

The Little Ecto Cookbook

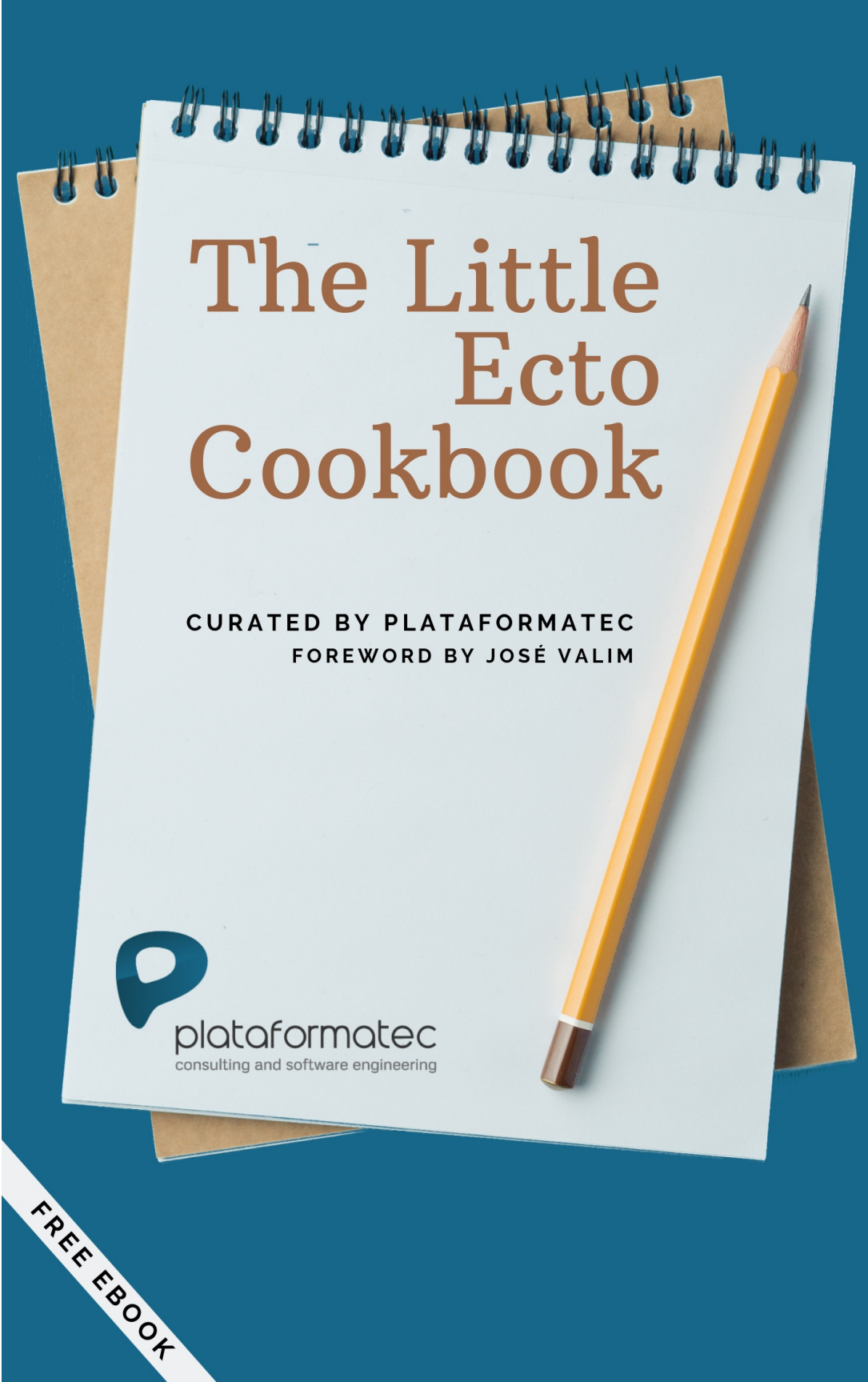
CURATED BY PLATAFORMATEC
FOREWORD BY JOSÉ VALIM



plataformatec
consulting and software engineering



FREE EBOOK



The Little Ecto Cookbook

CURATED BY PLATAFORMATEC
FOREWORD BY JOSÉ VALIM



plataformatec
consulting and software engineering

FREE EBOOK

Table of contents

- [Foreword](#)
- [Ecto is not your ORM](#)
- [Schemaless queries](#)
- [Data mapping and validation](#)
- [Dynamic queries](#)
- [Multi tenancy with query prefixes](#)
- [Aggregates and subqueries](#)
- [Test factories](#)
- [Constraints and Upserts](#)
- [Polymorphic associations with many to many](#)
- [Composable transactions with Multi](#)
- [Replicas and dynamic repositories](#)

Foreword

Ecto is one of oldest projects in the Elixir community. It started as a "Summer of Code" project back in 2013, lead by Eric Meadows-Jönsson, with myself (José Valim) as a mentor, and sponsored by Pragmatic Programmers and Interline Vacations.

At the time, Elixir was still being defined as a programming language. For instance, structs were not yet part of Elixir! And Ecto played a very important role for the development of Elixir itself. I will explain why.

When I started designing Elixir, I knew I would eventually use Elixir for building web applications and systems, as that's part of our daily job at Plataformatec, the company behind Elixir. At the same time, I didn't want Elixir to be a language focused on just Web programming. Quite the opposite, Elixir should be applicable to a wide variety of problems and domains. Therefore, **Elixir should be designed an extensible language**, and eventually Elixir would be extended to the Web domain. With this in mind, "Extensibility" became one of the three language goals, alongside "Productivity" and "Compatibility" (with the Erlang VM).

I spent months working on Elixir, with those goals in mind, but at some point we would need to test if Elixir was truly an extensible language. Ecto would eventually become one of those tests.

Ecto as a proof of concept

Ecto started as proof of concept that would touch many different aspects of Elixir as a programming language. That's because the amount of groundwork necessary to send a simple query to the database is enormous. For example, we need to write a driver to communicate with the database of choice (PostgreSQL), implement connection pooling, etc.

Still, working on Ecto was very exciting because it brought two interesting challenges:

1. Can we write a performant, safe, and readable query language? At the time, the benchmark was [LINQ from .NET](#). However, as the name says, LINQ was a Language Integrated Query, i.e. they had to change the host programming language (C#) to support it. Therefore, could Ecto implement something akin to LINQ, but without a need to change Elixir?
2. How does the existing literature of Object Oriented Enterprise Patterns, such as Active Record, Data Mapper, Table Records, etc, apply to a functional programming language? Is any pattern reusable? Or do we need to start everything from scratch?

For the first question, we got the query language surprisingly right. The query language we use today is still based on the one from our early drafts. We added more features with time, made the syntax less verbose and more dynamic, but the foundation is still the same and, more importantly, we didn't have to change Elixir itself to make Ecto Query possible!

When it comes to the second question though, we got a bunch of things right and a bunch of things wrong (although strangers on the internet are not always so pleasant when you get things wrong).

For example, Ecto v1.0 had models and life-cycle callbacks, in a poor attempt to emulate features found in patterns like Active Record. At the same time, when implementing other features commonly associated to ORMs (Object-Relational Mappers), such as dirty tracking, we were able to provide a lean and functional solution via [Ecto Changesets](#), which are well accepted to this day (the [original proposal](#) dates back to Jan 2015).

Ecto also hit the mark when it comes to relying on the database strengths, instead of using databases as dumb storage. Many applications that rely on ORMs that treat the database as dumb storage end up with corrupt and inconsistent data. Ecto, on the other hand, [knows how to walk the line between validations and constraints quite well](#). We will revisit the topic of ORMs as the first recipe in the book.

Nevertheless, Ecto v1.0 had enough weaknesses that eventually led to Ecto v2.0.

A brief history of major versions

Ecto v2.0 had two major goals: replace `Ecto.Model` by `Ecto.Schema` and tidy up how we interfaced with the different database adapters and drivers.

When Ecto v2.0 was released, Plataformatec also announced the "What's new in Ecto 2.0" ebook, which told readers how to migrate away from the "model mindset", common in other languages and frameworks, to a more functional approach. The ebook also covered many of the new Ecto features.

Eventually, the Ecto team started working on Ecto v3.0. The jump from Ecto v2.0 to Ecto v3.0 was much smaller than from Ecto v1.0 to Ecto v2.0. That's because Ecto v3.0 was mostly solidifying the choices done in Ecto v2.0.

For example, it turned out many developers were using the facilities in Ecto to work with data that would never touch the database, so we broke Ecto apart into Ecto and Ecto.SQL. Ecto v3.0 was also a good opportunity to remove outdated code and do further performance improvements. [We wrote a series of articles on those changes on the Plataformatec blog.](#)

After Ecto v3.0 was released, we also decided to open source the "What's new in Ecto 2.0" ebook as Ecto guides. This cookbook is a curation of the available Ecto guides into recipes. You can [find the complete list of guides online](#). If you find any typos or errors, [you can fix them directly at the source](#).

But perhaps, the most important news in Ecto v3.0 is that Ecto has finally become stable API. In other words, while we will continue releasing new Ecto versions with bug fixes, enhancements, and performance improvements, we don't have any plans for a new major version (v4.0). We know Ecto is not perfect (nothing is) but today it is clear what Ecto is and what Ecto isn't. If Ecto was never your cup of tea, now is a great time to explore different solutions, and build on top of the foundation of the database drivers built by the Ecto team and the Ecto community.

- José Valim, Creator of Elixir & Co-founder of Plataformatec

How to read this book

This book 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.

Acknowledgments

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

Finally we appreciate everyone who has reviewed the "What's new in Ecto 2.0" ebook, before it was made open source, 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 any case, for a book called "The little Ecto cookbook", this chapter does not look like a recipe at all. And you are right! In our defense, knowing "what we must not do" is sometimes as important as "knowing what to do". So consider this chapter our "don't put metals in the microwave" warning. :)

In the next two chapters, we will go back to the planned scheduled, and learn different recipes on how to break apart the "bad practices" above by exploring how to use Ecto without schemas or even with 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:

```
query =
  from p in Post,
    select: %Post{title: p.title, body: p.body, ...}

MyApp.Repo.all(query)
```

Although you might use schemas for most of your queries, Ecto also adds the ability to write regular schemaless queries when preferred.

One example is this ability to select all desired fields without duplication:

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

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 is available to almost all query constructs. 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]]
end
```

```
MyApp.Repo.update_all(query)
end
```

The [Ecto.Query.update/3](#) 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
```

Let's take a look at another example. Imagine 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)
  query =
    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)
    }

    MyApp.Repo.all(query)
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.

By allowing regular data structures to be given to most query operations, Ecto 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.

insert_all, update_all and delete_all

Ecto allows all database operations to be expressed without a schema. One of the functions provided 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"]
  ]
)

```

Updates and deletes can also be done without schemas via [Ecto.Repo.update_all/3](#) and [Ecto.Repo.delete_all/2](#) respectively:

```

# Use the ID to trigger updates
post = from p in "posts", where: [id: ^id]

# Update the title for all matching posts
{1, _} =

```



```
MyApp.Repo.update_all post, set: [title: "new title"]

# Delete all matching posts
{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.

Data mapping and validation

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.Schema](#) moduledoc 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
```

```
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 from changeset
  registration = Ecto.Changeset.apply_changes(changeset)
  account = Registration.to_account(registration)
  profile = Registration.to_profile(registration)

  MyApp.Repo.transaction fn ->
    MyApp.Repo.insert_all "accounts", [account]
    MyApp.Repo.insert_all "profiles", [profile]
  end

  {:ok, registration}
else
  # Annotate the action 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 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 = %{name: :string, email: :string}

# The data+types tuple is equivalent to %Registration{}
changeset =
  {data, types}
  |> 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 guide 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.

Another problem with the pre-compiled query syntax is that it has limited options to compose the 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.

To solve those problems, Ecto also provides a data-structure centric API to build queries as well as a very powerful mechanism for dynamic queries. Let's take a look.

Focusing on data structures

Ecto provides a simpler API for both keyword and pipe based queries by making data structures first-class. Let's see an example:

```
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, 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)
```

While using data-structures already brings a good amount of flexibility to Ecto queries, not all expressions can be converted to data structures. For example, `where` converts a key-value to a `key == value` comparison, and therefore order-based comparisons such as `p.published_at > ^minimum_date` need to be written as before.

Dynamic fragments

For cases where we cannot rely on data structures but still desire to build queries dynamically, Ecto includes the [Ecto.Query.dynamic/2](#) macro.

The `dynamic` macro allows us to conditionally build query fragments and interpolate them in the main query. For example, imagine that in the example above you may optionally filter posts by a date of publication. You could of course write it like this:

```
query =
  Post
  |> where(^where)
  |> order_by(^order_by)

query =
  if published_at = params["published_at"] do
    where(query, [p], p.published_at < ^published_at)
  else
    query
  end
```

But with dynamic fragments, you can also write it as:

```
where = [author: "José", category: "Elixir"]
order_by = [desc: :published_at]

filter_published_at =
  if published_at = params["published_at"] do
    dynamic([p], p.published_at < ^published_at)
  else
    true
  end

Post
|> where(^where)
|> where(^filter_published_at)
|> order_by(^order_by)
```

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.

By using dynamic fragments, we can decouple the processing of parameters from the query generation. Let's see a more complex example.

Building dynamic queries

Let's go back to the original problem. We want to build a search functionality where the user can configure how to traverse all posts in many different ways. For example, the user may choose how to order the data, filter by author and category, as well as select posts published after a certain date.

To tackle this in Ecto, we can break our problem into a bunch of small functions, that build either data structures or dynamic fragments, and then we interpolate it into the query:

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

def filter_order_by("published_at_desc"),
  do: dynamic([p], desc: p.published_at)

def filter_order_by("published_at"),
  do: dynamic([p], p.published_at)

def filter_order_by(_),
  do: []

def filter_where(params) do
  Enum.reduce(params, dynamic(true), fn
    {"author", value}, dynamic ->
      dynamic([p], ^dynamic and p.author == ^value)

    {"category", value}, dynamic ->
      dynamic([p], ^dynamic and p.category == ^value)

    {"published_at", value}, dynamic ->
      dynamic([p], ^dynamic and p.published_at > ^value)

    {_, _}, dynamic ->
      # Not a where parameter
      dynamic
  end)
end
```

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 use `reduce` to start with an empty dynamic (that always returns true) and refine it with new conditions as we traverse the parameters.

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 dynamic_match?(
    filter_where(%{}),
    "true"
  )

  assert dynamic_match?(
    filter_where(%{"published_at" => "2010-04-17"}),
    "true and q.published_at > ^\"2010-04-17\""
  )
end

defp dynamic_match?(dynamic, string) do
  inspect(dynamic) == "dynamic([q], #{string})"
end
```

In the example above, we created a small helper that allows us to assert on the dynamic contents by matching on the results of `inspect(dynamic)`.

Dynamic and joins

Even query joins can be tackled dynamically. For example, let's do two modifications to the example above. Let's say we can also sort by author name ("author_name" and "author_name_desc") and at the same time let's say that authors are in a separate table, which means our authors filter in `filter_where` now need to go through the join table.

Our final solution would look like this:

```

def filter(params) do
  Post
  # 1. Add named join binding
  |> join([p], assoc(p, :authors), as: :authors)
  |> order_by(^filter_order_by(params["order_by"]))
  |> where(^filter_where(params))
end

# 2. Returned dynamic with join binding
def filter_order_by("published_at_desc"),
  do: dynamic([p], desc: p.published_at)

def filter_order_by("published_at"),
  do: dynamic([p], p.published_at)

def filter_order_by("author_name_desc"),
  do: dynamic([authors: a], desc: a.name)

def filter_order_by("author_name"),
  do: dynamic([authors: a], a.name)

def filter_order_by(_),
  do: []

# 3. Change the authors clause inside reduce
def filter_where(params) do
  Enum.reduce(params, dynamic(true), fn
    {"author", value}, dynamic ->
      dynamic([authors: a], ^dynamic and a.name == ^value)

    {"category", value}, dynamic ->
      dynamic([p], ^dynamic and p.category == ^value)

    {"published_at", value}, dynamic ->
      dynamic([p], ^dynamic and p.published_at > ^value)

    {_, _}, dynamic ->
      # Not a where parameter
      dynamic
  end)
end

```

Adding more filters in the future is simply a matter of adding more clauses to the `Enum.reduce/3` call in `filter_where`.

Multi tenancy with query prefixes

With Ecto we can 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 guide. All the examples below assume you are using Postgres. Other databases engines may require slightly different solutions.

Connection 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,
    adapter: Ecto.Adapters.Postgres
```

```
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,
  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 - press Ctrl+C to exit
```

```
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 "connection_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:

```
query_args = ["SET search_path TO connection_prefix", []]  
  
config :my_app, MyApp.Repo,  
  username: "postgres",  
  password: "postgres",  
  database: "demo_dev",  
  hostname: "localhost",  
  pool_size: 10,  
  after_connect: {Postgrex, :query!, query_args}
```

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

```
$ iex -S mix  
Interactive Elixir - press Ctrl+C to exit  
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 connection_prefix"
```

Now we are ready to migrate and run our queries:

```
$ mix ecto.migrate
$ iex -S mix
Interactive Elixir - press Ctrl+C to exit
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.

Schema prefixes

Ecto also allows you 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 are 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 for the connection 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.

Per-query and per-struct prefixes

Now, suppose that while still configured to connect to the "connection_prefix" on `:after_connect`, we run the following queries:

```

iex(1) alias MyApp.Sample
MyApp.Sample
iex(2) MyApp.Repo.all Sample
[]
iex(3) MyApp.Repo.insert %Sample{name: "mary"}
{:ok, %MyApp.Sample{...}}
iex(4) MyApp.Repo.all Sample
[%MyApp.Sample{...}]

```

The operations above ran on the "connection_prefix". So 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(5)> query = Ecto.Queryable.to_query Sample
#Ecto.Query<from s in MyApp.Sample>
iex(6)> 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 also supports the `:prefix` option on all relevant repository operations:

```
iex(7)> MyApp.Repo.all Sample
[%MyApp.Sample{...}]
iex(8)> MyApp.Repo.all 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(9) [sample] = MyApp.Repo.all Sample
[%MyApp.Sample{}]
iex(10)> 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, "connection_prefix" will be used because of the `:after_connect` callback we added at the beginning of this guide.

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 "connection_prefix" to the "public" one:

```
iex(11)> new_sample = Ecto.put_meta(sample, prefix: "public")
%MyApp.Sample{}
iex(12)> MyApp.Repo.insert new_sample
{:ok, %MyApp.Sample{}}
iex(13)> [sample] = MyApp.Repo.all Sample, prefix: "public"
[%MyApp.Sample{}]
iex(14)> 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**.

Per from/join prefixes

Finally, Ecto allows you to set the prefix individually for each `from` and `join` expression. Here's an example:

```
from p in Post, prefix: "foo",  
  join: c in Comment, prefix: "bar"
```

Those will take precedence over all other prefixes we have defined so far. For each join/from in the query, the prefix used will be determined by the following order:

1. If the prefix option is given exclusively to join/from
2. If the `@schema_prefix` is set in the related schema
3. If the `:prefix` field is set on the query (i.e. `%{query | prefix: prefix}`) or to the repo operation (i.e. `Repo.all query, prefix: prefix`)
4. The connection prefix

Migration prefixes

When the connection 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
```

Summing up

Ecto provides many conveniences for working with querying prefixes. Those conveniences allow developers to configure prefixes with different precedence, starting with the highest one:

1. from/join prefixes
2. query/struct prefixes

3. schema prefixes
4. connection prefixes

This way developers can tackle different scenarios, from production requirements to multi-tenant applications.

Aggregates and subqueries

Now it's time to discuss aggregates and subqueries. As we will learn, one builds directly on the other.

Aggregates

Ecto 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 [Ecto.Repo.aggregate/4](#) function supports any of the aggregate operations listed in the [Ecto.Query.API](#) module.

At first, it looks like the implementation of `aggregate/4` is quite straightforward. 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/4` 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/4` automatically wraps the given query in a subquery. Therefore the query executed by `aggregate/4` above is rather equivalent to:

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

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, 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:

```
inner_query =
  from MyApp.Post,
    order_by: [desc: :visits],
    limit: 10

query =
  from q in subquery(inner_query),
    select: avg(q.visits)

MyApp.Repo.one(query)
```

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 also allows 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}
```

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 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
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.

Constraints and Upserts

In this guide we will learn how to use constraints and upserts. To showcase those features, we will work on a practical scenario: which is by studying a many to many relationship between posts and tags.

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`. 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. We can see an example of this in [Polymorphic associations with many to many](#). 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
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/4`. By calling `put_assoc/4`, 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/4`, 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/4`. `put_assoc/4` 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.

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 has a better option in stock.

Upserts

Ecto 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 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 accepts the `:on_conflict` option not only in [Ecto.Repo.insert/2](#) but also in the [Ecto.Repo.insert_all/3](#) function. 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 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`.

Polymorphic associations with many to many

Besides `belongs_to`, `has_many`, `has_one` and `:through` associations, Ecto 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 guide, 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])
end

```

```
|> 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 currently 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, such approach is not always preferable 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, especially if you're working with large tables. Bear in mind that if that's your case, you might be forced to remove such polymorphic references in the future when frequent polymorphic queries start grinding the database to a halt even after adding indexes and optimizing the database.

Luckily, the documentation for the [Ecto.Schema.belongs_to/3](#) macro includes a section named "Polymorphic associations" with some 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
  end
end

```

```

    timestamps()
  end
end

defmodule MyApp.TodoItem do
  use Ecto.Schema

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

```

Although we introduced `MyApp.TodoList` 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

```



```

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 guide:

```

%{"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 as well.

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 guide 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
end
```

```

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 guide we changed our code to cope with the parameter format required by `cast_assoc`, in [Constraints and Upserts](#) we drop `cast_assoc` altogether and use `put_assoc` which brings more flexibilities when working with associations.

Composable transactions with Multi

Ecto relies on database transactions when multiple operations must be performed atomically. The most common example used for transaction are bank transfers between two people:

```
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)
```

In Ecto, transactions can be performed via the `Repo.transaction` function. 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}} ->
      transfer = %Transfer{
        from: mary.id,
        to: john.id,
        amount: 10
      }

      Repo.insert!(transfer)

    {: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. In case of multiple transactions, they are all flattened, which means a failure in an inner transaction causes the outer transaction to also fail. That's why matching and rolling back on `{:error, error}` is important.

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 be 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 = %Transfer{
  from: mary.id,
```

```

    to: john.id,
    amount: 10
  }

  transfer_money(mary, john, 10)
  |> Ecto.Multi.insert(:transfer, transfer)

```

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 = %Transfer{
  from: mary.id,
  to: john.id,
  amount: 10
}

transfer_money(mary, john, 10)
|> Ecto.Multi.insert(:transfer, transfer)
|> Repo.transaction()
|> case do
  {:ok, %{transfer: transfer}} ->
    # Handle success case
  {:error, name, value, changes_so_far} ->
    # 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, changes_so_far}`, where `name` is the name of the failed operation, `value` is the failure value and `changes_so_far` 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` and `Ecto.Multi.run/5` can be used to define any operation that depends on the results of a previous multi operation.

Let's study a more practical example. In [Constraints and Upserts](#), we want to modify a post while possibly giving it a list of tags as a string separated by commas. At the end of the guide, we present a solution that 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
    []
  end
end
```

```

    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
end

```

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

```

alias MyApp.Tag

def insert_or_update_post_with_tags(post, params) do
  Ecto.Multi.new
  |> Ecto.Multi.run(:tags, fn _, changes ->
    insert_and_get_all_tags(changes, params)
  end)
  |> Ecto.Multi.run(:post, fn _, changes ->
    insert_or_update_post(changes, post, params)
  end)
  |> 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(Tag, maps, on_conflict: :nothing)
      query = from t in Tag, where: t.name in ^names
      {:ok, Repo.all(query)}
  end
end

defp insert_or_update_post(%{tags: tags}, post, params) do
  post = MyApp.Post.changeset(post, tags, params)
  Repo.insert_or_update post
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` or `run/5`.
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, as second argument, 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`.

Note: The first argument received by the function given to `run/3` is the repo in which the transaction is executing.

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.

Replicas and dynamic repositories

When applications reach a certain scale, a single database may not be enough to sustain the required throughput. In such scenarios, it is very common to introduce read replicas: all write operations are sent to primary database and most of the read operations are performed against the replicas. The credentials of the primary and replica databases are typically known upfront by the time the code is compiled.

In other cases, you may need a single Ecto repository to interact with different database instances which are not known upfront. For instance, you may need to communicate with hundreds of database very sporadically, so instead of opening up a connection to each of those hundreds of database when your application starts, you want to quickly start connection, perform some queries, and then shut down, while still leveraging Ecto's APIs as a whole.

This guide will cover how to tackle both approaches.

Primary and Replicas

Since the credentials of the primary and replicas databases are known upfront, adding support for primary and replica databases in your Ecto application is relatively straightforward. Imagine you have a `MyApp.Repo` and you want to add four read replicas. This could be done in three steps.

First, define the primary and replicas repositories in `lib/my_app/repo.ex`:

```
defmodule MyApp.Repo do
  use Ecto.Repo,
    otp_app: :my_app,
    adapter: Ecto.Adapters.Postgres

  @replicas [
    MyApp.Repo.Replica1,
    MyApp.Repo.Replica2,
    MyApp.Repo.Replica3,
    MyApp.Repo.Replica4
  ]
end
```

```

]

def replica do
  Enum.random(@replicas)
end

for repo <- @replicas do
  defmodule repo do
    use Ecto.Repo,
      otp_app: :my_app,
      adapter: Ecto.Adapters.Postgres,
      read_only: true
  end
end
end

```

The code above defines a regular `MyApp.Repo` and four replicas, called `MyApp.Repo.Replica1` up to `MyApp.Repo.Replica4`. We pass the `:read_only` option to the replica repositories, so operations such as `insert`, `update` and `friends` are not made accessible. We also define a function called `replica` with the purpose of returning a random replica.

Next we need to make sure both primary and replicas are configured properly in your `config/config.exs` files. In development and test, you can likely use the same database credentials for all repositories, all pointing to the same database address:

```

replicas = [
  MyApp.Repo,
  MyApp.Repo.Replica1,
  MyApp.Repo.Replica2,
  MyApp.Repo.Replica3,
  MyApp.Repo.Replica4
]

for repo <- replicas do
  config :my_app, repo,
    username: "postgres",
    password: "postgres",
    database: "my_app_prod",
    hostname: "localhost",
    pool_size: 10
end

```


In production, you want each database to connect to a different hostname:

```
repos = %{\n  MyApp.Repo => "prod-primary",\n  MyApp.Repo.Replica1 => "prod-replica-1",\n  MyApp.Repo.Replica2 => "prod-replica-2",\n  MyApp.Repo.Replica3 => "prod-replica-3",\n  MyApp.Repo.Replica4 => "prod-replica-4"\n}\n\nfor {repo, hostname} <- repos do\n  config :my_app, repo,\n    username: "postgres",\n    password: "postgres",\n    database: "my_app_prod",\n    hostname: hostname,\n    pool_size: 10\nend
```

Finally, make sure to start all repositories in your supervision tree:

```
children = [\n  MyApp.Repo,\n  MyApp.Repo.Replica1,\n  MyApp.Repo.Replica2,\n  MyApp.Repo.Replica3,\n  MyApp.Repo.Replica4\n]
```

Now that all repositories are configured, we can safely use them in your application code. Every time you are performing a read operation, you can the `replica/0` function that we have added to return a random replica we will send the query to:

```
MyApp.Repo.replica().all(query)
```

And now you are ready to work with primary and replicas, no hacks or complex dependencies required!

Testing replicas

While all of the work we have done so far should fully work in development and production, it may not be enough for tests. Most developers testing Ecto applications are using a sandbox, such as the [Ecto SQL Sandbox](#).

When using a sandbox, each of your tests run in an isolated and independent transaction. Once the test is done, the transaction is rolled back. Which means we can trivially revert all of the changes done in a test in a very performant way.

Unfortunately, even if you configure your primary and replicas to have the same credentials and point to the same hostname, each Ecto repository will open up their own pool of database connections. This means that, once you move to a primary + replicas setup, a simple test like this one won't pass:

```
user = Repo.insert!(%User{name: "jane doe"})
assert Repo.replica().get!(User, user.id)
```

That's because `Repo.insert!` will write to one database connection and the repository returned by `Repo.replica()` will perform the read in another connection. Since the write is done in transaction, its contents won't be available to other connections until the transaction commits, which will never happen for test connections.

There are two options to tackle this problem: one is to change replicas and the other is to use dynamic repos.

A custom `replica` definition

One simple solution to the problem above is to use a custom `replica` implementation during tests that always return the primary repository, like this:

```
if Mix.env() == :test do
  def replica, do: __MODULE__
else
  def replica, do: Enum.random(@replicas)
end
```

Now during tests, the replica will always return the repository primary repository itself. While this approach works fine, it has the downside that, if you accidentally invoke a write function in a replica, the test will pass, since the `replica` function is returning the primary repo, while the code will fail in production.

Using `:default_dynamic_repo`

Another approach to testing is to set the `:default_dynamic_repo` option when defining the repository. Let's see what we mean by that.

When you list a repository in your supervision tree, such as `MyApp.Repo`, behind the scenes it will start a supervision tree with a process named `MyApp.Repo`. By default, the process has the same name as the repository module itself. Now every time you invoke a function in `MyApp.Repo`, such as `MyApp.Repo.insert/2`, Ecto will use the connection pool from the process named `MyApp.Repo`.

From v3.0, Ecto has the ability to start multiple processes from the same repository. The only requirement is that they must have different process names, like this:

```
children = [
  MyApp.Repo,
  {MyApp.Repo, name: :another_instance_of_repo}
]
```

While the particular example doesn't make much sense (we will cover an actual use case for this feature next), the idea is that now you have two repositories running: one is named `MyApp.Repo` and the other one is named `:another_instance_of_repo`. Each of those processes have their own connection pool. You can tell Ecto which process you want to use in your repo operations by calling:

```
MyApp.Repo.put_dynamic_repo(MyApp.Repo)
MyApp.Repo.put_dynamic_repo(:another_instance_of_repo)
```

Once you call `MyApp.Repo.put_dynamic_repo(name)`, all invocations made on `MyApp.Repo` will use the connection pool denoted by `name`.

How can this help with our replica tests? If we look back to the supervision tree we defined earlier in this guide, you will find this:

```
children = [  
  MyApp.Repo,  
  MyApp.Repo.Replica1,  
  MyApp.Repo.Replica2,  
  MyApp.Repo.Replica3,  
  MyApp.Repo.Replica4  
]
```

We are starting five different repositories and five different connection pools. Since we want the replica repositories to use the `MyApp.Repo`, we can achieve this by doing the following on the setup of each test:

```
@replicas [  
  MyApp.Repo.Replica1,  
  MyApp.Repo.Replica2,  
  MyApp.Repo.Replica3,  
  MyApp.Repo.Replica4  
]  
  
setup do  
  for replica <- @replicas do  
    replica.put_dynamic_repo(MyApp.Repo)  
  end  
  
  :ok  
end
```

There is even a better way! We can pass a `:default_dynamic_repo` option when we define the repository. In this case, we want to set the `:default_dynamic_repo` to `MyApp.Repo` only during the test environment. In your `lib/my_app/repo.ex`, do this:

```
for repo <- @replicas do  
  dynamic_default_repo =
```

```

    if Mix.env() == :test do
      MyApp.Repo
    else
      repo
    end

    defmodule repo do
      use Ecto.Repo,
        otp_app: :my_app,
        adapter: Ecto.Adapters.Postgres,
        read_only: true,
        dynamic_default_repo: dynamic_default_repo
    end
  end
end

```

And now your tests should work as before, while still being able to detect if you accidentally perform a write operation in a replica.

Dynamic repositories

At this point, we have learned that Ecto allows you to start multiple connections based on the same repository. This is typically useful when you have to connect multiple databases or perform short-lived database connections.

For example, you can start a repository with a given set of credentials dynamically, like this:

```

MyApp.Repo.start_link(
  name: :some_client,
  hostname: "client.example.com",
  username: "...",
  password: "...",
  pool_size: 1
)

```

In other words, `start_link` accepts the same options as the database configuration. Now let's do a query on the dynamically started repository. If you attempt to simply perform `MyApp.Repo.all(Post)`, it may fail, as by default it will try to use a process named `MyApp.Repo`, which may or may not be running. So don't forget to call `put_dynamic_repo/1` before:

```
MyApp.Repo.put_dynamic_repo(:some_client)
MyApp.Repo.all(Post)
```

Ecto also allows you to start a repository with no name (just like that famous horse). In such cases, you need to explicitly pass `name: nil` and match on the result of `MyApp.Repo.start_link/1` to retrieve the PID, which should be given to `put_dynamic_repo`. Let's also use this opportunity and perform proper database clean-up, by shutting up the new repository and reverting the value of `put_dynamic_repo`:

```
default_dynamic_repo = MyApp.Repo.get_dynamic_repo()

{:ok, repo} =
  MyApp.Repo.start_link(
    name: nil,
    hostname: "client.example.com",
    username: "...",
    password: "...",
    pool_size: 1
  )

try do
  MyApp.Repo.put_dynamic_repo(repo)
  MyApp.Repo.all(Post)
after
  MyApp.Repo.put_dynamic_repo(default_dynamic_repo)
  MyApp.Repo.stop(repo)
end
```

We can encapsulate all of this in a function too, which you could define in your repository:

```
defmodule MyApp.Repo do
  use Ecto.Repo, ...

  def with_dynamic_repo(credentials, callback) do
    default_dynamic_repo = get_dynamic_repo()
    start_opts = [name: nil, pool_size: 1] ++ credentials
    {:ok, repo} = MyApp.Repo.start_link(start_opts)

    try do
```

```
    MyApp.Repo.put_dynamic_repo(repo)
    callback.()
  after
    MyApp.Repo.put_dynamic_repo(default_dynamic_repo)
    MyApp.Repo.stop(repo)
  end
end
end
end
```

And now use it as:

```
credentials = [
  hostname: "client.example.com",
  username: "...",
  password: "..."
]

MyApp.Repo.with_dynamic_repo(credentials, fn ->
  MyApp.Repo.all(Post)
end)
```

And that's it! Now you can have dynamic connections, all properly encapsulated in a single function and built on top of the dynamic repo API.