Refurbish Your Creaky Test Suite

Burn It Down and Build Anew: A Practical Guide to Testing

Brian Meeker

Good morning. If you were expecting to see ???. Unfortunately, they could not make it because of ???. I am here today as your humble backup speaker and we're going to be talking about how we have been refurbishing our legacy test suite at GetThru.

Brian Meeker

Blog: https://brianmeeker.me
GitHub: @CuriousCurmudgeon
Bluesky: @brianmeeker.bsky.social

My name is Brian Meeker. I spend my work hours as an Elixir dev at GetThru.

You can mostly find me these days on Bluesky. These slides will be available on GitHub. I'll have a QR code link at the end to the repo.

Before we dive in, I encourage everybody here in the room, watching virtually, or watching this later on YouTube, to follow the Rule of Two Feet. If you're not getting what you need out of this talk, use your two feet and leave. I will not be offended. Sometimes a talk doesn't click for you. Sometimes you just need a break. Sometimes the hallway track is where it's at. Doesn't matter the reason, if your time can be better spent elsewhere, use your two feet.

At GetThru, over the years our system has accumulated a lot of legacy cruft, including in our test suite.

Today, we're going to talk about how we declared tech debt bankruptcy on our existing test suite and started to build anew. This decision led us to reevaluate the patterns that had led us here. It wasn't going to do any good to start over and make the same mistakes.

The Breaking Point

In 2024 we rebuilt our texting infrastructure. This really pushed the technical limits of our team, including our testing skills. Our test suite had issues before, but this was the breaking point. There were a few major categories of test issues that we were running into.

Process Ownership

```
16:53:33.723 [error] GenServer {GetThru.CampaignRegistry, 12345} terminating
** (DBConnection.OwnershipError) cannot find ownership process for #PID<0.58671.0>.

When using ownership, you must manage connections in one
of the four ways:

* By explicitly checking out a connection
* By explicitly allowing a spawned process
* By running the pool in shared mode
* By using :caller option with allowed process
```

First, we have ownership issues. We've probably all seen issues like this in our test suite. This particular error comes from using the Ecto sandbox adapter.

We would commonly get errors like this in our test suite. These were almost always intermittent errors from flaky tests. There was a very good chance if you ran the test suite again, the test would pass the second time. This was especially true if you only ran the failing tests.

We really needed to improve how we managed processes under test.

Test Data

```
** (Ecto.InvalidChangesetError) could not perform insert because changeset is invalid.

Errors
    %{
      provider_id: [
        {"does not exist",
         [constraint: :foreign, constraint_name: "..."]
        }
      ]
    }
```

Next, we had a long-standing issue of intermittent errors because of missing test data. If you ran the test suite repeatedly, every once in awhile a number of tests would fail because of an issue like this. We had some amount of initially seeded data that all tests relied on, but something in the test suite was messing with it. Depending on the order the tests executed, it was usually fine, but not always.

And there was the broader issue of unrealistic test data. This wasn't causing flaky tests that we knew of, but it did make it harder to find certain bugs. Testing with unrealistic looking data can give you false confidence in your work.

Performance

```
Finished in 162.4 seconds (34.9s async, 127.5s sync)
29 doctests, 4977 tests, 25 failures, 15 excluded
```

Performance was also concerning at times. Unfortunately, I wasn't able to easily get our test suite from 1.5 years ago to run locally and the CI logs are long gone.

I do have recent output though. This is after porting over hundreds of tests. Performance isn't atrocious. I've worked on code bases with test suites way worse than this. Many of you probably have too. It could be better though. Only 34.9s is spent on async tests. The remaining 127.5 seconds is on sync tests. It used to be even worse than that. That implies that huge performance improvements are possible. How much faster is unclear, but it would not surprise me if our test suite could run twice as fast.

So, how did we get started fixing these problems?

The first thing we did was put our head in the sand and plow forward with rebuilding our texting infrastructure. That's what we had been doing and were dutifully continuing.

Unfortunately, our CI builds were failing more often because of these flaky tests. We would just run the build again and it would usually succeed. This was very annoying and was making everybody cranky.

So, we figured, if running the test suite again fixes the issue, then let's just make that part of CI.

```
mix test.all || mix test.all --failed
```

test.all is an alias we use in our test suite to pass all the common flags we want.

This is a terrible sin. Do not do this. This is the equivalent of my nine year old cleaning her room by throwing all of her stuff in the back of her closet where we can't see it. The problem has not been solved. It's only been deferred and it's going to be a much bigger issue later.

This worked for about four months. We successfully shoved the problem to the back of the closet while we continued rebuilding our texting infrastructure.

But the problem was only getting worse. We were introducing more and more flaky tests into our codebase and not realizing it. Running our test suite could have upwards of 80 tests failing at any time. It got to the point that running the test suite again wasn't reliably passing either. So many tests were failing that running only the failures again still had failures.

```
mix test.all || mix test.all --failed || mix test.all --failed
```

So, this is the obvious solution, right? Right...?

We didn't actually do this. I don't think anybody even suggested it. But we could have doubled down on previous bad decisions.

No, what we actually did was to start getting serious about understanding the causes of flaky tests and fixing them. Not all of them, but the worst, most frequent offenders. We were starting to rollout our new texting infrastructure to customers. We had a bit of breathing room to take stock of the situation.

Condemning a Test Suite

In June 2024 we decided to declare tech debt bankruptcy and start using a new test suite. This wasn't going to be a hard cut over. We still had to keep shipping software while we figured out the details. We would keep maintaining the old test suite, but the goal was to shift over to a new test suite for as much new code as possible. If we were doing large updates to code with legacy tests, we would evaluate shifting those tests over to the new test suite.

So, what are the nuts and bolts of actually doing this? Our goal is to run our new test suite with the command `mix test2`, just like we can run the legacy test suite with `mix test`. And we wanted to have very different configuration for these tests, so just putting them in a separate namespace or tagging them in a certain way wasn't a great option.

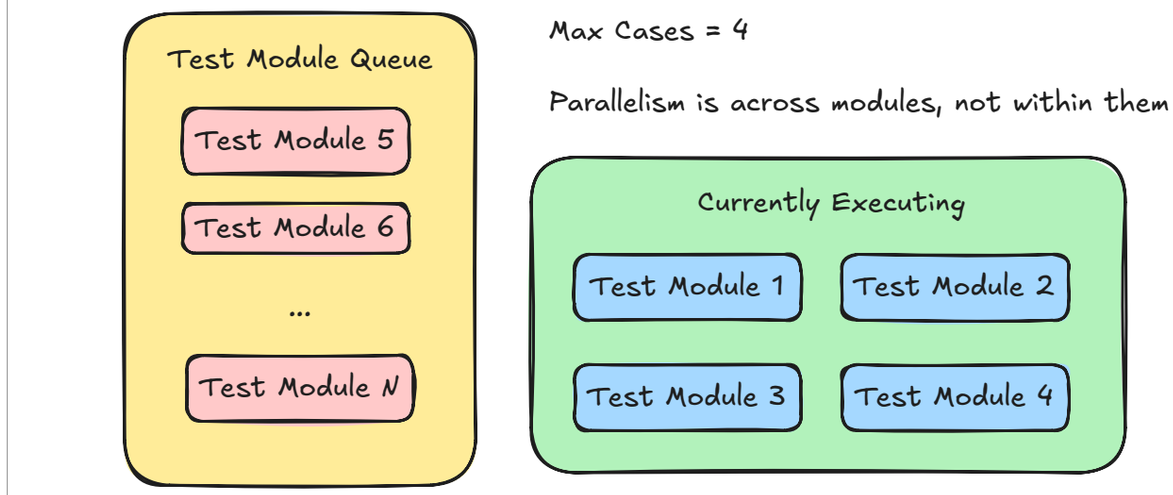
Creating a Second Test Suite

https://github.com/CuriousCurmudgeon/refurbish_test_suite_talk

I've created a sample repo with a new Phoenix 1.8 project that sets up a second test suite. The URL is at the bottom. The QR code will take you there as well. I'll put that same QR code on the final slide, so don't feel like you need to rush to get it now.

Let's take a look at that repo now.

- Show test2 folder
- Show duplicated config
- Show updated .formatter.exs file
- Show changes in mix.exs. This includes elixirc_paths, test_paths, cli function, updating dips, new test2 alias
- Run tests. Use --trace flag to show paths are correct



Let's talk very briefly about how ExUnit runs your tests. There is a `:max_cases` option that specifies the maximum number of tests that can be run in parallel. This defaults to the number of schedulers the BEAM has times two. This machine I'm presenting from has 12 logical cores, so it has 12 schedulers. 12 x 2 = 24, so this will run 24 tests in parallel.

This example show what happens if you pass :max_cases 4. ExUnit is running the four modules on the right in parallel. The modules on the left are queued up to execute.

Importantly, this parallelism is only across modules and only for asynchronous tests. `ex_unit` will never run tests within the same module in parallel and it will never run modules marked synchronous in parallel. This can and should influence how you structure your test modules. By default, we tend to setup one test module per module in your app. You don't have do that though.

Maybe you have a lot of tests for a particular context module. Maybe each of these tests are isolated from each other, so they could run in parallel. And maybe these tests are pretty heavy integration tests. If you logically break these up into separate modules, then `ex_unit` can run them in parallel.

For us, a real world use case for this is campaign setup across countries. You have different compliance frameworks we need to obey. For example, a text campaign in the US has different compliance rules than a campaign in Canada. It logically makes sense for us to break these campaign tests into separate modules for each country. This makes the tests easier to reason about, and has the side benefit of allowing these tests to run in parallel.

Prefer Pure Functions

If your goal is a simple, reliable, and fast test suite, the best thing you can do is lean into pure functions. If your function does not have any side effects, then they can all be async for free. Depending on the complexity of your function, this means that splitting a function's data manipulation from it's side effects can be valuable.

Split Out Side Effects

For example, if you have a function that does a complex calculation and then makes an HTTP call to send the result of that calculation to a 3rd party service, you can simplify your tests a lot by splitting that complex calculation out into a pure function. Inputs in and outputs out. No side effects. If you don't make the HTTP call, you don't need to mock out or stub it in any way.

This is one part of the beauty of Ecto changesets. You can setup your data with a changeset and then just call `apply_changes`. You don't need an external side effect to persist the data to a DB if your test doesn't need it.

Sometimes these wins are obvious, but architecting your app to give yourself more of these opportunities can be a big win.

Adapter Pattern

```
config :getthru, callers_queue_adapter: GetThru.Calls.MockQueueAdapter
```

```
defp caller_state_store do
  Application.get_env(:getthru, :callers_queue_adapter, GetThru.Calls.RedisAdapter)
end
```

We often find ourselves communicating with various dependencies, such as Redis, RabbitMQ, our DB, 3rd party APIs, and more. Some of those can run in your test suite, but they do increase the complexity of setting the test up. We need to setup resources in Redis that the tests expect to exist. We have to setup a wide variety of domain objects in our DB.

For 3rd party dependencies, we do make extensive use of the adapter pattern to mock or stub these dependencies out. But it's so much simpler if you've separated your concerns so you don't even have to. This particular example mocks out Redis in one part of our system with an adapter. We default to the actual implementation, so you only need to override the configuration in tests.

The big caveat is that this only works for unit tests. At some point you do need integration and system tests to holistically test your system. However, if you can cover the business logic of your functions in pure unit tests, then you don't have to test every single permutation in your integration tests.

But, you can't always do this.

Test Data Is Hard

https://blog.appsignal.com/2023/02/28/an-introduction-to-test-factories-and-fixtures-for-elixir.html

So, let's talk about setting up test data

My perspective on setting up test data is very heavily influenced by my experience in the C# world. I have worked in Ruby, but I didn't come into Elixir through that world. I came in through the C# world. And that world loves their DI frameworks to help with testing, but doesn't have a dominant pattern for setting up test data. That means that over almost a decade of consulting, I got to experiment with a lot of patterns.

In almost every case, the client had no coherent philosophy for setting up test data. Little to no thought was given to creating realistic looking test data and everything was done ad-hoc. Depending on the size of your codebase and the complexity of your domain, this might be fine. It wasn't going to cut it for us at GetThru though.

In the Elixir world, there a few major patterns. They are outlined very well in this blog series from AppSignal on test factories and fixtures. https://blog.appsignal.com/2023/02/28/an-introduction-to-test-factories-and-fixtures-for-elixir.html

Good Test Data Is...

- Easy to setup
- Flexible
- Realistic
- But unrealistic when necessary

I'm looking for a few qualities in my test data. First, I want it to be easy to setup. When I start writing new tests, I don't want to write a couple of hundred lines of code to setup the world. Tests don't need to be DRY, but there is a limit to that. I am fine with writing that code once and abstracting it away.

Second, I want it to be flexible. I want to be able to easily create variations of test data. In our domain, one of the core concepts is a campaign. We have two main types of campaigns, talk and text. Each type has many knobs that users turn during setup. I want to be able to easily turn all those knobs when setting up test data.

Third, I want it to be realistic. It's easy to setup unrealistic looking test data. This can hide subtle bugs. Our campaigns have many pieces of data on them. Some fields are mutually exclusive. Some fields only make sense on on talk campaigns, but not text and vice-versa. I want my test data to look like real campaigns.

Finally, I also want to make it unrealistic at times. I need an escape hatch. This is really useful when writing tests to protect against regressions. Bad data got into the system and we need to handle it. In cases like this, I want to be able to setup realistic test data and then break it in some way.

Let's look at how a couple of options stack up to these requirements.

ex_machina

- Easy to setup ✅
- Flexible ✅
- Realistic ❌
- But unrealistic when necessary ✅

Who here uses `ex_machina`? I'm going to guess it's by far the most used way to setup test data in this room and the Elixir ecosystem overall. It's born from the legacy of `FactoryBot` in the Ruby on Rails world, with all the good and bad that comes from that.

First, `ex_machina` is incredibly easy to setup. There is very little configuration to get started. Then you create a factory module with functions to build structs in your domain.

Second, it is very flexible. Your factory functions define default data for a struct. You can pass in a map or keyword list of attributes to override any of those defaults. This works even for nested structs in your schema. It's designed to work with Ecto out of the box, but could work with other persistence libraries too.

In my experience, where `ex_machina` falls down though is in setting up realistic test data. You can absolutely do it, but the path of least resistance encourages you to bypass your context modules and functions. It's a trap. Let's go back to campaigns in our domain as an example.

To get realistic looking campaigns with `ex_machina`, we're left with a couple of options. First, we could try keeping our factory functions in sync with our business context functions. As the domain changes and the rules for creating valid campaigns change, we could try to make similar changes to our factory functions. In practice, this did not work for us. I'm skeptical that it could work for any team. The functions will drift.

Alternatively, you could try calling your business functions inside of your factory methods. However, this probably requires rearchitecting these functions in one way or another. Your context functions probably setup data and persists it to the DB. Now, we previously talked about preferring pure functions. This would be a good use case for that. If you prefer pure functions, then you could have functions that purely operate on changesets. The factory function could use those functions to opereate on changesets and then just call `apply_changes` at the end. That could work.

But at that point, how much benefit are you even getting out of `ex_machina`?

Finally, you do get good support escape hatches. Because each factory function just takes in a list of attributes, you can override whatever you need to and ignore business rules at that point. You're outside of changesets with validation rules for your domain, so all bets are off.

ex_machina

It's possible that we could have used `ex_machina` in a more responsible way, but our experience in the legacy test suite had left a bad test in our mouth. Maybe you can be really careful and diligent with your factory functions. But we were burned too many times by unit tests not realistically testing our code because the data we were using was unrealistic. So, I reached into my bag of tricks from my C# experience and came out with the builder pattern.

Builder Pattern

In the C# world, a builder is a class used to build up a domain object, preferably from the bottom up. For example, we could have a class that builds campaigns. The constructor could initialize a barebones campaign and then methods on the class start adding additional metadata for the campaign. When you've configured it completely, you call a `build` method to turn it into an instance of your domain object.

Each method in the chain returns `this`, so you can chain the method calls together in a fluent interface.

I would create static method wrappers for common use cases. For example, we might want a wrapper that creates a US text campaign with one sender.

User Example

```elixir
def create_user(opts \\ []) do
  role = Keyword.get(opts, :role, :admin)

  {:ok, user} =
    User.registration_changeset(%{
      first_name: Keyword.get(opts, :first_name, "Joe"),
      last_name: Keyword.get(opts, :last_name, "Smith"),
      email: Keyword.get(opts, :email, Faker.Internet.safe_email()),
      phone: Keyword.get(opts, :phone, "800-555-2000"),
      password: Keyword.get(opts, :password, "s3cr3t"),
    })
    |> Repo.insert()

  case role do
    :admin ->
      user
      |> User.admin_changeset(%{
        admin: true,
        permissions: Keyword.get(opts, :permissions, [])
      })
      |> Repo.update!()

    _ ->
      user
  end
end
```

In the C# world, this is all object oriented, but it doesn't have to be. We can easily translate this into functional Elixir code. And the way that opts conventions work with maps and keyword lists in Elixir eliminate the need for many of the repetitive setter methods I would add on the class. The pipeline operator gives us the same feel of a fluent interface in the OO world.

In this example, we're creating a user for our system. This goes through our actual change sets to register a user and set them as the admin. We cannot create invalid test data here. If we change our registration changeset, our test data automatically must respect those changes.

Invitation Example

```elixir
def create_permanent_invitation(account) do
  # Permanent invites are always for the user role
  case Repo.get_by(RolesPermission, role: :user) do
    nil ->
      RolesPermission.changeset(%{
        role: :user,
        permissions: ["view_profile"]
      })
      |> Repo.insert()

    _ ->
      # Do nothing. The user role already exists
      nil
  end

  {:ok, account} = Invitation.create_permanent(account)
  Relay.Repo.get(PlatformInvitation, Account.permanent_invite_id(account))
end
```

The real kicker here is that these builder modules can just call into our context modules under the hood. As much as possible, they're using our actual business logic to setup our test data. This gets us realistic test data pretty much for free.

Builder Pattern

- Easy to setup ✅
- Flexible ✅
- Realistic ✅
- But unrealistic when necessary ✅

Now, the downside here is that this is more code than using `ex_machina`, so you can quibble with the checkmark on Easy To Setup. But it's code that we control and can easily modify as the domain changes. If we change the business rules to create a campaign in our application, our test code automatically respects those rules. It's absolutely more code and effort to get started than ex_machina. Once that hurdle is cleared, then each test is still easy to setup though.

Scenarios

We only recently started into this, but, once again back to my C# days, you can build on top of this pattern with "scenarios". A scenario uses builders to setup common test data in a top down fashion. Builders are bottom up, scenarios are top down.

The best example for us is campaigns. A campaign requires many business objects to exist, such as a user, an organization, a group to contact, contacts for that group, a compliance model the campaign must follow, etc. There is a lot. We can use a scenario to setup that test campaign in one function call.

Many of our tests don't care about the exact particulars of all the data a campaign requires. A scenario can setup that campaign's data with a very simple API. If you need to tweak all those knobs in various ways, you can either use the builders directly or tweak the data after it is inserted into the DB.

You're giving up flexibility in return for a much simpler API to setup common testing scenarios. But if you need that flexibility, you can always fall back to using builders directly

ExUnit Case Templates

This is orthogonal to how you build your test data, but be careful with complex ExUnit case templates. It's very easy to end up with an overly complex, overly configurable case template with a lot of action at a distance through tags. It becomes very hard to reason about these when the surface area is very large.

We made this mistake in multiple places in our legacy test suite. A new setup function would be added to the case template that had a very narrow use. This would be configured with yet more tags. It became a lot of spooky action at a distance that was very difficult to understand.

In our new test suite, we have limited the complexity of our case templates. Our shared data has a setup function to set up a organization with one logged in user. Attributes on those can be overridden with tags. Anything else you want setup in a test should be done in plain functions. You can have setup functions within your test module, but even those are probably going to call into our builder modules or context modules to setup test data.

Faker

One other minor thing related to test data is Faker. I've had good success with Faker in the past and we're starting to use it more in our builders now. It has so many modules to setup various kinds of fake test data. If you need properly formatted test data in various domains, then Faker is great.

Async vs. Sync

https://andrealeopardi.com/posts/async-tests-in-elixir/

Let's dive into asynchronous vs. synchronous tests.

Andrea Leopardi wrote a fantastic breakdown of this in late April. He literally published it a few days after I submitted this talk. The QR code will take you to his blog post. Please read it. It's very, very good. I'm going to give you a worse overview of this topic now.

Singletons Prevent Async Tests

Singletons are the enemy of async tests. If your app has a singleton GenServer, when multiple tests run in parallel, they are all calling the same GenServer process. If one test modifies the state of this GenServer, then it is modified for every other test that calls it after that. If your tests depend on the GenServer being in a specific state, this will cause flaky tests. Depending on the order the tests run in, sometimes they both pass. Sometimes one of them might fail. It's all up to the whims of how `ex_unit` schedules and executes the tests.

Now, one way around this is to mark the test as synchronous. Now `ex_unit` will not attempt to run this test in parallel with any other tests. And if you have a small test suite, this might just be good enough. If your tests are already fast enough, you can mark it as synchronous and move on with your life.

Because Andrea Leopardi already covered it so well, I recommend checking out his blog post. He goes over techniques for eliminating singleton GenServers in your test suite. As one of my co-workers put it

"I wonder how many times we as a programming community will learn that singletons are bad in a multithreaded world 😅"

Clay - GetThru Sr. Software Engineer

Application Environment

```elixir
Application.put_env(:getthru, :telecom_adapter, GetThru.TelecomAdapter.Provider1)

on_exit(fn →
  Application.delete_env(:getthru, :telecom_adapter)
end)
```

Let's talk about other common causes of flaky tests and how we've gone about fixing them.

This seemingly innocent piece of code we used in multiple setup functions was causing ~50% of our flaky tests. We make extensive use of the adapter pattern in our codebase, both for behaviors where we really do need different implementations and for testing purposes.

Anybody see the problem with this code?

These particular tests wanted to use a different implementation of our telecom adapter than other tests and would dutifully clean that up on exit. The problem is that other tests expect that `:telecom_adapter` is always defined. When all tests in this module ran, the final test would run the `on_exit` callback and dutifully delete it from the application environment.

Application Environment

```elixir
Application.put_env(:getthru, :telecom_adapter, GetThru.TelecomAdapter.Provider1)

on_exit(fn →
  Application.put_env(
    :getthru,
    :telecom_adapter,
    GetThru.TelecomAdapter.MockTelecomAdapter
  )
end)
```

That's not what we actually wanted to happen though. We wanted to put the configuration back how it was. Now we don't delete the environment setting in `on_exit`. We put it back to the original value, which we know is always the mock.

Unfortunately, these tests must be synchronous though. We're messing with the application environment. That is a singleton shared across all tests, so we can't run this test module in parallel with any other modules.

Because of that, mess with your application configuration sparingly in tests. Sometimes you need to do it, but it should be rarity in your test suite.

Ecto Sandbox

```
Postgrex.Protocol (#PID<0.646.0>) disconnected: **
(DBConnection.ConnectionError) client #PID<0.16066.0> exited
```

```
16:53:33.723 [error] GenServer {GetThru.CampaignRegistry, 12345} terminating
** (DBConnection.OwnershipError) cannot find ownership process for #PID<0.58671.0>.

When using ownership, you must manage connections in one
of the four ways:

* By explicitly checking out a connection
* By explicitly allowing a spawned process
* By running the pool in shared mode
* By using :caller option with allowed process
```

Another common test issue is usage of the Ecto sandbox. This might or might not cause test failures, but if your test logs are full of messages like these, then there is a good chance that you're not using the Ecto sandbox correctly.

Stepping back first, what is the Ecto sandbox? You've probably used it a bunch, but may not have ever looked into what it's actually doing for you.

```elixir
setup tags do
  pid = Ecto.Adapters.SQL.Sandbox.start_owner!(MyApp.Repo, shared: not tags[:async])
  on_exit(fn -> Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
  :ok
end
```

This is the standard setup code you'll find in the Ecto sandbox docs for start_owner. Let's walk through what it does.

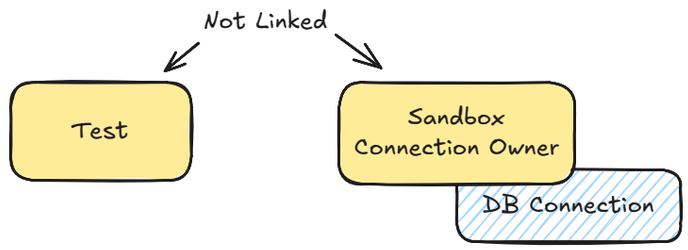
At a high level, the Ecto sandbox is a connection pool used for concurrent transactional tests. At the beginning of every test, you check out a connection that the test process will use. This connection will do everything within a transaction. start_owner starts a new process that checks out and owns this connection and returns that process's pid.

That process is not linked to the caller, so you must call stop_owner when your test exits to stop the process. This causes this connection to rollback the transaction so your tests do not trash each other's data. This is how you can test against a singleton DB concurrently. If the transaction gets rolled back every time, then tests never see the uncommitted data from other tests.

```elixir
setup tags do
  pid = Ecto.Adapters.SQL.Sandbox.start_owner!(MyApp.Repo, shared: not tags[:async])
  on_exit(fn -> Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
  :ok
end
```

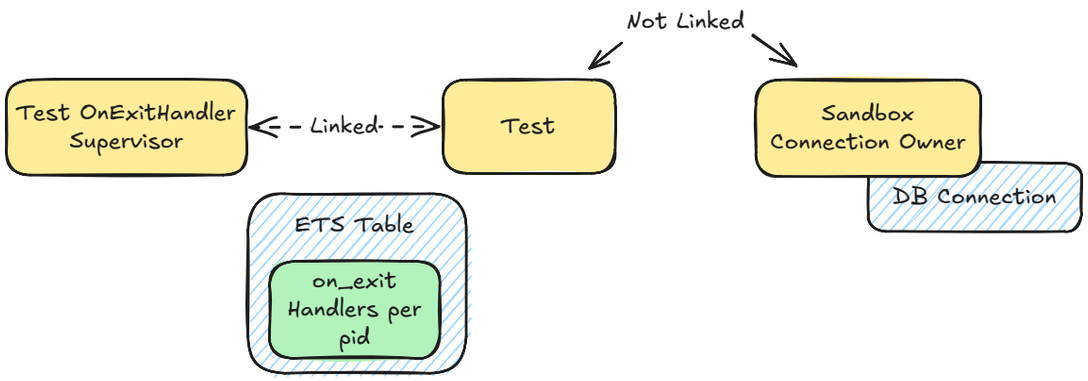
Let's try to visualize how all of this fits together. We start with a our lonely test process. In ExUnit, there are a lot of other processes running to, but we're not going to worry about those.

```elixir
setup tags do
  pid = Ecto.Adapters.SQL.Sandbox.start_owner!(MyApp.Repo, shared: not tags[:async])
  on_exit(fn -> Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
  :ok
end
```



When that test starts, it first executes the setup callbacks in order. Let's say this is our only setup callback. We call `start_owner!` on the Ecto sandbox. This starts another process that is not linked. This process gets the sandbox DB connection that be used for this test. `start_owner!` returns the pid of the process that owns this connection.

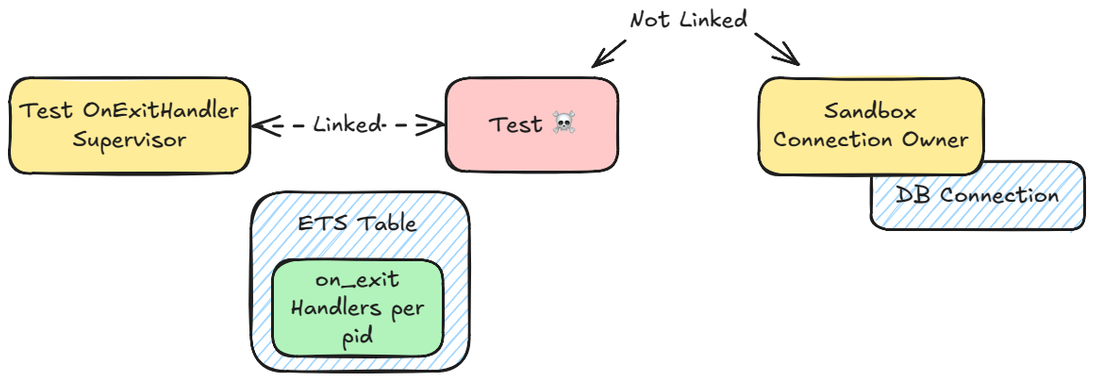
```elixir
setup tags do
  pid = Ecto.Adapters.SQL.Sandbox.start_owner!(MyApp.Repo, shared: not tags[:async])
  on_exit(fn -> Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
  :ok
end
```



Next, we register an `on_exit` callback that calls `stop_owner`. This will execute when the test process exits. This is important. The test process will have already exited at this point. This is why `start_owner!` starts a process that is not linked and why we must make sure to clean it up in the `on_exit` callback.

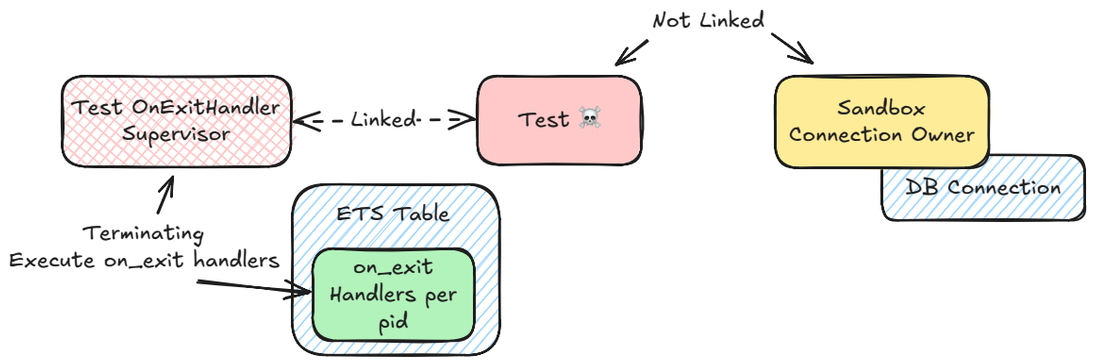
This is registered internally in an ETS table. The key is the pid of the test process and the value is a list of `on_exit` callbacks to execute when the test finishes.

```elixir
setup tags do
  pid = Ecto.Adapters.SQL.Sandbox.start_owner!(MyApp.Repo, shared: not tags[:async])
  on_exit(fn -> Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
  :ok
end
```



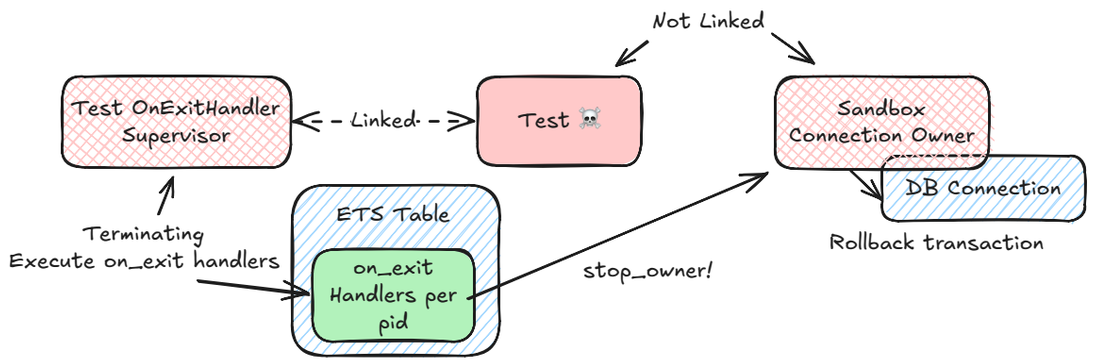
Now our test has finished. The test process has ended. ExUnit's OnExitHandler supervisor gets notified of this since the process's were linked and starts running the registered on_exit callbacks.

```elixir
setup tags do
  pid = Ecto.Adapters.SQL.Sandbox.start_owner!(MyApp.Repo, shared: not tags[:async])
  on_exit(fn -> Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
  :ok
end
```



We've registered one on_exit handler that calls `stop_owner`. That executes now.

```elixir
setup tags do
  pid = Ecto.Adapters.SQL.Sandbox.start_owner!(MyApp.Repo, shared: not tags[:async])
  on_exit(fn -> Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
  :ok
end
```



And it stops the sandbox connection owner process. This rolls back the transaction so the DB state is back where we started.

We won't start another test process until the OnExitHandler supervisor finishes cleaning up.

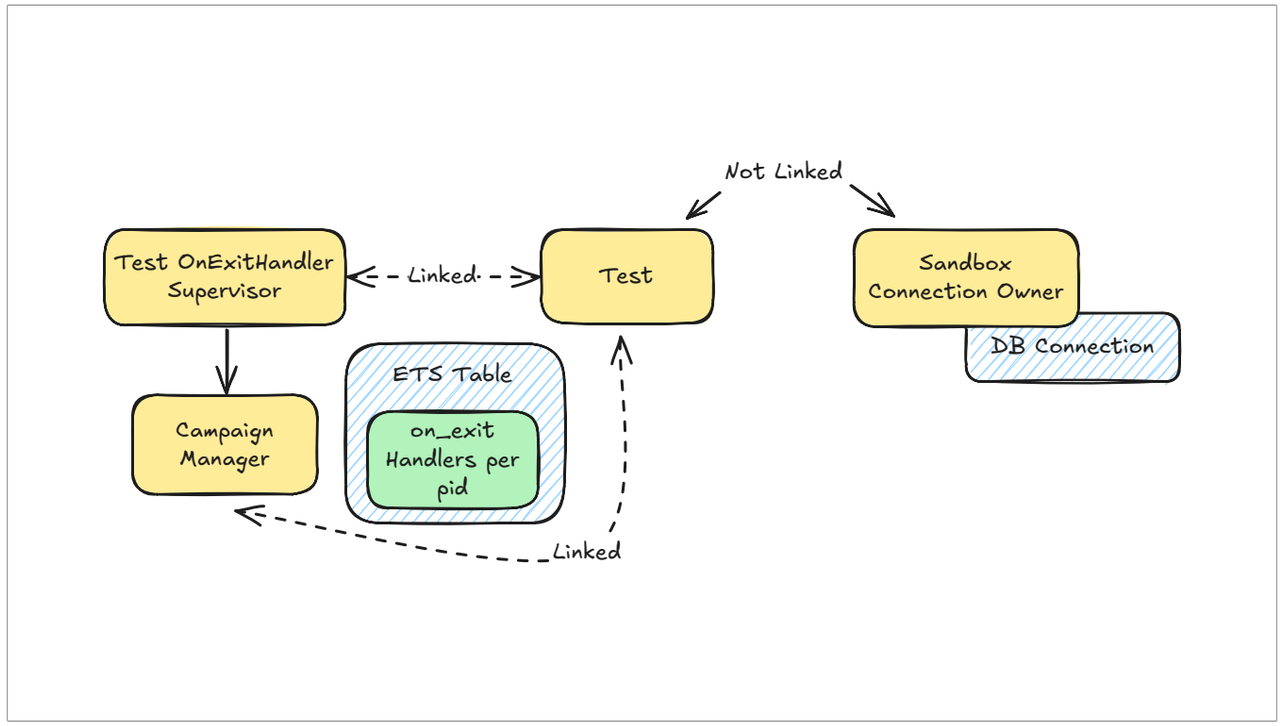
```elixir
pid =
  start_link_supervised!({
    CampaignManager,
    [[ets_table: ets_table_name]]
  })

Sandbox.allow(Repo, self(), pid)
```

Keeping that in mind, let's look at how we can allow other processes our tests may start to use this same Ecto sandbox connection.

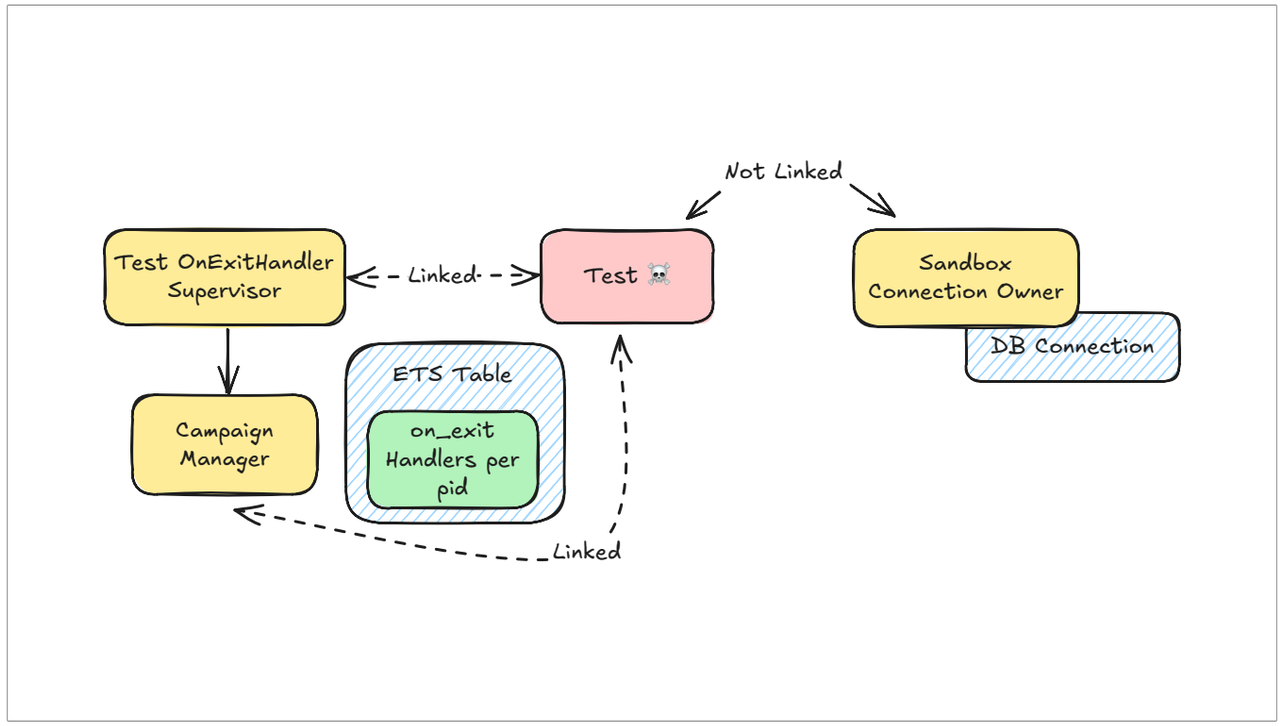
This code is from a test in our system. It starts a CampaignManager process that also needs a connection to talk to the DB. We want this to use the same sandbox connection. We let the Ecto sandbox know that this process should use the same connection with `allow/3`.

The first thing we do is start it with `start_link_supervised!`. You could use `start_supervised!` too. Link will take down the test process if other process exits. This returns the pid of the new process. This new process is not started under the test process. It's started under the test process supervisor. When the test exits, the test process supervisor will guarantee that this process exits before the next test starts.

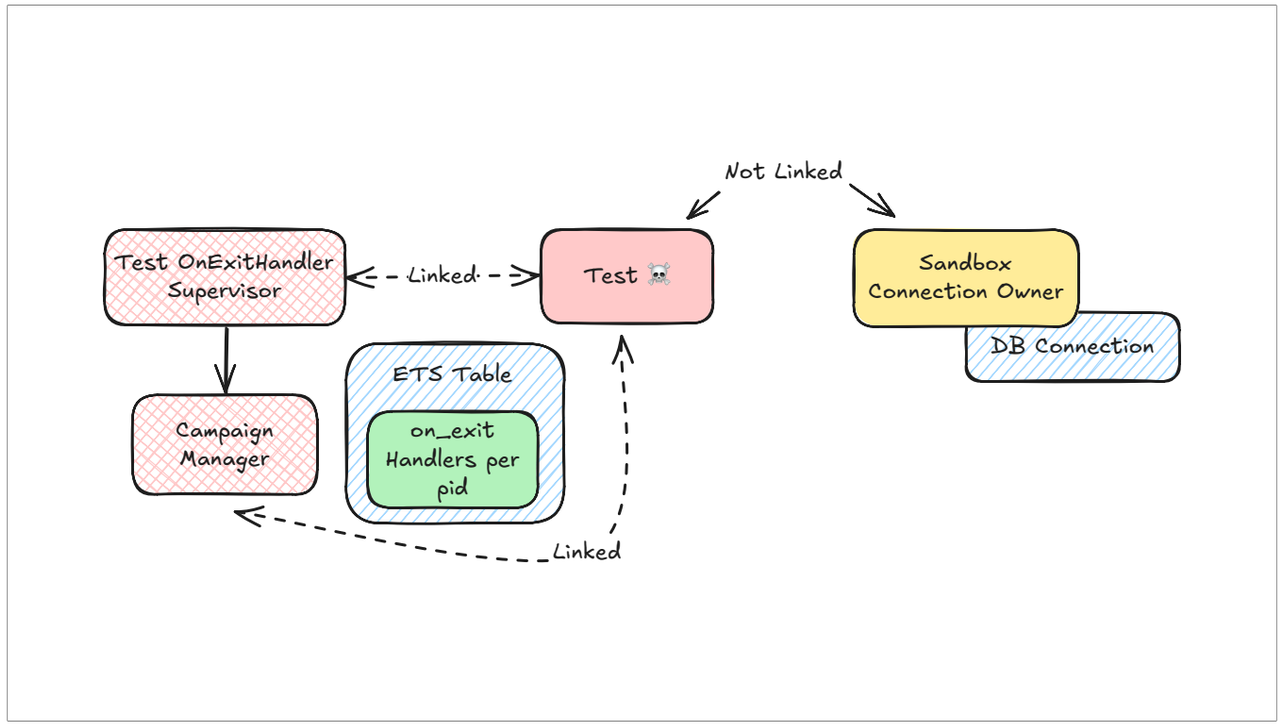
We then pass that pid to `Sandbox.allow/3`. We're saying the sandbox should allow the pid to use the same connection on Repo that self() uses.



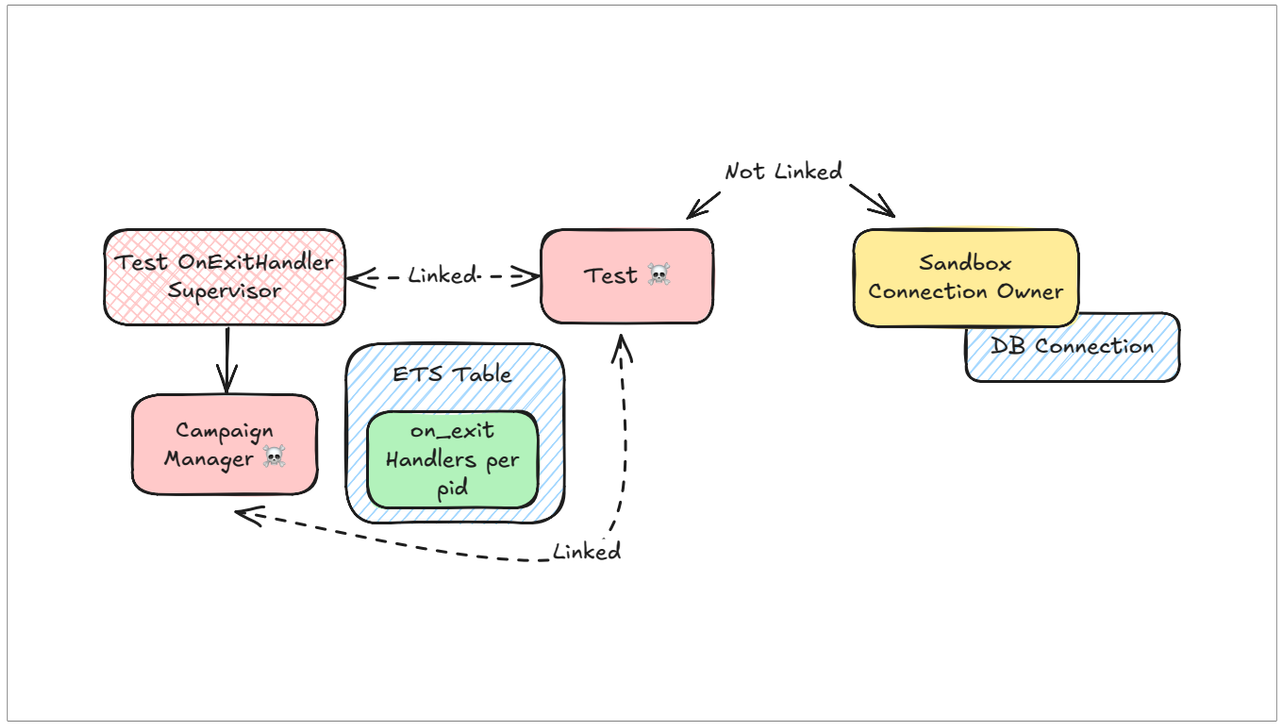
Going back to our diagram, it now looks like this. Using `start_supervised` or `start_link_supervised` starts processes under the `OnExitHandler` supervisor.



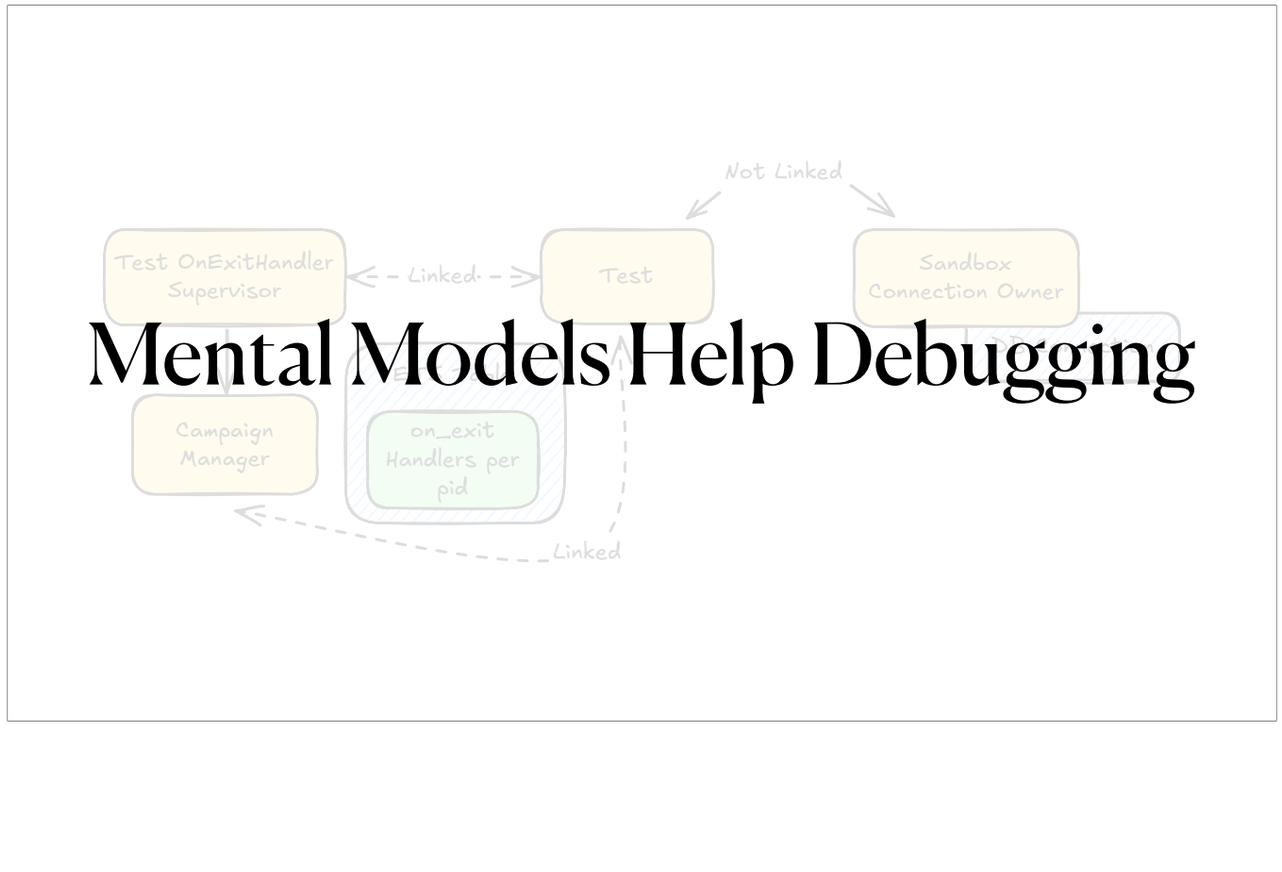
Similar to before, when the test process exits, the `OnExitHandler` supervisor will also start to tear down.



And it's a supervisor, so it will cleanly shut down its children first, ensuring that our CampaignManager process is cleaned up before the next test executes.



From here, the on_exit handler would execute as before to clean up the sandbox connection owner



Mental Models Help Debugging

When I started this talk, I didn't expect to go this deep into ExUnit internals, but I found it very valuable. Debugging flaky asynchronous tests can be hard. Having a good mental model of how everything is actually fitting together under the hood really helps when you don't understand why a test is failing intermittently. Being able to step back and slot the processes my particular test is using into these diagrams is really useful.

These types of Ecto sandbox and, more broadly, process ownership errors, are all over the place in the logs in our legacy test suite and a couple have creeped into test2 as well. We've mostly squashed them as they come up, but I think there is at least one still lingering. It's real easy just to re-run failed CI builds when you just want to ship a PR instead of taking the time to dive into what causes a flaky test.

Mocking

German Velasco

Let's build great software together.

Are you mocking me?

Friday 3pm

I didn't talk at all about mocking today. We use mox and are pretty happy with.

We don't have time to do the topic justice though, so I'm going to point you to check out German Valasco's talk on Friday at 3pm. He's going to compare many different mocking frameworks. If you've missed it, I think recordings are available for paid attendees of ElixirConf immediately. If not, it will be up on YouTube at some point.

Wrap Up

- It can be better
- Have clear goals
- Have a plan
- Have good mental models

To wrap up, your test suite can be better. Just because it's been bad doesn't mean it has to stay bad. You have options. We decided to split out a completely separate test suite, but you don't have to take that route.

But, no matter what route you're taking, make sure you understand what your goals are. What exactly don't you like about your test suite? Point to examples of patterns you find difficult to manage, test data causing issues with tests, problems with flakiness. And have a plan to reach those goals. I think this is a fantastic use case for SMART goals.

This will take time, you need to decide how much time and effort you're willing to put into it. It will feel worse before it gets better in some cases. You won't be coasting on the inertia of your legacy test suite. Yeah, it may be bad, but it's a familiar kind of bad. You're used to it. This is going to be jarring at times and you'll wonder if it's worth it. It is.

And finally, have a good mental model of how your tests execute. This is very useful when debugging intermittent issues. It's often correctly said that you can get very far in Elixir without understanding OTP. I do not believe that you can write a performant and reliable tests suite without understanding OTP though.

https://github.com/CuriousCurmudgeon/refurbish_test_suite_talk

Brian Meeker

Blog: https://brianmeeker.me
GitHub: @CuriousCurmudgeon
Bluesky: @brianmeeker.bsky.social

Once again, my name is Brian Meeker you can find the slides and the sample repo with a second test suite on GitHub. The QR code will take you there.

Thank you so much for your time. Thanks to the conference organizers for putting all of this together. Enjoy the rest of ElixirConf.