Extracted from:

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

Chris McCord,
Bruce Tate,
and José Valim

edited by Jacquelyn Carter

Programming Phoenix ≥ 1.4

Productive |> Reliable |> Fast

Chris McCord
Bruce Tate
José Valim

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Jacquelyn Carter
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-226-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2019

CHAPTER 7

Ecto Queries and Constraints

In the previous chapter, we extended our application domain by associating videos to users and categories. Now we want our users to select which category a video belongs to upon video creation. To build this feature, you'll need to learn how to programmatically populate the database with a hardcoded list of categories and add those new features to our context. Along the way we'll explore some of the different ways you can use Ecto to retrieve data from the database.

We want to build our feature safely so that corrupt data can't creep into our database, so we'll spend some time working with database constraints. Database engines like PostgreSQL are called *relational* for a reason. A tremendous amount of time and effort has gone into tools and features that help developers define and enforce the relationships between tables. Instead of treating the database as pure dumb storage, Ecto uses the strengths of the database to help keep the data consistent. You'll learn about error-reporting strategies so you'll know when to report an error and when to let it crash, letting other application layers handle the problem.

Let's get started.

Seeding and Associating Categories

Let's associate videos and categories. The first step is to make sure categories actually exist in our database by using seed data. Then we will change our web interface to allow users to pick the category for a new video.

Setting Up Category Seed Data

We need to define a handful of initial categories for our application to use. We could start an IEx session and directly invoke the repository to do that, but this approach has some issues. If we do this work manually, each teammate will have to do the same as soon as they want to take our application for a spin. Once our application grows in size, having to populate each table in our application with relevant data can get long and tedious.

Furthermore, categories won't have a web interface where we can manage them, so we need a mechanism to create them in production programatically. Elixir is a great language for writing scripts so let's create a small one to insert data in the database. We'll let that new script use a function in our Multimedia context to create the necessary records.

Phoenix already defines a convention for seeding data. Open up priv/repo/seeds.exs and read the comments Phoenix generated for us. Phoenix will make sure that our database is appropriately populated. We only need to drop in a script that uses our repository to directly add the data we want. Then, we'll be able to run Mix commands when it's time to create the data.

Since the seed script may be executed multiple times, namely every time more seed data is added, we need to make sure our seed script won't fail or won't generate duplicated categories every time it runs.

Let's see what happens when we create a category that already exists. Open up IEx and key this in:

```
iex> Rumbl.Repo.insert! %Rumbl.Multimedia.Category{name: "Test"}
%Rumbl.Multimedia.Category{
  __meta__: #Ecto.Schema.Metadata<:loaded, "categories">,
  id: 1,
  inserted_at: ~N[2019-05-19 13:06:12],
  name: "hello",
  updated_at: ~N[2019-05-19 13:06:12]
}
```

So far, so good. We used the insert! repository function, which will raise an error if anything goes wrong. Let's run the same command again:

```
iex> Rumbl.Repo.insert! %Rumbl.Multimedia.Category{name: "Test"}
** (Ecto.ConstraintError) constraint error when attempting to insert struct:

    * categories_name_index (unique_constraint)

...

The changeset has not defined any constraint.
```

Now Ecto has raised an a ConstraintError, letting us know that the unique_constraint defined in our database did not allow the operation to succeed. Ecto also tells us how to convert this constraint error into a changeset error, a technique we will employ later in this chapter.

However, in this particular case, instead of returning errors as part of a changeset, we would rather create the category only if it doesn't exist. Perhaps, we could write this operation as:

```
Repo.get_by(Category, name: name) || Repo.insert!(%Category{name: name})
```

While this behavior would likely be fine for our seed scripts, this idiom is inherently unsafe, and we should generally avoid it. For instance, if two users are trying to create a new category with the same name at the same time, the Repo.get_by(Category, name: name) would return nil to both, causing both of them to insert the same category. Thanks to our uniqueness constraint, only one of those operations will succeed and we will not get duplicate categories, but the other user would get an error page, leading to a poor user experience.

The answer to the problem is once more to let the database manage the data integrity. In particular, we want to let the database manage what happens when there is a conflict with the data we are inserting. This feature is commonly known as an "Upsert" because it is common to update the data whenever there is a conflict during an insert. In this case, we want to simply ignore the conflict.

Ecto allows us to do exactly that via the :on_conflict option:

```
iex> Rumbl.Repo.insert! %Rumbl.Multimedia.Category{name: "hello"},
...>   on_conflict: :nothing
%Rumbl.Multimedia.Category{
  __meta__: #Ecto.Schema.Metadata<:loaded, "categories">,
  id: nil,
  inserted_at: ~N[2019-05-19 13:06:22],
  name: "hello",
  updated_at: ~N[2019-05-19 13:06:22]
}
```

The default value for :on_conflict is :raise. Once we change it to :nothing, no exceptions are raised and you can see the returned category has a nil id, indicating that indeed the category was not inserted. Upserts allow us to do many different things in case of conflicts, from updating certain fields to even performing whole queries. The downside is that the upsert behaviour is often database specific, so make sure to explore the different options available to your database of choice. You can learn more about upserts in the documention for Repo.insert.[1]

1. https://hexdocs.pm/ecto/Ecto.Repo.html#c:insert/2-upserts

Finally, let's expose this operation in our Multimedia context with a new function called create_category!, like this:

queries/listings/rumbl/lib/rumbl/multimedia.change1.ex
```
alias Rumbl.Multimedia.Category

def create_category!(name) do
  Repo.insert!(%Category{name: name}, on_conflict: :nothing)
end
```

Now, use the new function in the seeds script like this:

queries/listings/rumbl/priv/repo/seeds.change1.exs
```
alias Rumbl.Multimedia

for category <- ~w(Action Drama Romance Comedy Sci-fi) do
  Multimedia.create_category!(category)
end
```

We use the sigil ~w to define a list of words. Each word represents a category. We then traverse the list of category names, writing them to the database with the new Multimedia.create_category! function.

Let's run the seeds file with mix run:

```
$ mix run priv/repo/seeds.exs
```

Presto! We have categories.

Associating Videos and Categories

Now that we've populated our database with categories, we want to allow users to choose a category when creating or editing a video. To do so, we'll do all of the following:

- Fetch all category names and IDs from the database
- Sort them by the name
- Pass them into the view as part of a select input

To build this feature, we'll need to start with a query. Let's spend a little time with Ecto exploring queries a little more deeply. Fire up your project in IEx, and let's warm up with some queries:

```
iex> import Ecto.Query
iex> alias Rumbl.Repo
iex> alias Rumbl.Multimedia.Category
```

Importing Ecto.Query makes the Ecto query language available to us. That module plays some games with macros to provide a simple and beautiful query syntax with as little ceremony as possible. Since it's a framework in such a central part of database development, the tradeoff of more complexity for the library against more productivity for users makes sense. We also alias Repo and Category. If you find yourself always issuing the same set of commands in a project directory, you can include them in a file called .iex.exs. If you want more details, you can read about customizing iex.[2]

```
iex> query = from c in Category,
...>           select: c.name
iex> Repo.all query
```

First, we create a query. In this case:

- from is a macro that builds a query.
- c in Category means we're pulling rows (labeled c) from the Category schema.
- select: c.name means we're going to return only the name field.

Repo.all is simply a repository function that takes a query and returns all rows. You can see Ecto returns a few debugging lines that contain the exact SQL query we're sending to the database, and the resulting five category names:

```
[debug] QUERY OK source="categories" db=1.9ms
SELECT c0."name" FROM "categories" AS c0 []

["Action", "Drama", "Romance", "Comedy", "Sci-fi"]
```

Ecto's real purpose is to efficiently translate Elixir concepts into a language the database understands. For us, that language will be SQL. We can order category names alphabetically by passing the :order_by option to our query. We can also return a tuple from both the id and name fields.

Let's give it another try:

```
iex> Repo.all from c in Category,
...>           order_by: c.name,
...>           select: {c.name, c.id}
[
  {"Action", 1},
  {"Comedy", 4},
  {"Drama", 2},
  {"Romance", 3},
  {"Sci-fi", 5}
]
```

However, we rarely need to define the whole query at once. Ecto queries are *composable*, which means you can define the query bit by bit:

2. https://hexdocs.pm/iex/IEx.html

```
iex> query = Category
Category
iex> query = from c in query, order_by: c.name
#Ecto.Query<>

iex> query = from c in query, select: {c.name, c.id}
#Ecto.Query<>

iex> Repo.all(query)
[
  {"Action", 1},
  {"Comedy", 4},
  {"Drama", 2},
  {"Romance", 3},
  {"Sci-fi", 5}
]
```

This time, instead of building the whole query at once, we write it in small steps, adding a little more information along the way. You'll see this strategy quite frequently in Elixir because it allows us to use pipes to build complex queries from simpler ones, bit by bit. This strategy works because Ecto defines something called the queryable protocol. from receives a queryable, and you can use any queryable as a base for a new query. A queryable is an Elixir protocol. Recall that protocols like Enumerable (for Enum) define APIs for specific language features. This one defines the API for something that can be queried.

That's also why we can call Repo.all either as Repo.all(Category) or Repo.all(query): because both Category and query implement the so-called Ecto.Queryable protocol. By abiding by the protocol, you can quickly layer together sophisticated queries with Ecto.Query, maintaining clear boundaries between your layers and adding sophistication without complexity.

Let's talk briefly about which pieces of our categories will go where. We'll put query functions in our schema layer. Complex interactions, such as those between our multimedia and users, will go in in contexts. This organization will leave controllers as thin and simple as possible.

Let's implement the layered, composable query strategy. To make our queries compose well, we need functions that take a query as the first argument and return a query. We'll add an alphabetical function to our Category module which will sort the results:

queries/listings/rumbl/lib/rumbl/multimedia/category.change1.ex
```
import Ecto.Query

def alphabetical(query) do
  from c in query, order_by: c.name
end
```

To be more precise, our alphabetical function must receive and return a queryable. With our function in place, let's expose this new feature from a well-named function in our Multimedia context:

queries/listings/rumbl/lib/rumbl/multimedia.change1.ex
```
def list_alphabetical_categories do
  Category
  |> Category.alphabetical()
  |> Repo.all()
end
```

In our user interface, we plan to build a picker that will need names for our users and ids for our backend relationships. We added a Multimedia.list_alphabetical_categories to fetch the data in the order we want. Let's complete the circle by using our new functions to load all the categories in our VideoController and shape the data into a select drop-down within our VideoView:

queries/listings/rumbl/lib/rumbl_web/controllers/video_controller.change1.ex
```
plug :load_categories when action in [:new, :create, :edit, :update]

defp load_categories(conn, _) do
  assign(conn, :categories, Multimedia.list_alphabetical_categories())
end
```

We define a plug that calls our new Multimedia.list_alphabetical_categories function. We also specify the actions that need the categories in the when clause. Now, all sorted categories are available inside @categories in our templates for the actions we specified. You can see how adding our context layer simplifies our controller code.

Let's change the video form template at lib/rumbl_web/templates/video/form.html.eex to include a new select field:

queries/listings/rumbl/lib/rumbl_web/templates/video/form.change1.html.eex
```
<%= label f, :category_id, "Category"%>
<%= select f, :category_id, category_select_options(@categories),
    prompt: "Choose a category" %>
```

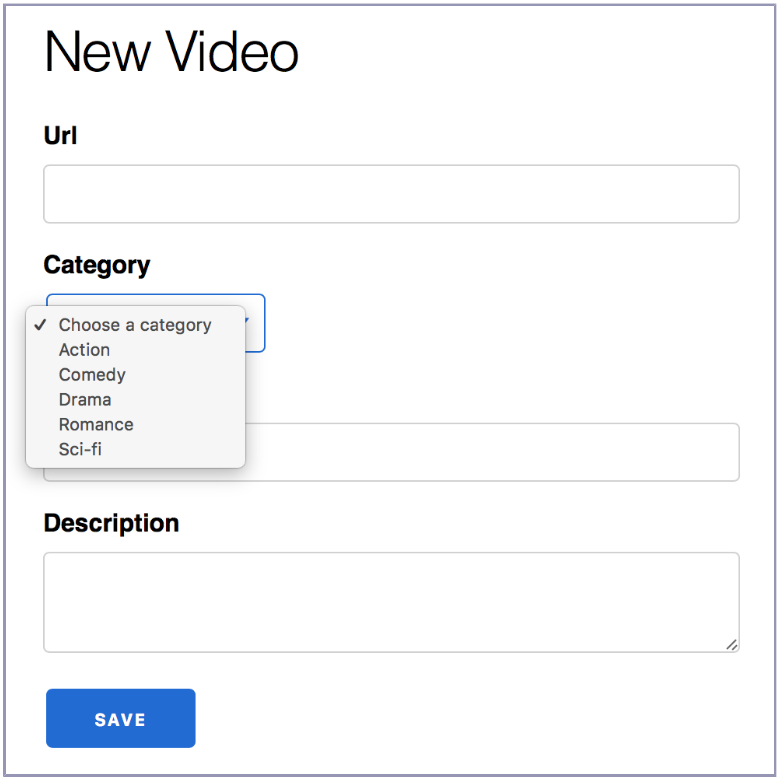
We added a new select field which builds a list of section options using category_select_options. Since that function is new, let's implement it inside our video view in lib/rumbl_web/views/video_view.ex, like this:

queries/listings/rumbl/lib/rumbl_web/views/video_view.change1.ex
```
defmodule RumblWeb.VideoView do
  use RumblWeb, :view

  def category_select_options(categories) do
    for category <- categories, do: {category.name, category.id}
  end
end
```

Remember, views are just modules with pure functions. We'll use the name as the label for each option in a select, and the id as the option value, and a simple for comprehension to walk through the available categories.

That's it. Now we can create videos with optional categories. We're doing so with query logic that lives in its own module so we'll be able to better test and extend those features. Try it out by visiting http://localhost:4000/manage/videos/new:



Before we finish this chapter, we'll add the proper mechanisms to ensure that the category sent by the user is valid. But first, let's take this opportunity to explore Ecto queries a little more deeply.

Diving Deeper into Ecto Queries

So far, you know Ecto queries like a YouTube dog knows how to ride a bike. We've written our first query and we know that queries compose, but we still haven't explored many concepts. It's time to take off the training wheels and see more-advanced examples.

Open up IEx once more, and let's retrieve a single user:

```
iex> import Ecto.Query
iex> alias Rumbl.Repo
iex> alias Rumbl.Accounts.User
iex> alias Rumbl.Multimedia.Video

iex> username = "josevalim"
"josevalim"

iex> Repo.one(from u in User, where: u.username == ^username)
...
%Rumbl.Accounts.User{username: "josevalim", ...}
```

We're using the same concepts you learned before:

- Repo.one means return one row.
- from u in User means we're reading from the Accounts.User schema.
- where: u.username == ^username means return the row where u.username == ^username. The ^ (caret) is used for injecting a value or expression for interpolation into an Ecto query
- When the select part is omitted, the whole struct is returned, as if we'd written select: u.

Repo.one doesn't mean "return the first result." It means "one result is expected, so if there's more, fail." This query language is a little different from what you may have seen before. This API is not just a composition of strings. By relying on Elixir macros, Ecto knows where user-defined variables are located, so it's easier to protect the user from security flaws like SQL-injection attacks.

Ecto queries also do a good part of the query normalization at compile time, so you'll see better performance while leveraging the information in our schemas for casting values at runtime. Let's see some of these concepts in action by using an incorrect type in a query:

```
iex> username = 123
123

iex> Repo.all(from u in User, where: u.username == ^username)

** (Ecto.Query.CastError) iex:7: value `123` in `where`
   cannot be cast to type :string in query:

from u in Rumbl.Accounts.User,
  where: u.username == ^123,
  select: u
```

The ^ operator interpolates values into our queries where Ecto can scrub them and safely put them to use, without the risk of SQL injection. Armed with our schema definition, Ecto is able to cast the values properly for us and match up Elixir types with the expected database types.

In other words, we define the repository and schemas and let Ecto changesets and queries tie them up together. This strategy gives developers the proper level of isolation because we mostly work with data, which is straightforward, and leave all complex operations to the repository.

The Query API

So far, we've used only the == operator in queries, but Ecto supports a wide range of them:

- Comparison operators: ==, !=, <=, >=, <, >
- Boolean operators: and, or, not
- Inclusion operator: in
- Search functions: like and ilike
- Null check functions: is_nil
- Aggregates: count, avg, sum, min, max
- Date/time intervals: datetime_add, date_add
- General: fragment, field, and type

In short, you can use many of the same comparison, inclusion, search, and aggregate operations for a typical query that you'd use in Elixir. You can see documentation and examples for many of them in the Ecto.Query.API documentation.[3] Those are the basic features you're going to use as you build queries. You'll use them from two APIs: keywords syntax and pipe syntax. Let's see what each API looks like.

Writing Queries with Keywords Syntax

The first syntax expresses different parts of the query by using a keyword list. For example, take a look at this code for counting all users with usernames starting with j or c. You can see keys for both :select and :where:

```
iex> Repo.one from u in User,
...>          select: count(u.id),
...>          where: ilike(u.username, "j%") or
...>                 ilike(u.username, "c%")
2
```

3. http://hexdocs.pm/ecto/Ecto.Query.API.html

The u variable is bound as part of Ecto's from macro. Throughout the query, it represents entries from the User schema. If you attempt to access u.unknown or match against an invalid type, Ecto raises an error. Bindings are useful when our queries need to join across multiple schemas. Each join in a query gets a specific binding.

Let's also build a query to count all users:

```
iex> users_count = from u in User, select: count(u.id)
#Ecto.Query<from u in Rumbl.Accounts.User, select: count(u.id)>
```

Simple enough. We use from to build a query, selecting count(u.id). Now, let's say that we want to take advantage of this fantastic count feature to build some more-complex queries. Since the best usernames have a j, let's count the users that match a case-insensitive search for j, like this:

```
iex> j_users = from u in users_count, where: ilike(u.username, ^"%j%")
#Ecto.Query<from u in Rumbl.Accounts.User,
 where: ilike(u.username, ^"%j%"), select: count(u.id)>
```

Beautiful. You've built a new query, based on the old one. Although we've used the same binding as before, u, we didn't have to. You're free to name your query variables however you like, because Ecto doesn't use their names. The following query is equivalent to the previous one:

```
iex> j_users = from q in users_count, where: ilike(q.username, ^"%j%")
#Ecto.Query<from u in Rumbl.Accounts.User,
 where: ilike(u.username, ^"%j%"), select: count(u.id)>
```

You can use that composition wherever you have a query, be it written with the keyword syntax or the pipe syntax that you'll learn next.