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

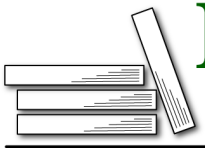
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

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

CHAPTER 9

Testing with Sandboxes

This chapter will show you how to turbo-charge your test suite with *sandboxes.*

Sandboxes allow you to run your database tests concurrently, while still keeping the database state of each test isolated from the others. The secret sauce underlying this feature is a special pool of database connections with an ownership mechanism that allows you to control how connections are used and shared between processes. Using the sandbox can significantly reduce the time it takes to run your test suite, so you should take advantage of this feature when you can.

We'll walk through the basics of setting up sandboxes, and show you what to do if your tests need connections shared across multiple processes. By the end, your machine will be zipping through your test suite at top speed—you'll need to find another excuse to go get coffee.

Setting Up an Async Test

To use the sandbox, change your `Repo` configuration to use the sandbox pool. We only want to do this when we're in our test environment, so make the following change in `config/test.exs` and only there:

priv/examples/sandboxes_01.exs

```
config :music_db, MusicDB.Repo,
  pool: Ecto.Adapters.SQL.Sandbox
  # other settings here
```

By changing the `pool:` setting, we're telling Ecto that we will not be using the default connection pool and instead give the sandbox full control over how connections are checked out and used by processes.

Next, we need to set our sandbox to the correct ownership mode. We'll cover this setting in more detail later, but for now, add this line to test/test_helper.exs:

```
Ecto.Adapters.SQL.Sandbox.mode(MusicDB.Repo, :manual)
```

Now we're ready to write our test case. Create a file called test/album_test.exs, and add the following code:

```
defmodule MusicDB.AlbumTest do
  use ExUnit.Case, async: true

  setup do
    :ok = Ecto.Adapters.SQL.Sandbox.checkout(MusicDB.Repo)
  end

  test "insert album" do
    album = MusicDB.Repo.insert!(%MusicDB.Album{title: "Giant Steps"})
    new_album = MusicDB.Repo.get!(MusicDB.Album, album.id)
    assert new_album.title == "Giant Steps"
  end
end
```

Notice that we set async: true on the first line of the test. This tells Elixir that it's safe to run this test concurrently with other tests.

We also added a setup block that calls the checkout function on the sandbox. This is how we obtain the database connection that we'll be using throughout our test. We need to make this call because we set the ownership mode to :manual in our test helper, but that's not the case for all of the ownership modes. Let's take a look at the different ownership modes we can use.

Changing the Ownership Mode

The ownership mode affects the way the sandbox interacts with different processes. You can use three different modes: :auto, :manual, and :shared.

With :auto the sandbox functions like a normal pool: each process gets its own connection from the pool and has exclusive access to the connection while it is checked out. And, like a normal pool, the connections are checked out automatically when your code needs to run a database operation, and checked back in when the operation completes. The connection is still in a sandboxed transaction that is rolled back when the connection is checked in, but you can't be sure that you'll get the same connection each time you access the database. This means that there's no guarantee that our calls to insert! and get! in our test case will use the same connection; if they use different connections, the get! call will fail when it can't find the album we inserted in the other connection.

Use :auto when you don't need to retain the state of your database throughout your test case. For example, if you're only making one database call in the test, or the result of a database call isn't dependent on the result of previous database calls, :auto is a good choice. This will likely be rare, however.

In :manual mode, which we used in the previous example, the connection is explicitly checked out in the setup block; and is not checked back in until the test exits. In this mode we can be sure that we're only using a single connection through our whole test. Any changes we make in one part of the test will be available to all the other parts, and everything will get rolled back at the end of the test.

Use :manual when you're making multiple database calls in your test case, and later calls are dependent on the result of earlier calls. It's also important that the database calls are run from the same Elixir process.

For tests that use multiple processes, Ecto has :shared mode. So far, our examples have only used one process, but consider an example like this:

priv/examples/sandboxes_02.exs

```
test "insert album" do
  task = Task.async(fn ->
    album = MusicDB.Repo.insert!(%MusicDB.Album{title: "Giant Steps"})
    album.id
  end)

  album_id = Task.await(task)
  assert MusicDB.Repo.get(MusicDB.Album, album_id).title == "Giant Steps"
end
```

In this somewhat contrived example, we're running a database operation in a separate process using Task.async. If we tried to run this test in manual or auto mode, it would crash with the error:

```
** (DBConnection.OwnershipError) cannot find ownership process
  for #PID<0.165.0>.
```

This is because the process that's trying to connect to the database (which we initiated with Task.async) was not the process that checked out the connection. With manual or auto, only the process that checked out the connection can use it.

We can work around this limitation with :shared mode. In this mode, the connection is checked out explicitly like in manual mode but the connection is available to all processes. So if you have a test that uses multiple processes, shared mode allows a single checked-out connection to be used by all of them, and you can be sure the database state is consistent throughout the test.

When working with shared mode, the setup in test/album_test.exs is a little different:

```
setup do
  :ok = Ecto.Adapters.SQL.Sandbox.checkout(MusicDB.Repo)
  Ecto.Adapters.SQL.Sandbox.mode(MusicDB.Repo, {:shared, self()})
end
```

In addition to setting the mode as :shared, we need to provide the process that's checking out the connection, so we pass in self() along with :shared as a tuple.

This seems like a good solution, but it comes at a cost: tests that use shared mode cannot be safely run concurrently. The connection is shared among *all* processes, so any test running concurrently could pollute the database state and cause other tests to fail. As a result, if you're using shared mode, you need to disable concurrency by removing async: true from use ExUnit.Case, async: true.

This is a reasonable trade-off if you don't have a large test suite. It's relatively easy to set up shared mode and not running a few tests concurrently won't be a huge time hit. But if you have a large test suite and need to run much of it in shared mode, this could be a significant setback. Fortunately, Ecto provides us with a way out. In the next section, we'll see how to get the best of both worlds: connections shared between multiple processes, and fast, concurrent tests.

Safely Sharing Connections with Allowances

To work around the limitations of shared mode, Ecto provides a mechanism called *allowances*. This allows us to pick and choose which processes we share our database connection with. We can keep a single database connection for all processes needed for our test, and be sure that the database state is isolated from any other tests running concurrently.

Let's go back to the test we looked at in the last section:

priv/examples/sandboxes_03.exs

```
test "insert album" do
  task = Task.async(fn ->
    album = MusicDB.Repo.insert!(%MusicDB.Album{title: "Giant Steps"})
    album.id
  end)

  album_id = Task.await(task)
  assert MusicDB.Repo.get(MusicDB.Album, album_id).title == "Giant Steps"
end
```

As we mentioned earlier, running this test in manual or auto mode will cause a crash, because the process in Task.async does not have access to the connection checked out by the test process. We need to "allow" the task to use the test

process as the ownership process. We can do that with Ecto.Adapters.SQL.Sandbox.allow/4:

```
test "insert album" do
  parent = self()
  task = Task.async(fn ->
    Ecto.Adapters.SQL.Sandbox.allow(MusicDB.Repo, parent, self())
    album = MusicDB.Repo.insert!(%MusicDB.Album{title: "Giant Steps"})
    album.id
  end)

  album_id = Task.await(task)
  assert MusicDB.Repo.get(MusicDB.Album, album_id).title == "Giant Steps"
end
```

The call to allow ensures that the test and task processes share the same connection.

It would also be possible to call allow/4 from the test process, rather than the async process, but this would introduce a race condition: it's possible that the async process would call MusicDB.Repo.insert! before allow has finished executing. To prevent this, we would have to synchronize the start of the async process and the call to allow. The synchronization could look something like this:

```
test "insert album" do
  task = Task.async(fn ->
    receive do
      :continue -> :ok
    end
    album = MusicDB.Repo.insert!(%MusicDB.Album{title: "Giant Steps"})
    album.id
  end)

  Ecto.Adapters.SQL.Sandbox.allow(MusicDB.Repo, self(), task.pid)
  send(task.pid, :continue)

  album_id = Task.await(task)
  assert MusicDB.Repo.get(MusicDB.Album, album_id).title == "Giant Steps"
end
```

That works, but it does introduce some complexity into the code, so you might find it easier to call allow from within the collaborating process, as we did in the first example.

As powerful as allowances are, it may not always be possible to use them: your code may be structured such that it would be complicated to add them, or you may be using third-party libraries that aren't aware of the Ecto sandbox. In those cases you can fall back to :shared mode, but remember to run those tests synchronously by removing the :async option on use ExUnit.Case.

Wrapping Up

Sandboxes are a powerful feature that can help keep your test suite running at high speed. We covered the basics here, but it's possible that you'll run into some concurrency hiccups, depending on the complexity of your app. If you do run into issues, check out Ecto's documentation for more details on how to troubleshoot.[1]

1. https://hexdocs.pm/ecto_sql/Ecto.Adapters.SQL.Sandbox.html