The very first year working on a full time Elixir job can be challenging. Libraries are new, syntax is new, the way to code is new. On this talk Vinny will share his experience on his first Elixir job and how he could get up to the speed by testing better.

Test to be Fast ⚡

How to Get Productive in Elixir

Who am I

Vinicius Negrisolo || Vinny

https://github.com/vnegrisolo

Java |> Ruby |> Elixir

The Dream

- Functional
- Immutability
- Compiled
- Performance
- Pattern Matching
- Fault Tolerance
- Concurrent
- Simpler than OO
- ...

 for Elixir

The Saga begins

Documentation / Tutorials

Blog posts (Elixir Radar / ElixirWeekly)

Meetups

Conferences

Small apps and libraries

The Job

Chat && Order

Restaurant |> Suppliers

Tech Overview

- 2.5 yo production app
- single git repo
- umbrella app (≃15)
- distillery releases (≃4)
- CI with auto deploy
- docker containers
- 3x AWS medium size

Team Velocity

- backend team is slow
- witch hunt

Test Feedback Cycle

- defines development speed
- ⚡ fast
- ✓ reliable

Test Situation

- test suite
  - brittle / intermittent / gaps
  - 11 min to run / all sequential
- manual tests on staging
- Continuous Integration
  - rerun if test fails? really?

How to revert this scenario?

- test
  - remove brittleness
  - fill the gaps
  - speed and parallelism

Test Data

=> The Big Win!

Test data - Situation

- module attrs with fixed values
- setup to build data
- attrs, setup and assertions were far apart

Test data
Code Situation

```
@attrs %{
  email: "user@mail.com",
  full_name: "Billy Bob",
  age: 30,
  gender: :male
}

setup do
  user = @attrs |> User.new() |> Repo.insert!()
  [user: user]
end

test "do something", %{user: user} do
  ...
  assert page.user_full_name == user.full_name
end
```

Test data - Factory

- Keep it simple
- A single factory definition per schema
- Random data all the time
- Build all regular attributes
- Build all belongs_to relation
- Everything else defined on tests

Test data - Factory Sample

```
defmodule Factory do
  use MapBot

  def new(User) do
    %User{
      email: &"#{&1}_#{Faker.Internet.email()}" ,
      full_name: Faker.Name.name(),
      age: 15 + :rand.uniform(100),
      gender: Enum.random(~w(male female)a)
    }
  end
end
```

```
# usage
user = Factory.build(User,  age: 17)
%User{} = Factory.insert!(User,  age: 21)
```

Test data - Factory Benefits

- no database cycles
- no need to open factory definitions
- no assertions on hardcoded factory values
- schema changes |> factory changes
- new test scenarios |> no factory changes

Unit Testing

Unit Testing

- single level of describe
- describe/test/assert/refute
- async false by default
- async by test file

Unit Testing

```elixir
defmodule MyApp.Ecto.NormalizerTest do
  use ExUnit.Case, async: true
  alias MyApp.Ecto.Normalizer

  describe "trim/2" do
    test "trims a valid changeset for single atom" do
      attrs = %{name: " Bob ", email: "bob@mail.com"}
      changeset = attrs |> changeset() |> Normalizer.trim( :name)

      assert changeset.changes.name == "Bob"
    end
  end

  defp changeset(attrs) do
    ...
  end
end
```

Unit Testing - Pattern Matching

- Left ⇔ Right
- = ⇔ ==
- ^ pin operator

Unit Testing - Pattern Matching

Left ⇔ Right && = ⇔ ==

```
assert changes == %{name:  "Bob"}
```

```
assert %{name: "Bob"} = changes
```

^ pin operator

```
assert changes == %{name: full_name}
```

```
assert %{name: ^full_name} = changes
```

^ existing variable

```
assert changes == %{name: user.name}
```

```
name = user.name
assert %{name: ^name} = changes
```

Unit Testing - Data Table

- controlled input/output
- scale up tests

Unit Testing Data Table

```elixir
defmodule MyApp.Ecto.Type.MoneyTest do
  use ExUnit.Case, async: true
  alias MyApp.Ecto.Type.Money

  @cast_data [
    {nil, {:ok, nil}},
    {"314.59", {:ok, 314.59}},
    {314.59, {:ok, 314.59}}
  ]

  describe "cast/1" do
    for {value, expected} <- @cast_data do
      @value value
      @expected expected

      test "casts '#{inspect(@value)}'" do
        assert Money.cast( @value) == @expected
      end
    end
  end
end
```

Unit Testing - Properties

- inputs are random and abundant
- catches some unthinkable edge cases

Unit Testing Properties

```elixir
defmodule MyApp.Ecto.Type.MoneyTest do
  use ExUnit.Case, async: true
  use ExUnitProperties
  alias MyApp.Ecto.Type.Money

  describe "cast/1, dump/1 and load/1" do
    property "cast/1, dump/1, load/1 binary dollars+cents" do
      check all dollars <- integer(),
                cents <- 0..99 |> integer() do
        value = "#{dollars}.#{cents}"
        {expected, ""} = Float.parse(value)

        assert {:ok, value} = Money.cast(value)
        assert {:ok, value} = Money.dump(value)
        assert Money.load(value) == {:ok, expected}
      end
    end
  end
end
```

Special Test Cases

Controller / WebSocket / Feature

Controller Testing

Controller Testing

- `use MyAppWeb.ConnCase`
- exposes `Plug.Conn`
- http request/response

Controller Testing

```
defmodule MyAppWeb.PostControllerTest do
  use MyAppWeb.ConnCase, async: true

  describe "show/2 when the user is authenticated" do
    setup [:authenticated_conn_setup]

    test "renders show page", %{conn: conn, current_user: user} do
      post = insert!(Post, user: user)
      conn = get(conn, Routes.post_path(conn, :show, post))

      assert response = html_response(conn, 200)
      assert response =~ "Post"
    end
  end
end
```

WebSocket Testing

WebSocket Testing

- `use MyAppWeb.ChannelCase`
- exposes `Phoenix.Socket`
- channel join/push/reply/broadcast
- joined channels runs on new pids
- ecto sandbox allow for async

WebSocket Testing

```elixir
defmodule MyAppWeb.UserChannelTest do
  use MyAppWeb.ChannelCase, async: true

  describe "handle_in/3 for list-posts when joined on user:lobby" do
    setup [:authenticated_user_socket_setup, :join_user_lobby_setup]

    test "returns all user posts", %{socket: socket, current_user: user} do
      post = insert!(Post, user: user)
      ref = push(socket, "list-posts")

      assert_reply(ref, :ok, response, 500)
      assert response == %{posts: [%{name: post.name}]}
    end
  end

  def join_user_lobby_setup(%{socket: socket}) do
    {:ok, _reply, socket} = subscribe_and_join(socket, UserChannel, "user:lobby")
    Ecto.Adapters.SQL.Sandbox.allow(MyApp.Repo, self(), socket.channel_pid)
    [socket: socket]
  end
end
```

Feature Testing

Feature Testing

- covers all the layers
- slower, but not slow
- can be parallel
- `wallaby` / `hound`

Feature Testing

```elixir
defmodule MyAppWeb.Features.PostTest do
  use MyAppWeb.FeatureCase, async: true
  import Wallaby.Query, only: [css: 1, link: 1]

  describe "when the user is authenticated" do
    setup [:authenticated_session_setup]

    test "posts page", %{session: session, current_user: user} do
      post = insert!(Post, user: user)
      session = visit(session, "/posts")

      assert "Posts" in texts_by(session, css("h1"))
      assert "Name" in texts_by(session, css("table th"))
      assert post.name in texts_by(session, css("table td"))

      session = click(session, link(post.name))

      assert current_path(session) == "/posts/#{post.id}"
    end
  end
end
```

Almost Achieved

Almost Achieved

- Parallel tests && CouchDB
  - New temporary DB every test
  - DB url defined on Application.put_env/4 => sequential
- Could use Ecto Sandbox idea
  - GenServer => %{pid => DB url}

Achievements

Achievements

- Not 100% perfect, but way better
- I learned more && more
- I got up to speed
- Faster to produce test && code
- Tests more reliable and faster
- No more team slowness “feeling”
- Oftener and smaller deploys

Don’t underestimate how tests can help you

Test to be Fast ⚡

Thanks! / Questions?