main"
  schema "mappings" do
    field :client_id, :integer
    field :db_prefix
    timestamps
  end
end
```

Now running `MyApp.Repo.all MyApp.Mapping` will by default run on the "main" prefix, regardless of the value configured globally on the `:after_connect` callback. Similar will happen to `insert`, `update`, and similar operations, the `@schema_prefix` is used unless the `:prefix` is explicitly changed via `Ecto.put_meta/2` or by passing the `:prefix` option to the repository operation.

Keep in mind, however, that queries run on a single prefix. For example, if `MyApp.Mapping` on prefix "main" depends on a schema named `MyApp.Other` on prefix "another", a query starting with `MyApp.Mapping` will always run on the "main" prefix. By design it is not possible to perform query joins across prefixes. If data belongs to different prefixes, it is best to not couple them structurally nor via queries, in order to **keep data in different prefixes isolated**.

Summing up

Ecto 2.0 provides many conveniences for working with querying prefixes. Those conveniences have been further improved in Ecto 2.1, allowing developers to configure prefix with different level of granularity:

```
global prefixes > schema prefix > query/struct prefixes
```

This allows developers to tackle different scenarios, from production requirements to multi-tenant applications. Our journey on exploring the new query constructs is almost over. The next and last query chapter is on aggregates and subqueries.

Aggregates and subqueries

The last features we will discuss regarding Ecto queries are aggregates and subqueries. As we will learn, one builds directly on the other.

Aggregates

Ecto 2.0 includes a convenience function in repositories to calculate aggregates.

For example, if we assume every post has an integer column named visits, we can find the average number of visits across all posts with:

```
MyApp.Repo.aggregate(MyApp.Post, :avg, :visits)
#=> #Decimal<1743>
```

Behind the scenes, the query above translates to:

```
MyApp.Repo.one(from p in MyApp.Post, select: avg(p.visits))
```

The `aggregate/3` function supports any of the aggregate operations listed in the Ecto Query API.

At first, it looks like the implementation of `aggregate/3` is quite straight-forward. You could even start to wonder why it was added to Ecto in the first place. However, complexities start to arise on queries that rely on `limit`, `offset` or `distinct` clauses.

Imagine that instead of calculating the average of all posts, you want the average of only the top 10. Your first try may be:

```
MyApp.Repo.one(from p in MyApp.Post,
               order_by: [desc: :visits],
               limit: 10,
               select: avg(p.visits))
#=> #Decimal<1743>
```

Oops. The query above returned the same value as the queries before. The option `limit: 10` has no effect here since it is limiting the aggregated result and queries with aggregates return only a single row anyway. In order to retrieve the correct result, we would need to first find the top 10 posts and only then aggregate. That's exactly what `aggregate/3` does:

```
query = from MyApp.Post, order_by: [desc: :visits], limit: 10
MyApp.Repo.aggregate(query, :avg, :visits) #=> #Decimal<4682>
```

When `limit`, `offset` or `distinct` is specified in the query, `aggregate/3` automatically wraps the given query in a subquery. Therefore the query executed by `aggregate/3` above is rather equivalent to:

```
query = from MyApp.Post, order_by: [desc: :visits], limit: 10
MyApp.Repo.one(from q in subquery(query), select: avg(q.visits))
```

Let's take a closer look at subqueries.

Subqueries

In the previous section we have already learned some queries that would be hard to express without support for subqueries. That's one of many examples that caused subqueries to be added to Ecto.

Subqueries in Ecto are created by calling `Ecto.Query.subquery/1`. This function receives any data structure that can be converted to a query, via the `Ecto.Queryable` protocol, and returns a subquery construct (which is also queryable).

In Ecto 2.0, it is allowed for a subquery to select a whole table (`p`) or a field (`p.field`). All fields selected in a subquery can be accessed from the parent query. Let's revisit the aggregate query we saw in the previous section:

```
query = from MyApp.Post, order_by: [desc: :visits], limit: 10
MyApp.Repo.one(from q in subquery(query), select: avg(q.visits))
```

Because the query does not specify a `:select` clause, it will return `select: p` where `p` is controlled by `MyApp.Post` schema. Since the query will return all fields in `MyApp.Post`, when we convert it to a subquery, all of the fields from `MyApp.Post` will be available on the parent query, such as `q.visits`. In fact, Ecto will keep the schema properties across queries. For example, if you write `q.field_that_does_not_exist`, your Ecto query won't compile.

Ecto 2.1 further improves subqueries by allowing an Elixir map to be returned from a subquery, making the map keys directly available to the parent query.

Let's see one last example. Imagine you manage a library (as in an actual library in the real world) and there is a table that logs every time the library lends a book. The "lendings" table uses an auto-incrementing primary key and can be backed by the following schema:

```
defmodule Library.Lending do
  use Ecto.Schema

  schema "lendings" do
    belongs_to :book, MyApp.Book       # defines book_id
    belongs_to :visitor, MyApp.Visitor # defines visitor_id
  end
end
```

Now consider we want to retrieve the name of every book alongside the name of the last person the library has lent it to. To do so, we need to find the last lending ID of every book, and then join on the book and visitor tables. With subqueries, that's straight-forward:

```
last_lendings =
  from l in MyApp.Lending,
    group_by: l.book_id,
    select: %{book_id: l.book_id, last_lending_id: max(l.id)}

from l in Lending,
  join: last in subquery(last_lendings),
    on: last.last_lending_id == l.id,
  join: b in assoc(l, :book),
  join: v in assoc(l, :visitor),
  select: {b.name, v.name}
```

Subqueries are an important improvement to Ecto which makes it possible to express queries that were not possible before. On top of that, we were able to add features such as aggregates, which provide useful functionality while shielding the user from corner cases.

Improved associations and factories

Ecto 2.0 largely improved how associations work. To understand why and how, let's talk about Ecto's original design goals.

Ecto first started as a Summer of Code project from Eric Meadows-Jönsson, today part of the Elixir team and creator of Hex, with José Valim, creator of Elixir, as mentor. This was back in 2013 while Elixir was still at version 0.9!

Similar to many projects at the time, one of the goals behind Ecto was to validate Elixir itself as a programming language. One of the questions it aimed to answer was: "can Elixir be used to create a database wrapper that is performant and secure?". By focusing on a stable, performant and secure foundation, it would be straight-forward to add conveniences and dynamism later on - while the opposite direction would have been exceptionally hard.

Ecto 1.0 was this secure and performant foundation. Since the focus was on Ecto's building blocks, the higher-level usage felt rigid in many aspects and was frequently reported as limitations by the community.

While Ecto 2.0 fixes some missteps, it mostly builds on top of this foundation by adding the flexibilities users have longed for. We have explored many of them in the first chapters of this book: schemaless queries, schemaless changesets, dynamic queries and more. In the next three chapters, we will explore the enhancements done to schemas and associations.

In this particular chapter, we will learn how Ecto is capable of inserting complex data structures without the need to use changesets and how to use this feature to manage complex data, which may be useful when building database scripts, interacting with your application test suite, and more.

Less changesets

At the same time Ecto 2.0 brings many features to changesets, it makes changesets less necessary throughout Ecto APIs. For example, in Ecto 1.0, `Ecto.Repo.insert/2` required changesets. This means that, in order to insert any entry to the database, such as a post, we had to wrap it in a changeset first:

```
%Post{title: "hello world"}
|> Ecto.Changeset.change
|> Repo.insert!()
```

This reflected throughout Ecto 1.0 APIs. If you wanted to create a post with some comments, you had to wrap each comment in a changeset and then put it in the post changeset:

```
comment1 = %Comment{body: "excellent article"} |> Ecto.Changeset.change
comment2 = %Comment{body: "I learned something new"} |> Ecto.Changeset.change

%Post{title: "hello world"}
|> Ecto.Changeset.put_assoc(:comments, [comment1, comment2])
|> Repo.insert!()
```

Furthermore, when handling associations, Ecto 1.0 forced you to always write the parent changeset first and then the children. While the example above where we insert a post (the parent) with multiple comments (children) worked, the following example would not:

```
post = %Post{title: "hello world"} |> Ecto.Changeset.change()

%Comment{body: "excellent article"}
|> Ecto.Changeset.put_assoc(:post, [post])
|> Repo.insert!()
```

Ecto 2.0 goes away with all of those limitations. You can now also pass structs to the repository and Ecto will take care of building the changesets for you behind the scenes. In Ecto 2.0, a post with comments can be inserted directly as follows:

```
Repo.insert! %Post{
  title: "hello world",
  comments: [
    %Comment{body: "excellent article"},
    %Comment{body: "I learned something new"}
  ]
}
```

You are also able to insert, update and delete associations from any direction, be it from parent to child or child to parent:

```
Repo.insert! %Comment{
  body: "excellent article",
  post: %Post{title: "hello world"}
}
```

This feature is not only useful when writing our applications but also when testing them, as we will see next.

Test factories

Many projects depend on external libraries to build their test data. Some of those libraries are called factories because they provide convenience functions for producing different groups of data. However, given Ecto 2.0 is able to manage complex data trees, we can implement such functionality without relying on third-party projects.

To get started, let's create a file at "test/support/factory.ex" with the following contents:

```
defmodule MyApp.Factory do
  alias MyApp.Repo

  # Factories

  def build(:post) do
    %MyApp.Post{title: "hello world"}
  end

  def build(:comment) do
    %MyApp.Comment{body: "good post"}
  end

  def build(:post_with_comments) do
    %MyApp.Post{
      title: "hello with comments",
      comments: [
        build(:comment, body: "first"),
        build(:comment, body: "second")
      ]
    }
  end

  def build(:user) do
    %MyApp.User{
      email: "hello#{System.unique_integer()}",
      username: "hello#{System.unique_integer()}"
    }
  end

  # Convenience API

  def build(factory_name, attributes) do
    factory_name |> build() |> struct(attributes)
  end

  def insert!(factory_name, attributes \\ []) do
    Repo.insert! build(factory_name, attributes)
  end
end
```

Our factory module defines four "factories" as different clauses to the build function: `:post`, `:comment`, `:post_with_comments` and `:user`. Each clause defines structs with the fields that are required by the database. In certain cases, the generated struct also needs to generate unique fields, such as the user's email and username. We did so by calling Elixir's `System.unique_integer()` - you could call `System.unique_integer([:positive])` if you need a strictly positive number.

At the end, we defined two functions, `build/2` and `insert!/2`, which are conveniences for building structs with specific attributes and for inserting data directly in the repository respectively.

That's literally all that is necessary for building our factories. We are now ready to use them in our tests. First, open up your "mix.exs" and make sure the "test/support/factory.ex" file is compiled:

```
def project do
  [...,
   elixirc_paths: elixirc_paths(Mix.env),
   ...]
end

defp elixirc_paths(:test), do: ["lib", "test/support"]
defp elixirc_paths(_), do: ["lib"]
```

Now in any of the tests that need to generate data, we can import the `MyApp.Factory` module and use its functions:

```
import MyApp.Factory

build(:post)
#=> %MyApp.Post{id: nil, title: "hello world", ...}

build(:post, title: "custom title")
#=> %MyApp.Post{id: nil, title: "custom title", ...}

insert!(:post, title: "custom title")
#=> %MyApp.Post{id: ..., title: "custom title"}
```

By building the functionality we need on top of Ecto capabilities, we are able to extend and improve our factories on whatever way we desire, without being constrained to third-party limitations.

Many to many and casting

Besides `belong_to`, `has_many`, `has_one` and `:through` associations, Ecto 2.0 also includes `many_to_many`. `many_to_many` relationships, as the name says, allows a record from table X to have many associated entries from table Y and vice-versa. Although `many_to_many` associations can be written as `has_many :through`, using `many_to_many` may considerably simplify some workflows.

In this chapter, we will talk about polymorphic associations and how `many_to_many` can remove boilerplate from certain approaches compared to `has_many :through`.

Todo lists v65131

The web has seen its share of todo list applications. But that won't stop us from creating our own!

In our case, there is one aspect of todo list applications we are interested in, which is the relationship where the todo list has many todo items. We have explored this exact scenario in detail in an article we posted on Plataformatec's blog about nested associations and embeds. Let's recap the important points.

Our todo list app has two schemas, `Todo.List` and `Todo.Item`:

```
defmodule MyApp.TodoList do
  use Ecto.Schema

  schema "todo_lists" do
    field :title
    has_many :todo_items, MyApp.TodoItem
    timestamps()
  end
end

defmodule MyApp.TodoItem do
  use Ecto.Schema

  schema "todo_items" do
    field :description
    timestamps()
  end
end
```

One of the ways to introduce a todo list with multiple items into the database is to couple our UI representation to our schemas. That's the approach we took in the blog post with Phoenix. Roughly:

```
<%= form_for @todo_list_changeset, todo_list_path(@conn, :create), fn f -> %>
  <%= text_input f, :title %>
  <%= inputs_for f, :todo_items, fn i -> %>
    ...
  <% end %>
<% end %>
```

When such a form is submitted in Phoenix, it will send parameters with the following shape:

```
%{"todo_list" => %{
  "title" => "shipping list",
  "todo_items" => %{
    0 => %{"description" => "bread"},
    1 => %{"description" => "eggs"},
  }
}}
```

We could then retrieve those parameters and pass it to an Ecto changeset and Ecto would automatically figure out what to do:

```
# In MyApp.TodoList
def changeset(struct, params \\ %{}) do
  struct
  |> Ecto.Changeset.cast(params, [:title])
  |> Ecto.Changeset.cast_assoc(:todo_items, required: true)
end

# And then in MyApp.TodoItem
def changeset(struct, params \\ %{}) do
  struct
  |> Ecto.Changeset.cast(params, [:description])
end
```

By calling `Ecto.Changeset.cast_assoc/3`, Ecto will look for a "todo_items" key inside the parameters given on cast, and compare those parameters with the items stored in the todo list struct. Ecto will automatically generate instructions to insert, update or delete todo items such that:

- if a todo item sent as parameter has an ID and it matches an existing associated todo item, we consider that todo item should be updated
- if a todo item sent as parameter does not have an ID (nor a matching ID), we consider that todo item should be inserted
- if a todo item is currenetly associated but its ID was not sent as parameter, we consider the todo item is being replaced and we act according to the `:on_replace` callback. By default `:on_replace` will raise so you choose a behaviour between replacing, deleting, ignoring or nilifying the association

The advantage of using `cast_assoc/3` is that Ecto is able to do all of the hard work of keeping the entries associated, **as long as we pass the data exactly in the format that Ecto expects**. However, as we learned in the first three chapters of this book, such approach is not always preferrable and in many situations it is better to design our associations differently or decouple our UIs from our database representation.

Polymorphic todo items

To show an example of where using `cast_assoc/3` is just too complicated to be worth it, let's imagine you want your "todo items" to be polymorphic. For example, you want to be able to add todo items not only to "todo lists" but to many other parts of your application, such as projects, milestones, you name it.

First of all, it is important to remember Ecto does not provide the same type of polymorphic associations available in frameworks such as Rails and Laravel. In such frameworks, a polymorphic association uses two columns, the `parent_id` and `parent_type`. For example,

one todo item would have `parent_id` of 1 with `parent_type` of "TodoList" while another would have `parent_id` of 1 with `parent_type` of "Project".

The issue with the design above is that it breaks database references. The database is no longer capable of guaranteeing the item you associate to exists or will continue to exist in the future. This leads to an inconsistent database which end-up pushing workarounds to your application.

The design above is also extremely inefficient. In the past we have worked with a large client on removing such polymorphic references because frequent polymorphic queries were grinding the database to a halt even after adding indexes and optimizing the database.

Luckily, the documentation for the `belongs_to` macro includes examples on how to design sane and performant associations. One of those approaches consists in using many join tables. Besides the "todo_lists" and "projects" tables and the "todo_items" table, we would create "todo_list_items" and "project_items" to associate todo items to todo lists and todo items to projects respectively. In terms of migrations, we are looking at the following:

```
create table(:todo_lists)  do
  add :title
  timestamps()
end

create table(:projects)  do
  add :name
  timestamps()
end

create table(:todo_items)  do
  add :description
  timestamps()
end

create table(:todo_lists_items) do
  add :todo_item_id, references(:todo_items)
  add :todo_list_id, references(:todo_lists)
  timestamps()
end

create table(:projects_items) do
  add :todo_item_id, references(:todo_items)
  add :project_id, references(:projects)
  timestamps()
end
```

By adding one table per association pair, we keep database references and can efficiently perform queries that relies on indexes.

First let's see how implement this functionality in Ecto using a `has_many :through` and then use `many_to_many` to remove a lot of the boilerplate we were forced to introduce.

Polymorphism with has_many :through

Given we want our todo items to be polymorphic, we can no longer associate a todo list to todo items directly. Instead we will create an intermediate schema to tie `MyApp.TodoList` and `MyApp.TodoItem` together.

```
defmodule MyApp.TodoList do
  use Ecto.Schema

  schema "todo_lists" do
    field :title
    has_many :todo_list_items, MyApp.TodoListItem
    has_many :todo_items, through: [:todo_list_items, :todo_item]
    timestamps()
  end
end

defmodule MyApp.TodoListItem do
  use Ecto.Schema

  schema "todo_list_items" do
    belongs_to :todo_list, MyApp.TodoList
    belongs_to :todo_item, MyApp.TodoItem
    timestamps()
  end
end

defmodule MyApp.TodoItem do
  use Ecto.Schema

  schema "todo_items" do
    field :description
    timestamps()
  end
end
```

Although we introduced `MyApp.TodoListItem` as an intermediate schema, `has_many :through` allows us to access all todo items for any todo list transparently:

```
todo_lists |> Repo.preload(:todo_items)
```

The trouble is that `:through` associations are **read-only** since Ecto does not have enough information to fill in the intermediate schema. This means that, if we still want to use `cast_assoc` to insert a todo list with many todo items directly from the UI, we cannot use the `:through` association and instead must go step by step. We would need to first `cast_assoc(:todo_list_items)` from `TodoList` and then call `cast_assoc(:todo_item)` from the `TodoListItem` schema:

```
# In MyApp.TodoList
def changeset(struct, params \\ %{}) do
  struct
  |> Ecto.Changeset.cast(params, [:title])
  |> Ecto.Changeset.cast_assoc(:todo_list_items, required: true)
end

# And then in the MyApp.TodoListItem
def changeset(struct, params \\ %{}) do
  struct
  |> Ecto.Changeset.cast_assoc(:todo_item, required: true)
end

# And then in MyApp.TodoItem
def changeset(struct, params \\ %{}) do
  struct
  |> Ecto.Changeset.cast(params, [:description])
end
```

To further complicate things, remember `cast_assoc` expects a particular shape of data that reflects your associations. In this case, because of the intermediate schema, the data sent through your forms in Phoenix would have to look as follows:

```
%{"todo_list" => %{
  "title" => "shipping list",
  "todo_list_items" => %{
    0 => %{"todo_item" => %{"description" => "bread"}},
    1 => %{"todo_item" => %{"description" => "eggs"}},
  }
}}
```

To make matters worse, you would have to duplicate this logic for every intermediate schema, and introduce `MyApp.TodoListItem` for todo lists, `MyApp.ProjectItem` for projects, etc.

Luckily, `many_to_many` allows us to remove all of this boilerplate.

Polymorphism with many_to_many

In a way, the idea behind `many_to_many` associations is that it allows us to associate two schemas via an intermediate schema while automatically taking care of all details about the intermediate schema. Let's rewrite the schemas above to use `many_to_many`:

```
defmodule MyApp.TodoList do
  use Ecto.Schema

  schema "todo_lists" do
    field :title
    many_to_many :todo_items, MyApp.TodoItem, join_through: MyApp.TodoListItem
    timestamps()
  end
end

defmodule MyApp.TodoListItem do
  use Ecto.Schema

  schema "todo_list_items" do
    belongs_to :todo_list, MyApp.TodoList
    belongs_to :todo_item, MyApp.TodoItem
    timestamps()
  end
end

defmodule MyApp.TodoItem do
  use Ecto.Schema

  schema "todo_items" do
    field :description
    timestamps()
  end
end
```

Notice `MyApp.TodoList` no longer needs to define a `has_many` association pointing to the `MyApp.TodoListItem` schema and instead we can just associate to `:todo_items` using `many_to_many`.

Differently from `has_many :through`, `many_to_many` associations are also writeable. This means we can send data through our forms exactly as we did at the beginning of this chapter:

```
%{"todo_list" => %{
  "title" => "shipping list",
  "todo_items" => %{
    0 => %{"description" => "bread"},
    1 => %{"description" => "eggs"},
  }
}}
```

And we no longer need to define a changeset function in the intermediate schema:

```
# In MyApp.TodoList
def changeset(struct, params \\ %{}) do
  struct
  |> Ecto.Changeset.cast(params, [:title])
  |> Ecto.Changeset.cast_assoc(:todo_items, required: true)
end

# And then in MyApp.TodoItem
def changeset(struct, params \\ %{}) do
  struct
  |> Ecto.Changeset.cast(params, [:description])
end
```

In other words, we can use exactly the same code we had in the "todo lists has_many todo items" case. So even when external constraints require us to use a join table, `many_to_many` associations can automatically manage them for us. Everything you know about associations will just work with `many_to_many` associations, including the improvements we discussed in the previous chapter.

Finally, even though we have specified a schema as the `:join_through` option in `many_to_many`, `many_to_many` can also work without intermediate schemas altogether by simply giving it a table name:

```
defmodule MyApp.TodoList do
  use Ecto.Schema

  schema "todo_lists" do
    field :title
    many_to_many :todo_items, MyApp.TodoItem, join_through: "todo_list_items"
    timestamps()
  end
end
```

In this case, you can completely remove the `MyApp.TodoListItem` schema from your application and the code above will still work. The only difference is that when using tables, any autogenerated value that is filled by Ecto schema, such as timestamps, won't be filled

as we no longer have a schema. To solve this, you can either drop those fields from your migrations or set a default at the database level.

Summary

In this chapter we used `many_to_many` associations to drastically improve a polymorphic association design that relied on `has_many :through`. Our goal was to allow "todo_items" to associate to different entities in our code base, such as "todo_lists" and "projects". We have done this by creating intermediate tables and by using `many_to_many` associations to automatically manage those join tables.

At the end, our schemas may look like:

```
defmodule MyApp.TodoList do
  use Ecto.Schema

  schema "todo_lists" do
    field :title
    many_to_many :todo_items, MyApp.TodoItem, join_through: "todo_list_items"
    timestamps()
  end

  def changeset(struct, params \\ %{}) do
    struct
    |> Ecto.Changeset.cast(params, [:title])
    |> Ecto.Changeset.cast_assoc(:todo_items, required: true)
  end
end

defmodule MyApp.Project do
  use Ecto.Schema

  schema "todo_lists" do
    field :name
    many_to_many :todo_items, MyApp.TodoItem, join_through: "project_items"
    timestamps()
  end

  def changeset(struct, params \\ %{}) do
    struct
    |> Ecto.Changeset.cast(params, [:name])
    |> Ecto.Changeset.cast_assoc(:todo_items, required: true)
  end
end

defmodule MyApp.TodoItem do
  use Ecto.Schema

  schema "todo_items" do
    field :description
    timestamps()
  end

  def changeset(struct, params \\ %{}) do
    struct
    |> Ecto.Changeset.cast(params, [:description])
  end
end
```

And the database migration:

```
create table("todo_lists")  do
  add :title
  timestamps()
end

create table("projects")  do
  add :name
  timestamps()
end

create table("todo_items")  do
  add :description
  timestamps()
end

# Primary key and timestamps are not required if using many_to_many without schemas
create table("todo_lists_items", primary_key: false) do
  add :todo_item_id, references(:todo_items)
  add :todo_list_id, references(:todo_lists)
  # timestamps()
end

# Primary key and timestamps are not required if using many_to_many without schemas
create table("projects_items", primary_key: false) do
  add :todo_item_id, references(:todo_items)
  add :project_id, references(:projects)
  # timestamps()
end
```

Overall our code looks structurally the same as `has_many` would, although at the database level our relationships are expressed with join tables.

While in this chapter we changed our code to cope with the parameter format required by `cast_assoc`, in the next chapter we will drop `cast_assoc` altogether and use `put_assoc` which brings more flexibilities when working with associations.

Many to many and upserts

In the previous chapter we have learned about `many_to_many` associations and how to map external data to associated entries with the help of `Ecto.Changeset.cast_assoc/3`. While in the previous chapter we were able to follow the rules imposed by `cast_assoc/3`, doing so is not always possible nor desired.

In this chapter, we are going to look at `Ecto.Changeset.put_assoc/4` in contrast to `cast_assoc/3` and explore some examples. Finally, we will learn how to use the upsert feature from Ecto 2.1.

put_assoc vs cast_assoc

Imagine we are building an application that has blog posts and such posts may have many tags. Not only that, a given tag may also belong to many posts. This is a classic scenario where we would use `many_to_many` associations. Our migrations would look like:

```
create table(:posts) do
  add :title
  add :body
  timestamps()
end

create table(:tags) do
  add :name
  timestamps()
end

create unique_index(:tags, [:name])

create table(:posts_tags, primary_key: false) do
  add :post_id, references(:posts)
  add :tag_id, references(:tags)
end
```

Note we added a unique index to the tag name because we don't want to have duplicated tags in our database. It is important to add an index at the database level instead of using a validation since there is always a chance two tags with the same name would be validated and inserted simultaneously, passing the validation and leading to duplicated entries.

Now let's also imagine we want the user to input such tags as a list of words split by comma, such as: "elixir, erlang, ecto". Once this data is received in the server, we will break it apart into multiple tags and associate them to the post, creating any tag that does not yet exist in the database.

While the constraints above sound reasonable, that's exactly what put us in trouble with `cast_assoc/3`. Remember the `cast_assoc/3` changeset function was designed to receive external parameters and compare them with the associated data in our structs. To do so correctly, Ecto requires tags to be sent as a list of maps. However here we expect tags to be sent in a string separated by comma.

Furthermore, `cast_assoc/3` relies on the primary key field for each tag sent in order to decide if it should be inserted, updated or deleted. Again, because the user is simply passing a string, we don't have the ID information at hand.

When we can't cope with `cast_assoc/3`, it is time to use `put_assoc/4`. In `put_assoc/4`, we give Ecto structs or changesets instead of parameters, giving us the ability to manipulate the data as we want. Let's define the schema and the changeset function for a post which may receive tags as a string:

```
defmodule MyApp.Post do
  use Ecto.Schema

  schema "posts" do
    field :title
    field :body
    many_to_many :tags, MyApp.Tag, join_through: "posts_tags", on_replace: :delete
    timestamps()
  end

  def changeset(struct, params \\ %{}) do
    struct
    |> Ecto.Changeset.cast(params, [:title, :body])
    |> Ecto.Changeset.put_assoc(:tags, parse_tags(params))
  end

  defp parse_tags(params)  do
    (params["tags"] || "")
    |> String.split(",")
    |> Enum.map(&String.trim/1)
    |> Enum.reject(& &1 == "")
    |> Enum.map(&get_or_insert_tag/1)
  end

  defp get_or_insert_tag(name) do
    Repo.get_by(MyApp.Tag, name: name) ||
      Repo.insert!(MyApp.Tag, %Tag{name: name})
  end
end
```

In the changeset function above, we moved all the handling of tags to a separate function, called `parse_tags/1`, which checks for the parameter, breaks each tag apart via `String.split/2`, then removes any left over whitespace with `String.trim/1`, rejects any empty string and finally checks if the tag exists in the database or not, creating one in case none exists.

The `parse_tags/1` function is going to return a list of `MyApp.Tag` structs which are then passed to `put_assoc/3`. By calling `put_assoc/3`, we are telling Ecto those should be the tags associated to the post from now on. In case a previous tag was associated to the post and not given in `put_assoc/3`, Ecto will invoke the behaviour defined in the `:on_replace` option, which we have set to `:delete`. The `:delete` behaviour will remove the association between the post and the removed tag from the database.

And that's all we need to use `many_to_many` associations with `put_assoc/3`. `put_assoc/3` is very useful when we want to have more explicit control over our associations and it also works with `has_many`, `belongs_to` and all others association types.

However, our code is not yet ready for production. Let's see why.

Constraints and race conditions

Remember we added a unique index to the tag `:name` column when creating the tags table. We did so to protect us from having duplicate tags in the database.

By adding the unique index and then using `get_by` with a `insert!` to get or insert a tag, we introduced a potential error in our application. If two posts are submitted at the same time with a similar tag, there is a chance we will check if the tag exists at the same time, leading both submissions to believe there is no such tag in the database. When that happens, only one of the submissions will succeed while the other one will fail. That's a race condition: your code will error from time to time, only when certain conditions are met. And those conditions are time sensitive.

Many developers have a tendency to think such errors won't happen in practice or, if they happened, they would be irrelevant. But in practice they often lead to very frustrating user experiences. I have heard a first-hand example coming from a mobile game company. In the game, a player is able to play quests and on every quest you have to choose a guest character from another player out of a short list to go on the quest with you. At the end of the quest, you have the option to add the guest character as a friend.

Originally the whole guest list was random but, as time passed, players started to complain sometimes old accounts, often inactive, were being shown in the guests options list. To improve the situation, the game developers started to sort the guest list by most recently active. This means that, if you have just played recently, there is a higher chance of you to be on someone's guest lists.

However, when they did such change, many errors started to show up and users were suddenly furious in the game forum. That's because when they sorted players by activity, as soon as two players logged in, their characters would likely appear on each other's guest list. If those players picked each other's characters, the first to add the other as friend at the end of a quest would be able to succeed but an error would appear when the second player tried to add the other as a friend since the relationship already existed in the database! When that happened, all the progress done in the quest would be lost, because the server was unable to properly persist the quest results to the database. Understandably, players started to file complaints.

Long story short: we must address the race condition.

Luckily Ecto gives us a mechanism to handle constraint errors from the database.

Checking for constraint errors

Since our `get_or_insert_tag(name)` function fails when a tag already exists in the database, we need to handle such scenarios accordingly. Let's rewrite it taking race conditions into account:

```
defp get_or_insert_tag(name) do
  %Tag{}
  |> Ecto.Changeset.change(name: name)
  |> Ecto.Changeset.unique_constraint(:name)
  |> Repo.insert
  |> case do
    {:ok, tag} -> tag
    {:error, _} -> Repo.get_by!(MyApp.Tag, name: name)
  end
end
```

Instead of inserting the tag directly, we now build a changeset, which allows us to use the `unique_constraint` annotation. Now if the `Repo.insert` operation fails because the unique index for `:name` is violated, Ecto won't raise, but return an `{:error, changeset}` tuple. Therefore, if `Repo.insert` succeeds, it is because the tag was saved, otherwise the tag already exists, which we then fetch with `Repo.get_by!` .

While the mechanism above fixes the race condition, it is a quite expensive one: we need to perform two queries for every tag that already exists in the database: the (failed) insert and then the repository lookup. Given that's the most common scenario, we may want to rewrite it to the following:

```
defp get_or_insert_tag(name) do
  Repo.get_by(MyApp.Tag, name: name) || maybe_insert_tag(name)
end

defp maybe_insert_tag(name) do
  %Tag{}
  |> Ecto.Changeset.change(name: name)
  |> Ecto.Changeset.unique_constraint(:name)
  |> Repo.insert
  |> case do
    {:ok, tag} -> tag
    {:error, _} -> Repo.get_by!(MyApp.Tag, name: name)
  end
end
```

The above performs 1 query for every tag that already exists, 2 queries for every new tag and possibly 3 queries in the case of race conditions. While the above would perform slightly better on average, Ecto 2.1 has a better option in stock.

Upserts

Ecto 2.1 supports the so-called "upsert" command which is an abbreviation for "update or insert". The idea is that we try to insert a record and in case it conflicts with an existing entry, for example due to a unique index, we can choose how we want the database to act by either raising an error (the default behaviour), ignoring the insert (no error) or by updating the conflicting database entries.

"upsert" in Ecto 2.1 is done with the `:on_conflict` option. Let's rewrite `get_or_insert_tag(name)` once more but this time using the `:on_conflict` option. Remember that "upsert" is a new feature in PostgreSQL 9.5, so make sure you are up to date.

Your first try in using `:on_conflict` may be by setting it to `:nothing`, as below:

```
defp get_or_insert_tag(name) do
  Repo.insert!(%MyApp.Tag{name: name}, on_conflict: :nothing)
end
```

While the above won't raise an error in case of conflicts, it also won't update the struct given, so it will return a tag without ID. One solution is to force an update to happen in case of conflicts, even if the update is about setting the tag name to its current name. In such cases, PostgreSQL also requires the `:conflict_target` option to be given, which is the column (or a list of columns) we are expecting the conflict to happen:

```
defp get_or_insert_tag(name) do
  Repo.insert!(%MyApp.Tag{name: name},
               on_conflict: [set: [name: name]], conflict_target: :name)
end
```

And that's it! We try to insert a tag with the given name and if such tag already exists, we tell Ecto to update its name to the current value, updating the tag and fetching its id. While the above is certainly a step up from all solutions so far, it still performs one query per tag. If 10 tags are sent, we will perform 10 queries. Can we further improve this?

Upserts and insert_all

Ecto 2.1 did not only add the `:on_conflict` option to `Repo.insert/2` but also to the `Repo.insert_all/3` function introduced in Ecto 2.0. This means we can build one query that attempts to insert all missing tags and then another query that fetches all of them at once. Let's see how our `Post` schema will look like after those changes:

```
defmodule MyApp.Post do
  use Ecto.Schema

  # Schema is the same
  schema "posts" do
    add :title
    add :body
    many_to_many :tags, MyApp.Tag, join_through: "posts_tags", on_replace: :delete
    timestamps()
  end

  # Changeset is the same
  def changeset(struct, params \\ %{}) do
    struct
    |> Ecto.Changeset.cast(params, [:title, :body])
    |> Ecto.Changeset.put_assoc(:tags, parse_tags(params))
  end

  # Parse tags has slightly changed
  defp parse_tags(params)  do
    (params["tags"] || "")
    |> String.split(",")
    |> Enum.map(&String.trim/1)
    |> Enum.reject(& &1 == "")
    |> insert_and_get_all()
  end

  defp insert_and_get_all([]) do
    []
  end
  defp insert_and_get_all(names) do
    maps = Enum.map(names, &%{name: &1})
    Repo.insert_all MyApp.Tag, maps, on_conflict: :nothing
    Repo.all from t in MyApp.Tag, where: t.name in ^names
  end
end
```

Instead of attempting to get and insert each tag individually, the code above work on all tags at once, first by building a list of maps which is given to `insert_all` and then by looking up all tags with the existing names. Therefore, regardless of how many tags are sent, we will perform only 2 queries (unless no tag is sent, in which we return an empty list back

promptly). This solution is only possible in Ecto 2.1 thanks to the `:on_conflict` option, which guarantees `insert_all` won't fail in case a unique index is violated, such as duplicate tag names.

Finally, keep in mind that we haven't used transactions in any of the examples so far. That decision was deliberate as we relied on the fact that getting or inserting tags is an idempotent operation, i.e. we can repeat it many times for a given input and it will always give us the same result back. Therefore, even if we fail to introduce the post to the database due to a validation error, the user will be free to resubmit the form and we will just attempt to get or insert the same tags once again. The downside of this approach is that tags will be created even if creating the post fails, which means some tags may not have posts associated to them. In case that's not desired, the whole operation could be wrapped in a transaction or modeled with the `Ecto.Multi` abstraction we will discuss in the next chapter.

Composable transactions with Ecto.Multi

Ecto relies on database transactions when multiple operations must be performed atomically. Transactions can be performed via the `Repo.transaction` function:

```
Repo.transaction(fn ->
  mary_update = from Account, where: [id: ^mary.id], update: [inc: [balance: +10]]
  {1, _} = Repo.update_all(mary_update)
  john_update = from Account, where: [id: ^john.id], update: [inc: [balance: -10]]
  {1, _} = Repo.update_all(john_update)
end)
```

When we expect both operations to succeed, as above, transactions are quite straight-forward. However, transactions get more complicated if we need to check the status of each operation along the way:

```
Repo.transaction(fn ->
  mary_update = from Account, where: [id: ^mary.id], update: [inc: [balance: +10]]
  case Repo.update_all query do
    {1, _} ->
      john_update = from Account, where: [id: ^john.id], update: [inc: [balance: -10]]
      case Repo.update_all query do
        {1, _} ->
          {mary, john}
        {_, _} ->
          Repo.rollback({:failed_transfer, john})
      end
    {_, _} ->
      Repo.rollback({:failed_transfer, mary})
  end
end)
```

Transactions in Ecto can also be nested arbitrarily. For example, imagine the transaction above is moved into its own function that receives both accounts, defined as `transfer_money(mary, john, 10)`, and besides transferring money we also want to log the transfer:

```
Repo.transaction(fn ->
  case transfer_money(mary, john, 10) do
    {:ok, {mary, john}} ->
      Repo.insert!(%Transfer{from: mary.id, to: john.id, amount: 10})
    {:error, error} ->
      Repo.rollback(error)
  end
end)
```

The snippet above starts a transaction and then calls `transfer_money/3` that also runs in a transaction. This works because Ecto converts nested transaction into savepoints. In case an inner transaction fails, it rolls back to its specific savepoint.

While nesting transactions can improve the code readability by breaking large transactions into multiple smaller transactions, there is still a lot of boilerplate involved in handling the success and failure scenarios. Furthermore, composition is quite limited, as all operations must still be performed inside transaction blocks.

A more declarative approach when working with transactions would be to define all operations we want to perform in a transaction decoupled from the transaction execution. This way we would be able to compose transactions operations without worrying about its execution context or about each individual success/failure scenario. That's exactly what `Ecto.Multi` allows us to do.

Composing with data structures

Let's rewrite the snippets above using `Ecto.Multi`. The first snippet that transfers money between mary and john can rewritten to:

```
mary_update = from Account, where: [id: ^mary.id], update: [inc: [balance: +10]]
john_update = from Account, where: [id: ^john.id], update: [inc: [balance: -10]]
Ecto.Multi.new
|> Ecto.Multi.update_all(:mary, mary_update)
|> Ecto.Multi.update_all(:john, john_update)
```

`Ecto.Multi` is a data structure that defines multiple operations that must be performed together, without worrying about when they will be executed. `Ecto.Multi` mirrors most of the `Ecto.Repo` API, with the difference each operation must be explicitly named. In the example

above, we have defined two update operations, named `:mary` and `:john`. As we will see later, the names are important when handling the transaction results.

Since `Ecto.Multi` is just a data structure, we can pass it as argument to other functions, as well as return it. Assuming the multi above is moved into its own function, defined as `transfer_money(mary, john, value)`, we can add a new operation to the multi that logs the transfer as follows:

```
transfer_money(mary, john, 10)
|> Ecto.Multi.insert(:transfer, %Transfer{from: mary.id, to: john.id, amount: 10})
```

This is considerably simpler than the nested transaction approach we have seen earlier. Once all operations are defined in the multi, we can finally call `Repo.transaction`, this time passing the multi:

```
transfer_money(mary, john, 10)
|> Ecto.Multi.insert(:transfer, %Transfer{from: mary.id, to: john.id, amount: 10})
|> Repo.transaction()
|> case do
  {:ok, %{transfer: transfer}} ->
    # Handle success case
  {:error, name, value, rolled_back_changes} ->
    # Handle failure case
end
```

If all operations in the multi succeed, it returns `{:ok, map}` where the map contains the name of all operations as keys and their success value. If any operation in the multi fails, the transaction is rolled back and `Repo.transaction` returns `{:error, name, value, rolled_back_changes}`, where `name` is the name of the failed operation, `value` is the failure value and `rolled_back_changes` is a map of the previously successful multi operations that have been rolled back due to the failure.

In other words, `Ecto.Multi` takes care of all the flow control boilerplate while decoupling the transaction definition from its execution, allowing us to compose operations as needed.

Dependent values

Besides operations such as `insert`, `update` and `delete`, `Ecto.Multi` also provides functions for handling more complex scenarios. For example, `prepend` and `append` can be used to merge multis together. And more generally, the `Ecto.Multi.run/3` can be used to define any operation that depends on the results of a previous multi operation.

Let's study a more practical example by revisiting the problem defined in the previous chapter. Back then, we wanted to modify a post while possibly giving it a list of tags as a string separated by commas. At the end of the chapter, we built a solution that would insert any missing tag and then fetch all of them using only two queries:

```
defmodule MyApp.Post do
  use Ecto.Schema

  # Schema is the same
  schema "posts" do
    field :title
    field :body
    many_to_many :tags, MyApp.Tag, join_through: "posts_tags", on_replace: :delete
    timestamps()
  end

  # Changeset is the same
  def changeset(struct, params \\ %{}) do
    struct
    |> Ecto.Changeset.cast(params, [:title, :body])
    |> Ecto.Changeset.put_assoc(:tags, parse_tags(params))
  end

  # Parse tags has slightly changed
  defp parse_tags(params)  do
    (params["tags"] || "")
    |> String.split(",")
    |> Enum.map(&String.trim/1)
    |> Enum.reject(& &1 == "")
    |> insert_and_get_all()
  end

  defp insert_and_get_all([]) do
    []
  end
  defp insert_and_get_all(names) do
    maps = Enum.map(names, &%{name: &1})
    Repo.insert_all MyApp.Tag, maps, on_conflict: :nothing
    Repo.all from t in MyApp.Tag, where: t.name in ^names
  end
end
```

While `insert_and_get_all/1` is idempotent, allowing us to run it multiple times and get the same result back, it does not run inside a transaction, so any failure while attempting to modify the parent post struct would end-up creating tags that have no posts associated to them.

Let's fix the problem above by introducing using `Ecto.Multi`. Let's start by splitting the logic into both `Post` and `Tag` modules and keeping it free from side-effects such as database operations:

```
defmodule MyApp.Post do
  use Ecto.Schema

  schema "posts" do
    field :title
    field :body
    many_to_many :tags, MyApp.Tag, join_through: "posts_tags", on_replace: :delete
    timestamps()
  end

  def changeset(struct, tags, params) do
    struct
    |> Ecto.Changeset.cast(params, [:title, :body])
    |> Ecto.Changeset.put_assoc(:tags, tags)
  end
end

defmodule MyApp.Tag do
  use Ecto.Schema

  schema "tags" do
    field :name
    timestamps()
  end

  def parse(tags) do
    (tags || "")
    |> String.split(",")
    |> Enum.map(&String.trim/1)
    |> Enum.reject(& &1 == "")
  end
end
```

Now, whenever we need to introduce a post with tags, we can create a multi that wraps all operations and the repository access:

```
def insert_or_update_post_with_tags(post, params) do
  Ecto.Multi.new
  |> Ecto.Multi.run(:tags, &insert_and_get_all_tags(&1, params))
  |> Ecto.Multi.run(:post, &insert_or_update_post(&1, post, params)
  |> Repo.transaction()
end

defp insert_and_get_all_tags(_changes, params) do
  case MyApp.Tag.parse(params["tags"]) do
    [] ->
      {:ok, []}
    tags ->
      maps = Enum.map(names, &%{name: &1})
      Repo.insert_all(MyApp.Tag, maps, on_conflict: :nothing)
      {:ok, Repo.all(from t in MyApp.Tag, where: t.name in ^names)}
  end
end

defp insert_or_update_post(%{tags: tags}, post, params) do
  Repo.insert_or_update MyApp.Post.changeset(post, tags, params)
end
```

In the example above we have used `Ecto.Multi.run/3` twice, albeit for two different reasons.

1. In `Ecto.Multi.run(:tags, ...)`, we used `run/3` because we need to perform both `insert_all` and `all` operations, and while the multi exposes `Ecto.Multi.insert_all/4`, it does not yet expose a `Ecto.Multi.all/3`. Whenever we need to perform a repository operation that is not supported by `Ecto.Multi`, we can always fallback to `run/3`
2. In `Ecto.Multi.run(:post, ...)`, we used `run/3` because we need to access the value of a previous multi operation. The function given to `run/3` receives a map with the results of the operations performed so far. To grab the tags returned in the previous step, we simply pattern match on `%{tags: tags}` on `insert_or_update_post`

While `run/3` is very handy when we need to go beyond the functionalities provided natively by `Ecto.Multi`, it has the downside that operations defined with `Ecto.Multi.run/3` are opaque and therefore they cannot be inspected by functions such as `Ecto.Multi.to_list/1`. Still, `Ecto.Multi` allows us to greatly simplify control flow logic and remove boilerplate when working with transactions.

Concurrent tests with the SQL Sandbox

Our last chapter is about one of the most important features in Ecto 2.0: the concurrent SQL sandbox. Given Elixir's capability of using all of the machine resources available, the ability to run database dependent tests concurrently gives developers a low effort opportunity to speed up their test suite by 2x, 4x, 8x or more times, depending on the number of cores available.

Whenever you start an Ecto repository in your supervision tree, such as `supervisor(MyApp.Repo, [])`, Ecto starts a supervisor with a connection pool. The connection pool holds multiple open connections to the database. Whenever you want to perform a database operation, for example in a web request, Ecto automatically gets a connection from the pool, performs the operation you requested, and then puts the connection back in the pool.

This means that, when writing tests using Ecto's default connection pool (and not the SQL sandbox), each time you run a query, you will likely get a different connection from the pool. This is not good for tests since we want all operations in the same test to use the same connection.

Not only that, we also want data isolation between the tests. If I introduce a record to the "users" table in test A, test B should not see those entries when querying the same "users" table.

Ecto 1.0 solved the first problem by simply forcing tests to have only one connection in the database pool. Similarly, data isolation was addressed by simply not allowing tests to run concurrently. While it worked, it has the major downside of being unable to leverage all of our machine resources.

Explicit checkouts

Ecto 2.0 solves the problems above differently. The main idea is that we will allow the pool to have multiple connections but, instead of the connection being checked out implicitly every time we run a query, the connection must be explicitly checked out at the beginning of every test.

Once a connection is explicitly checked out, the test now owns that particular connection until the test is over. This way we guarantee that every time a connection is used in a test, it is always the same connection.

Let's start by setting up our database to the use `Ecto.Adapters.SQL.Sandbox` pool. You can set those options in your `config/config.exs` (or preferably `config/test.exs`):

```
config :my_app, Repo,
  pool: Ecto.Adapters.SQL.Sandbox
```

By default the sandbox pool starts in `:automatic` mode, which is exactly how Ecto works without the SQL sandbox pool: connections are checked out automatically as needed. This allows us to set up the database, for example by running migrations or in your `test/test_helper.exs`, as usual.

Before our tests start, we need to convert the pool to `:manual` mode, where each connection must be explicitly checked out. We do so by calling the `mode/2` function, typically at the end of the `test/test_helper.exs` file:

```
# At the end of your test_helper.exs
# Set the pool mode to manual for explicit checkouts
Ecto.Adapters.SQL.Sandbox.mode(MyApp.Repo, :manual)
```

If you simply add the line above and you do not change your tests to explicitly check a connection out from the pool, all of your tests will now fail. To solve this, you could explicitly check out the connection on each test but, to avoid repetition, let's define a `ExUnit.CaseTemplate` that automatically does so in `setup`:

```
defmodule MyApp.RepoCase do
  use ExUnit.CaseTemplate

  setup do
    # Explicitly get a connection before each test
    :ok = Ecto.Adapters.SQL.Sandbox.checkout(MyApp.Repo)
  end
end
```

Now in your tests, instead of `use ExUnit.Case`, you may write `use MyApp.RepoCase, async: true`. By following the steps above, we are now able to have multiple tests running concurrently, each owning a specific database transaction.

However, you may wonder, how does Ecto guarantees that the data generated in one test does not affect other tests?

Transactions

The second main insight in the SQL Sandbox is the idea of running each explicitly checked out connection inside a transaction. Every time you run `Ecto.Adapters.SQL.Sandbox.checkout(MyApp.Repo)` in a test, it does not only check out a connection but it also guarantees that connection has opened a transaction to the database. This way, any insert, update or delete you perform in your tests will be visible only to that test.

Furthermore, at the end of every test, we automatically rollback the transaction, effectively reverting all of the database changes you have performed in your tests. This guarantees a test won't affect tests running concurrently nor any test that may run afterwards.

While the approach of using multiple connections with transactions works in many cases, it also imposes some limitations depending on how database engines manage transactions and perform concurrency control. For example, while both PostgreSQL and MySQL support SQL Sandbox, only PostgreSQL supports concurrent tests with the SQL Sandbox. Therefore, do not use `async: true` with MySQL as you may run into deadlocks.

When using the concurrent SQL sandbox, there is also a chance of running into deadlocks when running tests with PostgreSQL with shared resources, such as database indexes. But those cases are well documented in the `Ecto.Adapters.SQL.Sandbox`, under the FAQ section.

Ownership

Whenever a test explicitly checks out a connection from the SQL Sandbox pool, we say the test process **owns** the connection. If a test, or any other process, does not own a connection, that test will error with a message describing it has no database connection.

Let's see an example:

```
use MyApp.RepoCase, async: true

test "create two posts, one sync, another async" do
  task = Task.async(fn ->
    Repo.insert!(%Post{title: "async"})
  end)
  assert %Post{} = Repo.insert!(%Post{title: "sync"})
  assert %Post{} = Task.await(task)
end
```

The test above will fail with an error similar to:

```
** (RuntimeError) cannot find ownership process for #PID<0.35.0>
```

Once we spawn a `Task`, there is no connection assigned to the task process, causing it to fail.

While most times we want different processes to have their own database connection, sometimes a test may need to interact with multiple processes that collaborate over the same data. Therefore, all of those processes must the same connection so they all belong to the same transaction.

The sandbox module provides two ways of doing so, via allowances or by running in shared mode.

Allowances

If a process explicitly owns a connection, that process may also *allow* other processes to use that connection, effectively allowing multiple processes to collaborate over the same connection at the same time. Let's give it a try:

```
test "create two posts, one sync, another async" do
  parent = self()
  task = Task.async(fn ->
    Ecto.Adapters.SQL.Sandbox.allow(Repo, parent, self())
    Repo.insert!(%Post{title: "async"})
  end)
  assert %Post{} = Repo.insert!(%Post{title: "sync"})
  assert %Post{} = Task.await(task)
end
```

And that's it! By calling `allow/3`, we are explicitly assigning the parent's connection (i.e. the test process' connection) to the task.

Because allowances use an explicit mechanism, their advantage is that you can still run your tests in async mode. The downside is that you need to explicitly control and allow every single process, which is not always possible. In such cases, you may resort to shared mode.

Shared mode

Shared mode allows a process to share its connection with any other process automatically, without relying on explicit allowances.

Let's change the example above to use shared mode:

```
test "create two posts, one sync, another async" do
  # Setting the shared mode must be done only after checkout
  Ecto.Adapters.SQL.Sandbox.mode(Repo, {:shared, self()})

  task = Task.async(fn ->
    Repo.insert!(%Post{title: "async"})
  end)
  assert %Post{} = Repo.insert!(%Post{title: "sync"})
  assert %Post{} = Task.await(task)
end
```

By calling `mode({:shared, self()})`, any process that needs to talk to the database will now use the same connection as the one checked out by the test process.

The advantage of shared mode is that by calling a single function, you will ensure all upcoming processes and operations will use that shared connection, without a need to explicitly allow them. The downside is that tests can not run concurrently in shared mode.

Summing up

In this chapter we have learned about the powerful concurrent SQL sandbox and how it leverages transactions and explicit checkouts to allow tests to run concurrently even when they need to communicate to the database.

We have also discussed two mechanism for a test to share ownerships:

- Using allowances - requires explicit allowances via `allow/3`. Tests may run concurrently.
- Using shared mode - does not require explicit allowances. Tests cannot run concurrently.

While throughout the book we covered how Ecto is a collection of tools for working on your domain, the last chapters also shows Ecto provides tools to better interact to the database regardless of your domain, such as Ecto.Multi which leverages the functional properties behind Elixir, as well as the SQL Sandbox which exploits the concurrency power behind the Erlang VM.

We hope you have learned a lot throughout this journey and that you are ready to write clean, performant and maintainable applications.