Extracted from:

Programming Ecto

Build Database Apps in Elixir for Scalability and Performance

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

ecto

Programming Ecto

Build Database Apps in Elixir for Scalability and Performance

Darin Wilson
Eric Meadows-Jönsson
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Programming Ecto

Build Database Apps in Elixir for Scalability and Performance

Darin Wilson

Eric Meadows-Jönsson

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
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Kim Cofer
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

ISBN-13: 978-1-68050-282-4
Book version: P1.0—April 2019

At the end of the last chapter, we saw how schemas provide a quick method for inserting new records into the database, even with associated records. But a database is only as good as the quality of the data that it contains, so we need to be careful about the modifications we make to that data. The Ecto.Changeset module provides a rich data structure and a wide array of functions that helps us manage making changes safely and securely.

In this chapter, we will take a deep dive into the world of changesets. We will start by taking a high-level look at the process of making a change, then look at each step of the process in detail: casting and filtering user-provided data, validating the data, and capturing errors. Finally, we will look at how changesets help us with the often-tricky process of working with associations and embeds.

Introducing Changesets

Changesets manage the update process by breaking it into three distinct stages: casting and filtering user input, validating the input, then sending the input to the database and capturing the result. If you think of it as a pipeline, it would look something like this:

```
data
|> cast_and_filter_fields
|> validate_change
|> validate_another_change
|> send_to_database
```

We'll look at each step in detail, but here's what the process looks like in code. The following example inserts a new Artist record, based on data supplied by the user:

priv/examples/changeset_01.exs
```
import Ecto.Changeset

params = %{name: "Gene Harris"}
changeset =
  %Artist{}
  |> cast(params, [:name])
  |> validate_required([:name])

case Repo.insert(changeset) do
  {:ok, artist} -> IO.puts("Record for #{artist.name} was created.")
  {:error, changeset} -> IO.inspect(changeset.errors)
end
```

As this example demonstrates, changesets help us with the entire life cycle of making a change, starting with raw data, and ending with the operation succeeding or failing at the database level. Let's now zoom in on each step.

Casting and Filtering

The first step is to take the raw input data that you want to apply to the database and generate an Ecto.Changeset struct. We call this “casting and filtering” because we perform any needed type casting operations (for example, turning a string into an integer), and we filter out any values we don’t want to use. You can do this two different ways, depending on where your data is coming from. We’ll look at both in the following sections.

Creating Changesets Using Internal Data

If the data is internal to the application (that is, you’re generating the data yourself in your application code), you can create a changeset using the Ecto.Changeset.change function. Here’s how you would create a changeset that inserts a new Artist record:

priv/examples/changeset_02.exs
```
import Ecto.Changeset

changeset = change(%Artist{name: "Charlie Parker"})
```

The import statement makes all of the functions in Ecto.Changeset available to our code. For brevity, we won’t include this in the rest of the examples.

To make changes to an existing record, the process is similar, but instead of passing in a new struct, we use a record fetched from Repo:

```
artist = Repo.get_by(Artist, name: "Bobby Hutcherson")
changeset = change(artist)
```

We can add the data we’d like to change as optional arguments to the change function. This is how we might change the name field to something more formal:

```
artist = Repo.get_by(Artist, name: "Bobby Hutcherson")
changeset = change(artist, name: "Robert Hutcherson")
```

At this point, changeset is just a data structure in memory—no communication with the database has happened yet. As we’ve seen with the Repository pattern, nothing happens with the database until we get Repo involved. If we wanted to commit the change, we’d need to call Repo.update(changeset) and check the result to see if it succeeded.

Before we do that, we can take a peek at the changes that will be applied. The changes field of our changeset tells us what’s going to be updated:

```
changeset.changes
#=> %{name: "Robert Hutcherson"}
```

We can also use the change function to add more changes to a changeset that's already been created—instead of passing in an Artist struct as the first argument, we can pass another changeset. Using the changeset value we created in the last code example, we could add the artist's birth date to the list of items we're going to update:

```
changeset = change(changeset, birth_date: ~D[1941-01-27])
```

And of course, it's possible to add both changes into a single change call:

```
artist = Repo.get_by(Artist, name: "Bobby Hutcherson")
changeset = change(artist, name: "Robert Hutcherson",
  birth_date: ~D[1941-01-27])
```

In either case, calling changes will now show both of the values that we are updating:

```
changeset.changes
#=> %{birth_date: ~D[1941-01-27], name: "Robert Hutcherson"}
```

The data we've been using so far has been generated in our code. In most cases however, the data you want to apply will be coming from outside of the controlled environment of your own code: forms your application presents to end users, API calls, command-line parameters, CSVs or other data files, and so forth. To deal with this potentially unruly data, Ecto provides the cast function for creating changesets.

Creating Changesets Using External Data

When working with data coming from external sources, it's important to take extra care. The cast function plays a similar role to change, as it's used to take raw data and return a Changeset struct, but it's got a few extra features to help make sure you're getting only the data you want.

The cast function has three required arguments. The first is the same as change: it should be a data structure representing the record you want to apply your changes to. This could be a new schema struct (for example %Artist{}), a schema struct representing a record fetched from the database, or another changeset.

The second argument is a map containing raw data that you want to apply. The third is a list of the parameters that you'll allow to be added to the changeset. It acts like a filter: only parameters specified in the list will be added to the changeset. The rest will be discarded.

Here is how we could create a changeset for a new Artist record using user-supplied parameters. (In the following examples, we'll use the params variable to represent values supplied by the user.)

priv/examples/changeset_03.exs

```
# values provided by the user
params = %{"name" => "Charlie Parker", "birth_date" => "1920-08-29",
  "instrument" => "alto sax"}

changeset = cast(%Artist{}, params, [:name, :birth_date])
changeset.changes
#=> %{birth_date: ~D[1920-08-29], name: "Charlie Parker"}
```

Take a close look at the result of the changes call, and you'll see what cast has done for us. First, the instrument value provided in the params map does not appear in the changeset. This is because we only specified :name and :birth_date in the list of allowed values, so Ecto dropped the instrument field for us. This can be useful when importing data from sources you don't control. If you were importing data from a CSV, for example, there could be extra columns of data that you don't need. This setting helps you get rid of them.

Second, the call to cast converted the birth_date value from the string "1920-08-29" to an Elixir Date struct. As the name suggests, cast will perform type casting when turning the raw input into a changeset, whenever it can. In this case, our Artist schema defined birth_date as the :date type, so Ecto parsed the string value into a Date when creating the changeset. This worked because we received the date in a standard format. If we got an unknown date format, Ecto would not be able to cast it and the changeset would be invalid. We'll talk more about validating changesets in the next section.

By default, the cast function will treat the empty string "" as nil when creating the changeset. But there may be times when you want other values turned into nil as well. For example, when working with spreadsheets, you'll often see data that looks like this:

```
params = %{"name" => "Charlie Parker", "birth_date" => "NULL"}
```

Instead of getting an empty cell, you get the string "NULL." We can tell Ecto that we want to consider "NULL" an empty value, by adding the empty_values option to cast:

```
params = %{"name" => "Charlie Parker", "birth_date" => "NULL"}
changeset = cast(%Artist{}, params, [:name, :birth_date],
  empty_values: ["", "NULL"])
changeset.changes
#=> %{name: "Charlie Parker"}
```

By adding "NULL" to the empty_values option, we were able to treat the birth_date value as empty, and Ecto dropped it from the list of changes. You can specify as many different values as you need, but don't forget to include "" if you want to convert empty strings as well.