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

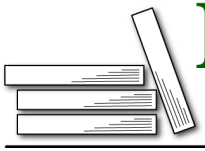
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Programming Ecto

Build Database Apps in Elixir for Scalability and Performance

Darin Wilson
Eric Meadows-Jönsson

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

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

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

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

Running Transactions with Functions

The first way to run Repo.transaction is by passing in a function containing the operations you'd like to run within the transaction. This can be an anonymous function or a named function defined elsewhere. This seems like a good idea—we're functional programmers, and this approach will let us keep using functions. Let's try it out.

To illustrate how this works, we're going to introduce a new database table, and a module to go with it. Imagine that we've decided that we want to keep a log of the changes we make to our database. Every time we make a change, we'll insert a new record into a logs table. We'll use functions in the MusicDB.Log module to create changesets for logging the different operations that we want to perform. It's not too fancy, but it will suffice for our purposes here. Take a peek at the lib/music_db/log.ex module if you're curious to see the details.

Here's what we would do if we wanted to insert a new Artist record, and log the change:

priv/examples/transactions_01.exs

```
artist = %Artist{name: "Johnny Hodges"}
Repo.insert(artist)
Repo.insert(Log.changeset_for_insert(artist))
```

That would work most of the time, but we want to be absolutely certain that both of these inserts succeed: we don't want to add a new Log record if the Artist insert didn't go through, and if the Log insert fails, we want to back out the Artist insert. We can do this by wrapping the two calls in an anonymous function, and passing that function directly to the Repo.transaction function:

```
artist = %Artist{name: "Johnny Hodges"}
Repo.transaction(fn ->
  Repo.insert!(artist)
  Repo.insert!(Log.changeset_for_insert(artist))
end)
#=> {:ok, %MusicDB.Log{ ...}}
```

When a transaction succeeds (as this one did), the transaction function returns a tuple consisting of :ok and the return value of the passed-in function. In this case, the last line of the function inserts the Log struct, so we get the return value of that operation: %MusicDB.Log{...}.

If an error occurs anywhere in the transaction, the database rolls back all of the changes that it performed up to that point, and the transaction function itself raises the error. We can demonstrate this by trying to insert nil for the second operation:

```
artist = %Artist{name: "Ben Webster"}
Repo.transaction(fn ->
  Repo.insert!(artist)
  Repo.insert!(nil) # <-- this will fail
end)
#=> ** (FunctionClauseError) no function clause matching in
#=> Ecto.Repo.Schema.insert/4
```

Elixir rightfully complained about our attempt to insert nil and raised the error. We expect that any changes performed within transaction got rolled back, and we can verify that by making sure no Artist record now exists for Ben Webster:

```
Repo.get_by(Artist, name: "Ben Webster")
# => nil
```

Our transaction worked. The failure of the second insert forced a rollback of the first insert. We're back to where we were before we started.

Forcing a Rollback Within a Transaction

Notice that we've been using insert! with a bang, rather than insert. The two functions are identical, except for one crucial difference: insert will return {:error, value} if the insert fails, but insert! will raise an error. This is a convention that's used in many Elixir libraries, and it's essential when executing transaction with a function.

The documentation for Repo.transaction says this:

> If an unhandled error occurs the transaction will be rolled back and the error will bubble up from the transaction function.

This means that only unhandled errors will trigger the rollback behavior—a return value of {:error, value} from one of the operations isn't going to cut it.

We can demonstrate this by rewriting our transaction so we're inserting changesets rather than schema structs. If we pass an invalid changeset to insert (without the bang) it will return an :error tuple without raising an error. We'll add some debug output so we can see exactly what's going on:

priv/examples/transactions_02.exs

```
cs =
  %Artist{name: nil}
  |> Ecto.Changeset.change()
  |> Ecto.Changeset.validate_required([:name])
Repo.transaction(fn ->
  case Repo.insert(cs) do
    {:ok, _artist} -> IO.puts("Artist insert succeeded")
    {:error, _value} -> IO.puts("Artist insert failed")
```

```
  end
  case Repo.insert(Log.changeset_for_insert(cs)) do
    {:ok, _log} -> IO.puts("Log insert succeeded")
    {:error, _value} -> IO.puts("Log insert failed")
  end
end)
```

We start by creating an intentionally invalid changeset: we pass in nil for the name field, then add a validation declaring that name is required. This should give us :error when we try to insert it. Then we try to insert the changeset and a separate Log changeset within the transaction. The case statements help us to see how each of those operations fare. Here's what happens when we run this:

```
# => Artist insert failed
# => Log insert succeeded
# => {:ok :ok}
```

This is exactly what we *don't* want when working with transactions. The first insert failed, but because we used insert rather than insert! the function returned the tuple {:error, _value} instead of raising an error. If we want to trigger a rollback, we have to raise an Elixir error, and passing an invalid changeset to insert won't do that. You have to use insert! (with a bang) instead. Because we used insert, the transaction continued, and the second insert succeeded. Our database is now in an incorrect state: we have a log record for an insert that didn't actually happen.

One workaround for this behavior is to use the Repo.rollback function. Calling this function will abort the transaction and roll back any changes made so far, just as if an error had occurred. When you call rollback, the transaction function returns {:error, value} where value is the argument passed to the rollback function. With this in mind, we can rewrite the previous example to get the behavior we want:

```
cs = Ecto.Changeset.change(%Artist{name: nil})
  |> Ecto.Changeset.validate_required([:name])
Repo.transaction(fn ->
  case Repo.insert(cs) do
    {:ok, _artist} -> IO.puts("Artist insert succeeded")
    {:error, _value} -> Repo.rollback("Artist insert failed")
  end
  case Repo.insert(Log.changeset_for_insert(cs)) do
    {:ok, _log} -> IO.puts("Log insert succeeded")
    {:error, _value} -> Repo.rollback("Log insert failed")
  end
end)
# => {:error, "Artist insert failed"}
```

That's better. This time, the first insert failed as expected so the rest of the transaction didn't run. The transaction function returned an :error tuple with the value we provided.

Executing Non-Database Operations Within a Transaction

With this knowledge in hand, we can see an opportunity to expand transactions to include non-database operations. Imagine that our app uses an external search engine, such as Elasticsearch. Whenever we change the database, we want to update our search engine as well. But it's important to keep the database and the search engine in sync: if the database changes fail, we don't want to update the search engine, and if the search engine update fails, we want to roll back the changes to the database.

To explore this scenario, our MusicDB app has a MusicDB.SearchEngine module that handles search engine updates via its update function. This is just a placeholder module—our sample app doesn't include a real search engine, so the module's functions just simulate the behavior.

To update the search engine along with the changes to the database, we call the appropriate functions from within the transaction:

priv/examples/transactions_03.exs

```
artist = %Artist{name: "Johnny Hodges"}
Repo.transaction(fn ->
  artist_record = Repo.insert!(artist)
  Repo.insert!(Log.changeset_for_insert(artist_record))
  SearchEngine.update!(artist_record)
end)
```

Provided that our update! function raises an error if it fails, this will do what we want: if either of the insert! calls fail, the search engine update won't run. And if the search engine update fails, Ecto will roll back the database changes and the transaction function will bubble up the error.

Of course, Ecto has no knowledge of how our search engine works, so it would be impossible for it to roll back changes to the search engine. This means that you should run all of your database operations first, then run any non-database operations: you don't want those to run until you're sure the database operations succeeded.

Drawbacks of Using Functions

Running transactions with functions works reasonably well, but it has some drawbacks.

The most serious problem, demonstrated in the last section, is that *we have to be careful that we call Repo functions in the correct way.* Calling insert rather than insert! broke the behavior we were trying to achieve. The compiler can't help us with something like this, so one missed character could put our database into a bad state.

Another problem is that *anonymous functions are not composable:* this limits their reusability. Our last example made changes to an Artist record, saved a log of the change, and updated the search engine. It's possible that in another part of the app we might want to update the artist's albums along with the artist record. It would be nice to take the logic we already have and just add to it, but our anonymous function doesn't lend itself to being extended in that way.

There's still another problem. *We don't have good visibility into exactly what went wrong when a transaction fails.* Recall how much code we had to add when we wanted to see where a failure occurred:

priv/examples/transactions_04.exs

```
cs = Ecto.Changeset.change(%Artist{name: nil})
  |> Ecto.Changeset.validate_required([:name])
Repo.transaction(fn ->
  case Repo.insert(cs) do
    {:ok, _artist} -> IO.puts("Artist insert succeeded")
    {:error, _value} -> Repo.rollback("Artist insert failed")
  end
  case Repo.insert(Log.changeset_for_insert(cs)) do
    {:ok, _log} -> IO.puts("Log insert succeeded")
    {:error, _value} -> Repo.rollback("Log insert failed")
  end
end)
```

That's a lot of extra code for only two Repo calls.

Fortunately, there's a better way. The Ecto.Multi module can help us out with all of these issues. We'll explore that option in the next section.

Running Transactions with Ecto.Multi

The other way to use Repo.transaction is pass in an Ecto.Multi struct, rather than a function. Ecto.Multi allows you to group your database operations into a data structure. When handed to the transaction function, the Multi's operations run in order, and if any of them fail, all of the others are rolled back.

Let's take a look at an earlier example where we ran a transaction with an anonymous function:

priv/examples/transactions_05.exs

```
artist = %Artist{name: "Johnny Hodges"}
```

```
Repo.transaction(fn ->
  Repo.insert!(artist)
  Repo.insert!(Log.changeset_for_insert(artist))
end)
```

Here's how we can rewrite it using Multi:

```
alias Ecto.Multi

artist = %Artist{name: "Johnny Hodges"}
multi =
  Multi.new
  |> Multi.insert(:artist, artist)
  |> Multi.insert(:log, Log.changeset_for_insert(artist))
Repo.transaction(multi)
```

There's a lot here, so let's walk through it.

We start by creating a new Multi with the new function. The Ecto team recommends using this approach rather than trying to create the struct directly; that is, don't try to write something like multi = %Multi{}. The exact structure of Ecto.Multi is subject to future change. Calling new ensures that the struct will come back to you properly initialized. If you create the struct directly, you're on your own.

We then add the two insert operations by piping the Multi into the insert function. The Ecto.Multi module has several functions that mirror the database operation functions in Repo: insert, update, delete, and so on. Each of the operations that we add to the Multi must have a unique name—that's what the :artist and :log atoms are for. After that, we pass exactly what we would pass to the Repo.insert function: an Artist struct for the first call, and our Log changeset for the second.

For this example, we don't have any other options we need to include in our insert calls, but if we did, we could add them here. The functions in Multi can accept the same options as their counterparts in Repo, so anything you might send to Repo.insert can be sent to Multi.insert as well.

At this point, we still haven't touched the database. We just have a list of operations stored in the Multi struct. When we finally pass the struct to Repo.transaction, the database begins executing the operations queued in the Multi. The return value, however, is different than what we get when we pass in a function:

```
Repo.transaction(multi)
#=> {:ok,
#=>  %{
#=>    artist: %MusicDB.Artist{...}
#=>    log: %MusicDB.Log{...}
```

```
#=>  }}
```

The transaction succeeded, so we get a tuple with :ok and a map. The keys in the map are the unique names we provided to each operation in the Multi (:artist and :log in this case). The values are the return values for each of those operations. This makes it easy for us to grab the return values of any or all of the operations we ran. In this case, both of the operations were inserts, so we get structs representing our newly inserted records.

Capturing Errors with Multi

Here's where the two approaches really diverge. If an error occurs in a Multi, we get detailed information on where the error occurred, and what happened just before. Let's take a look.

Examining the Return Value

To see this in action, let's create a new Multi that performs an update on the Artist record we just inserted, then tries to insert an invalid changeset:

priv/examples/transactions_06.exs

```
artist = Repo.get_by(Artist, name: "Johnny Hodges")
artist_changeset = Artist.changeset(artist,
  %{name: "John Cornelius Hodges"})
invalid_changeset = Artist.changeset(%Artist{},
  %{name: nil})
multi =
  Multi.new
  |> Multi.update(:artist, artist_changeset)
  |> Multi.insert(:invalid, invalid_changeset)
Repo.transaction(multi)
#=> {:error, :invalid,
#=>  #Ecto.Changeset<
#=>    action: :insert,
#=>    changes: %{},
#=>    errors: [name: {"can't be blank", [validation: :required]}],
#=>    data: #MusicDB.Artist<>,
#=>    valid?: false
#=>  >, %{}}
```

This time, the Multi failed, so we get a tuple with four items: the :error atom, the name of the operation that failed (:invalid), the value that caused the failure (in this case, the invalid changeset, with a populated errors field), and a map containing the changes so far. The database will have already rolled back these changes, but Ecto provides them for you to inspect if needed.

The benefit of this arrangement is that this single return value tells if we succeeded, or, if we failed, exactly *where* we failed. This means that we can

use pattern matching to respond to each of the success or failure scenarios separately:

```
case Repo.transaction(multi) do
  {:ok, _results} ->
    IO.puts "Operations were successful."
  {:error, :artist, changeset, _changes} ->
    IO.puts "Artist update failed"
    IO.inspect changeset.errors
  {:error, :invalid, changeset, _changes} ->
    IO.puts "Invalid operation failed"
    IO.inspect changeset.errors
end
```

That's a lot cleaner than what we had when we were using anonymous functions with Repo.transaction. Here we used a single case statement as our responses were fairly short. But you could also use pattern-matched functions if you needed more complex responses.

Examining the List of Changes So Far

The last value of the returned tuple is supposed to be a list of changes that occurred before the error happened. Let's take another look at what we got in the last example:

```
artist = Repo.get_by(Artist, name: "Johnny Hodges")
artist_changeset = Artist.changeset(artist,
  %{name: "John Cornelius Hodges"})
invalid_changeset = Artist.changeset(%Artist{},
  %{name: nil})
multi =
  Multi.new
  |> Multi.update(:artist, artist_changeset)
  |> Multi.insert(:invalid, invalid_changeset)
Repo.transaction(multi)
#=> {:error, :invalid,
#=>  #Ecto.Changeset<
#=>    action: :insert,
#=>    changes: %{},
#=>    errors: [name: {"can't be blank", [validation: :required]}],
#=>    data: #MusicDB.Artist<>,
#=>    valid?: false
#=>  >, %{}}
```

We got an empty map—that seems surprising. The return value told us that the second operation in the Multi failed, so we would expect to see the result of the first operation in the list of changes so far.

This is because Ecto doesn't like to waste the database's time. If the Multi contains operations that use changesets, Ecto first checks to make sure all the changesets are valid. If any are not, Ecto won't bother running the transaction at all. It just flags the invalid changeset and sends it back to us in the return value. There's no need to trouble the database with an invalid changeset.

Let's try a different example so we can see something besides an empty map. We'll create a new Multi that starts with a successful update. We'll then force an error by trying to insert a new %Genre{} record with a name that already exists in the database (as you might recall from Working with Constraints, on page ?, the genres table has a unique index on the name column).

```
artist = Repo.get_by(Artist, name: "Johnny Hodges")
artist_changeset = Artist.changeset(artist,
  %{name: "John Cornelius Hodges"})
genre_changeset =
  %Genre{}
  |> Ecto.Changeset.cast(%{name: "jazz"}, [:name])
  |> Ecto.Changeset.unique_constraint(:name)
multi =
  Multi.new
  |> Multi.update(:artist, artist_changeset)
  |> Multi.insert(:bad_genre, genre_changeset)
Repo.transaction(multi)
#=> {:error, :bad_genre, #Ecto.Changeset< ... >,
#=> %{
#=>   artist: %MusicDB.Artist{
#=>     __meta__: #Ecto.Schema.Metadata<:loaded, "artists">,
#=>     albums: #Ecto.Association.NotLoaded<association
#=>       :albums is not loaded>,
#=>     birth_date: nil,
#=>     death_date: nil,
#=>     id: 4,
#=>     inserted_at: ~N[2018-03-23 14:02:28],
#=>     name: "John Cornelius Hodges",
#=>     tracks: #Ecto.Association.NotLoaded<association
#=>       :tracks is not loaded>,
#=>     updated_at: ~N[2018-03-23 14:02:28]
#=>   }
#=> }}
```

Now we can get a good look at that last value. The keys in the map correspond to our named Multi functions that have already been run. In this example, we just had the one :artist update so that's all this map contains. The value of the item is the result of the operation. Here we can see that our "Johnny Hodges" record was updated to "John Cornelius Hodges" as we expected. But because

the Multi failed (thanks to the addition of our bad_genre operation), the database rolled back the change. We can confirm that by looking at the database again:

```
Repo.get_by(Artist, name: "John Cornelius Hodges")
#=> nil
```

We get no records back when we search for "John Cornelius Hodges," which confirms that our update was indeed rolled back.

Optimizing Multi with Changesets

One important consideration with Multi is that the transaction call works with unhandled errors the same way as it does with functions: they're bubbled up to the function that called the transaction. Consider this example:

```
multi =
  Multi.new
  |> Multi.insert(:artist, %Artist{})
    Repo.transaction(multi)
    #=> ** (Postgrex.Error) ERROR 23502 (not_null_violation): null value
    #=>  in column "name" violates not-null constraint
```

Instead of passing a changeset to insert we passed in an empty Artist struct. Our database requires that all records in artists have a non-null name field, so the insert operation fails. This results in transaction raising an error, rather than returning the nicely arranged tuple we saw in the last example.

Given this behavior, it's best to use changesets with Multi whenever possible. Creating changesets with validations will help Ecto catch errors within the bounds of your Elixir code before they hit the database. Of course, you always need to consider that unhandled errors can happen, and you'll need to design your code to respond to those errors in a way that minimizes impact to your users. But you can reduce the occurrences of those kinds of errors by fortifying your changesets as much as possible.

Executing Non-Database Operations with Multi

Based on what we've seen of Multi so far, it might appear that executing transaction with functions has one clear advantage: functions allow you to run any Elixir code within the transaction. Recall our earlier example of updating a search engine within a transaction call. Fortunately, Multi offers this functionality as well. The run function allows you to add any named or anonymous function to be run as part of the Multi. Here's how we might add the search engine update logic we talked about earlier in this chapter:

priv/examples/transactions_07.exs

```
artist = %Artist{name: "Toshiko Akiyoshi"}
```

```
multi =
  Multi.new()
  |> Multi.insert(:artist, artist)
  |> Multi.insert(:log, Log.changeset_for_insert(artist))
  |> Multi.run(:search, fn _repo, changes ->
    SearchEngine.update(changes[:artist])
  end)
Repo.transaction(multi)
```

In this example, we used an anonymous function for the run operation. The function accepts two arguments, our current Repo and a map of the changes made in the Multi so far. We need the Artist record that we inserted, so we grab the :artist item from the changes map. Ecto expects our function to return {:ok, value} if the function succeeded or {:error, value} if it failed. In that case, value can be any value of our choosing.

For more flexibility, we can use Multi.run/5, which lets us specify the module, the function, and a list of additional arguments separately:

```
multi =
  Multi.new()
  |> Multi.insert(:artist, artist)
  |> Multi.insert(:log, Log.changeset_for_insert(artist))
  |> Multi.run(:search, SearchEngine, :update, ["extra argument"])
```

With this form of run, Ecto will still pass in the Repo and the list of changes to the specified function—these will be the first arguments passed to the function, with the arguments you specify immediately following. The last line in the preceding code will result in SearchEngine.update being called like this: SearchEngine.update(repo, changes, "extra argument").

The run function gives you the flexibility to execute any Elixir code as part of your transaction. This is useful for non-database operations, but it's also useful for database operations that Multi does not directly support. For example, there is no Multi.all function to mirror the Repo.all function. If you need to run a query within an operation, you could call Repo.all within a function called by run.