

Extracted from:

# Programming Phoenix

Productive |> Reliable |> Fast

This PDF file contains pages extracted from *Programming Phoenix*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2016 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The  
Pragmatic  
Programmers

# Programming Phoenix

Productive |> Reliable |> Fast



Chris McCord,  
Bruce Tate,  
and José Valim

*edited by Jacquelyn Carter*

# Programming Phoenix

Productive |> Reliable |> Fast

Chris McCord  
Bruce Tate  
and José Valim

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at <https://pragprog.com>.

The team that produced this book includes:

Jacquelyn Carter (editor)  
Potomac Indexing, LLC (index)  
Eileen Cohen (copyedit)  
Gilson Graphics (layout)  
Janet Furlow (producer)

For customer support, please contact [support@pragprog.com](mailto:support@pragprog.com).

For international rights, please contact [rights@pragprog.com](mailto:rights@pragprog.com).

Copyright © 2016 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-13: 978-1-68050-145-2

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—April 2016

# Ecto Queries and Constraints

In the last chapter, we extended our application domain by associating videos to users. Now we'll let users organize their videos with categories. We want our users to select which category a video belongs to upon video creation. To build this feature, you'll need to learn more about Ecto queries and the different ways you can 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 Postgres 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.

## Adding Categories

In this section, we're going to add some categories. We'll use many of the same techniques we discovered in our user-to-video relationship to manage the relationships between videos and categories. A video optionally belongs to a category, one chosen by the end user. First, let's generate the model and migration, using `phoenix.gen.model`, like this:

```
$ mix phoenix.gen.model Category categories name:string
* creating priv/repo/migrations/20150829145417_create_category.exs
* creating web/models/category.ex
* creating test/models/category_test.exs
```

This generator will build the model with a schema and migration for us.

## Generating Category Migrations

Next let's edit our migration to mark the name field as NOT NULL and create an unique index for it:

```
queries/listings/rumbl/priv/repo/migrations/20150918041601_create_category.exs
```

```
defmodule Rumbl.Repo.Migrations.CreateCategory do
  use Ecto.Migration

  def change do
    create table(:categories) do
      add :name, :string, null: false

      timestamps
    end

    create unique_index(:categories, [:name])
  end
end
```

Next, we add the referential constraints to our Video schema, like this:

```
queries/listings/rumbl/web/models/video.change1.ex
```

```
Line 1 schema "videos" do
-   field :url, :string
-   field :title, :string
-   field :description, :string
5   belongs_to :user, Rumbl.User
-   belongs_to :category, Rumbl.Category
-
-   timestamps
- end
10
- @required_fields ~w(url title description)
- @optional_fields ~w(category_id)
```

On lines 6 and 12, we create a simple belongs-to relationship and make a new category\_id field optional.

Let's use `mix ecto.gen.migration` to build a migration that adds category\_id to video:

```
$ mix ecto.gen.migration add_category_id_to_video
* creating priv/repo/migrations
* creating
priv/repo/migrations/20150829190252_add_category_id_to_video.exs
```

This relationship allows us to add a new category ID to our existing videos. Now open up your new `priv/repo/migrations/20150829190252_add_category_id_to_video.exs` and key this in:

```
queries/listings/rumbl/priv/repo/migrations/20150918042635_add_category_id_to_video.exs
def change do
  alter table(:videos) do
    add :category_id, references(:categories)
  end
end
```

This code means that we want the database to enforce a constraint between videos and categories. The database will help make sure that the `category_id` specified in the video exists, similar to what we've done between videos and users. Finally, migrate your database with your two new migrations:

```
$ mix ecto.migrate
15:05:52.249 [info] ==
    Running Rumbl.Repo.Migrations.CreateCategory.change/0 forward
15:05:52.249 [info] create table categories
15:05:52.494 [info] == Migrated in 2.4s
15:05:52.573 [info] ==
    Running Rumbl.Repo.Migrations.AddCategoryIdToVideo.change/0 forward
15:05:52.573 [info] alter table videos
15:05:52.587 [info] == Migrated in 0.1s
```

We migrated our categories and added the proper foreign keys. The database will maintain the database integrity, regardless of what we do on the Phoenix side. It's time to populate our database with some categories.

## Setting Up Category Seed Data

We expect our categories to be fixed. After we define a few of them, we don't expect them to change. For this reason, we don't need to create a controller with a view and templates to manage them. Instead, let's create one small script that we can run every time we need to insert data in the database.

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.

Let's add the following to the end of the seeds file:

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

for category <- ~w(Action Drama Romance Comedy Sci-fi) do
  Repo.insert!(%Category{name: category})
end
```

We set up some aliases and then traverse a list of category names, writing them to the database. Let's run the seeds file with `mix run`:

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

Presto! We have categories. Before we move on, let's look at a potential error condition here. We've all been in situations where small developer mistakes snowballed, creating bigger problems. Consider the case in which a developer accidentally adds our categories twice. Then, before the developer discovers the mistake, *an end user uses two different categories of the same name*. Then, the developer mistakenly deletes a category with user data, and our snowball rolls on, picking up destructive mass and speed.

Let's check to see what happens if someone runs the seeds file twice:

```
$ mix run priv/repo/seeds.exs
** (Ecto.ConstraintError) constraint error when attempting to insert
model:

  * unique: categories_name_index
```

One of the constraints we added to the database was a unique constraint for the name column. When we try to insert an existing category, the database refuses and Ecto throws an exception, *as it should*. Our developer's snowball is snuffed out from the very beginning, before the tiny mistake has any chance to grow. We can do better, though. Let's prevent an error in the first place by simply checking if the category exists before creating it:

```
queries/listings/rumbl/priv/repo/seeds.change2.exs
alias Rumbl.Repo
alias Rumbl.Category

for category <- ~w(Action Drama Romance Comedy Sci-fi) do
  Repo.get_by(Category, name: category) ||
    Repo.insert!(%Category{name: category})
end
```

For a script, that's an adequate solution. We're not going to run it that often, and we're likely to be able to address any problems quickly. However, for production code—which might process thousands of requests per second—we



need a more robust error strategy that works with the protections we've built into the database. You'll explore such strategies later on in this chapter when you read about constraints.

## 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 categories 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 want to write 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.Category

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

The `Repo.all` function takes an Ecto query, and we've passed it a basic one. In this case:

- `Repo.all` means return all rows.
- `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.

And 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] SELECT c0."name" FROM "categories" AS c0 [] OK query=0.7ms
["Action", "Drama", "Romance", "Comedy", "Sci-fi"]
```

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:

```
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. 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.

Use what you've learned to associate videos and categories in our application. As with `changesets`, add code that builds and transforms queries to models while all interaction with the repository belongs to the controller—because the controller is the place we want complex interactions to live.

Let's add two functions to our `Category` module, one that sorts the results and another that fetches names and IDs:

```
queries/listings/rumbl/web/models/category.change1.ex
def alphabetical(query) do
  from c in query, order_by: c.name
end

def names_and_ids(query) do
  from c in query, select: {c.name, c.id}
end
```

Those functions receive queries, or more precisely, queryables, and return queryables. With our functions in place, you can now load all categories in `VideoController`:

```
queries/listings/rumbl/web/controllers/video_controller.change1.ex
```

```
alias Rumbl.Category

plug :load_categories when action in [:new, :create, :edit, :update]

defp load_categories(conn, _) do
  query =
    Category
    |> Category.alphabetical
    |> Category.names_and_ids
  categories = Repo.all query
  assign(conn, :categories, categories)
end
```

We define a plug that builds a query by composing multiple functions that we define in our Category model. Once the query is built, we hand it off to the repository, which fetches the names and IDs tuples and assigns them to the connection. Now, those names and IDs are available as @categories in our templates for the actions we specify in our when clause. We'll use the name as the label for each option in a select and the id as the option value.

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

```
queries/listings/rumbl/web/templates/video/form.change1.html.eex
```

```
<div class="form-group">
  <%= label f, :category_id, "Category", class: "control-label" %>
  <%= select f, :category_id, @categories, class: "form-control",
    prompt: "Choose a category" %>
</div>
```

And change video/new.html.eex to pass the @categories in conn.assigns when rendering the form:

```
queries/listings/rumbl/web/templates/video/new.change1.html.eex
```

```
<h2>New video</h2>

<%= render "form.html", changeset: @changeset, categories: @categories,
  action: video_path(@conn, :create) %>

<%= link "Back", to: video_path(@conn, :index) %>
```

Also change video/edit.html.eex:

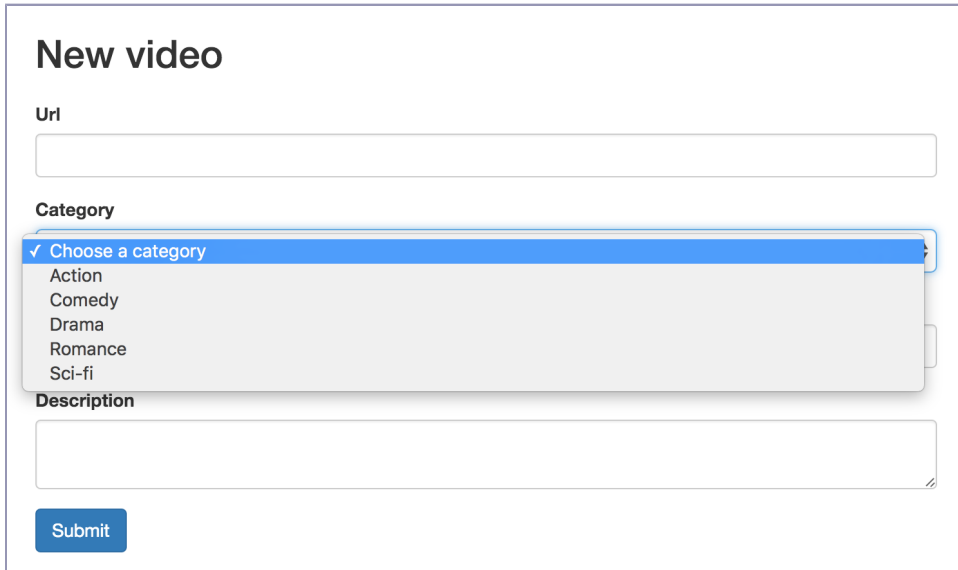
```
queries/listings/rumbl/web/templates/video/edit.change1.html.eex
```

```
<h2>Edit video</h2>

<%= render "form.html", changeset: @changeset, categories: @categories,
  action: video_path(@conn, :update, @video) %>

<%= link "Back", to: video_path(@conn, :index) %>
```

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>:



**New video**

Url

Category

- ✓ Choose a category
- Action
- Comedy
- Drama
- Romance
- Sci-fi

Description

Submit

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.User

iex> username = "josevalim"
"josevalim"

iex> Repo.one(from u in User, where: u.username == ^username)
...
%Rumbl.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 `User` schema.
- `where: u.username == ^username` means return the row where `u.username == ^username`. Remember, the `^` operator (called the pin operator) means we want to keep `^username` the same.
- 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:

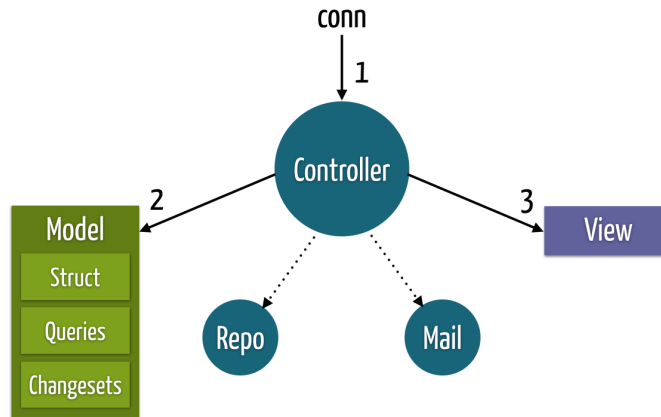
```
iex> username = 123
123

iex> Repo.all(from u in User, where: u.username == ^username)
** (Ecto.CastError) iex:4: value `123` in `where`
cannot be cast to type :string in query:
  from u in Rumbl.User,
  where: u.username == ^123
```

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.

You already know a bit about the differences between traditional MVC and Phoenix's tweak from the perspective of controllers. More explicitly, we'd like to keep functions with side effects—the ones that change the world around us—in the *controller* while the *model* and *view* layers remain side effect free. Since Ecto splits the responsibilities between the repository and its data API, it fits our world view perfectly. This figure shows how it all fits together:



When a request comes in, the controller is invoked. The controller might read data from the socket (a side effect) and parse it into data structures, like the params map. When we have the parsed data, we send it to the model, which transforms those parameters into changesets or queries.

Elixir structs, Ecto changesets, and queries are just data. We can build or transform any of them by passing them from function to function, slightly modifying the data on each step. When we've molded the data to the shape of our business-model requirements, we invoke the entities that can change the world around us, like the repository (Repo) or the system responsible for delivering emails (Mail). Finally, we can invoke the view. The view converts the model data, such as changesets and structs, into view data, such as JSON maps or HTML strings, which is then written to the socket via the controller—another side effect.

Because the controller already encapsulates side effects by reading and writing to the socket, it's the perfect place to put interactions with the repository, while the model and view layers are kept free of side effects. When you get the layers of an application right, you often see that these kinds of benefits come in bunches. The same strategy that improves the manageability of our code will also make our code easier to test.

## 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.<sup>1</sup> 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
```

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.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.User, where: ilike(u.username, ^"%j%"),
select: count(u.id)>
```

1. <http://hexdocs.pm/ecto/Ecto.Query.API.html>

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.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.