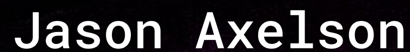


Choosing an Effective Testing Structure

Practical tips for Elixir and Erlang engineers

About me

CODE BEAM AMERICA

- Been using Elixir since 2016
- Libraries I help maintain:
  MainProxy, Scenic, ExSync, DataTracer, PasswordValidator
- Senior Software engineer at Felt

Me on the Internet

- Mastodon: @axelson@fosstodon.org

Why talk about testing?

CODE BEAM AMERICA

- Testing is an important piece of writing and especially *maintaining* software
- I wish there were more talks about testing in practice

What are we talking about today?

CODE BEAM AMERICA

- Writing Tests
- Running Tests
- Anti-patterns
- Libraries

Warning! Opinions Ahead!

Writing Tests

Writing Tests

Running Tests

Anti-patterns

Libraries

Write tests to fail

Write tests to fail

Consider this test

```
test "my test" do
  user = admin_user()
  attrs = %{name: "123"}
  changeset = MyApp.update_user(user, attrs)
  assert changeset.valid?
end
```

What happens when this test fails?

Womp womp

```
1) test my test (SimpleTest)
   test/simple_test.exs:4
   Expected truthy, got false
   code: assert changeset.valid?
   stacktrace:
     test/simple_test.exs:7: (test)
```

Write tests to fail

CODE BEAM AMERICA

Consider this test

```
test "my test" do
  user = admin_user()
  attrs = %{name: "123"}
  changeset = MyApp.update_user(user, attrs)
  assert changeset.valid?
end
```

How can we rewrite this test for failure?

Write tests to fail

CODE BEAM AMERICA

```elixir
test "my better test" do
  user = admin_user()
  attrs = %{name: "123"}
  changeset = MyApp.update_user(user, attrs)

  assert errors_on(changeset) == %{}
  assert changeset.valid?
end
```

A successful failed test!

```
1) test my better test (SimpleTest)
   test/simple_test.exs:11
   Assertion with == failed
   code:  assert errors_on(changeset) == []
   left:  %{email: ["can't be blank"]}
   right: []
   stacktrace:
     test/simple_test.exs:15: (test)
```

Put expected values on the right side of assertions

Put expected values on the right side of assertions

`assert actual == expected`

```
actual = MyApp.calc(2, 2)
assert actual == 4
```

- Consistent pattern helps readability
- Read the test from top to bottom, left to right

Put expected values on the right side of assertions

CODE BEAM AMERICA

```
test "bad" do
  post = create_post(%{name: "on testing"})
  post_id = post.id

  assert {:ok, %{id: ^post_id, name: "on
testing", author: "joe"}} =
           MyApp.fetch_last_post()
end
```

Put expected values on the right side of assertions

CODE BEAM AMERICA

```
test "better" do
  post = create_post(%{name: "on testing"})

  assert MyApp.fetch_last_post() ==
           {:ok, %Post{id: post.id, name: "on
testing", author: "joe"}}
end
```

Assertions with Machete

CODE BEAM AMERICA

Machete github.com/mtrudel/machete

```
test "with machete" do
  post = create_post(%{name: "on testing"})

  assert MyApp.fetch_last_post() ~>
           {:ok, superset(%{name: "on testing"})}
end
```

Machete

CODE BEAM AMERICA

Machete allows you to:

- Keep your assertions on the right-hand side
- Not need intermediate values just for pattern matching
- Use flexible matchers like
  - `integer(min: 10, max: 25)`
  - `string(matches: ~r/abc/)`
  - `json(%{a: 1})`

Use async tests

Why write async tests?

CODE BEAM AMERICA

- Faster test suite execution
- It helps ensure that you understand your system
- Catch race conditions
- It's... fun?

What prevents async tests?

CODE BEAM AMERICA

- Forgetting to add `async: true` to your tests
- Modifying global state
  - Application environment
  - ETS
  - Database
  - Process registry
  - plus many others!

Bypass library

CODE BEAM AMERICA

github.com/PSPDFKit-labs/bypass
allows you to mock an HTTP server and return prebaked responses

```
test "client can handle an error response" do
   bypass = Bypass.open()

   Bypass.expect_once(bypass, "POST", "/", fn conn ->
     Plug.Conn.resp(conn, 429, "Rate limit exceeded")
   end)

   assert HttpClient.post("http://localhost:#{bypass.port}/", "Hello World!") ==
     {:error, :rate_limited}
 end
```

Dealing with the Application environment

CODE BEAM AMERICA

How would you test this code using a bypass server?

```
def send_http_request(path, body) do
  base_url = Application.get_env(:my_app, :base_url)

  Req.post(base_url <> path, body)
end
```

You can't use `Application.put_env/2` because it modifies global state

Dealing with the Application environment

Add a parameter to pass in the base_url

```
def send_http_request(path, body, base_url \\ nil) do
  base_url = base_url || Application.get_env(:my_app, :base_url)

  Req.post(base_url <> path, body)
end
```

That works…

But what if this function is called deep within your code?

Enter ProcessTree

github.com/jbsf2/process-tree

A library for avoiding global state in Elixir applications

ProcessTree

CODE BEAM AMERICA

ProcessTree lets you mimic the Application Environment

- It looks up values in the Process dictionary of the current process
- Then in any `$ancestors`
- Then in any `$callers`

ProcessTree

CODE BEAM AMERICA

```
def send_http_request(path, body) do
  base_url = ProcessTree.get(:base_url, @default_base_url)
  Req.post(base_url <> path, body)
end

test "with process tree" do
  bypass = Bypass.open()
  Process.put(:base_url, "http://localhost:#{bypass.port}")
  MyApp.send_http_request("/register", "abc")
end
```

ProcessTree wrapper

CODE BEAM AMERICA

```
defmodule AppEnv do
  if Mix.env() == :test do
    def get(key) do
      ProcessTree.get(key, default_value(key))
    end
  else
    def get(key), do: default_value(key)
  end

  if Mix.env() == :test do
    def put(key, value), do: Process.put(key, value)
  end

  defp default_value(key), do: Application.get_env(:my_app, key)
end
```

Verify your tests by changing code

CODE BEAM AMERICA

- If you modify your code to have bugs, does your test suite fail?
  - If not, then you're missing a test
- This is formalized in an approach called mutation testing
  - https://devonestes.com/announcing_muzak

Hat tip to Jeffrey Matthias for the idea!

Tip: Don't depend on factories

CODE BEAM AMERICA

- Write your tests to not implicitly depend on factories
  - Makes your tests more reliable
- Otherwise changes to the factory could break unrelated tests

Tip: Don't depend on factories

CODE BEAM AMERICA

```
test "bad" do
  user = create_user(age: 25)

  assert {:ok, updated_user} =
           MyApp.update_user(user, age: 30)

  assert updated_user == %User{
           name: "Joe",
           age: 30
         }
end
```

Tip: Don't depend on factories

CODE BEAM AMERICA

```
test "good" do
  user = create_user(age: 25)

  assert {:ok, updated_user} =
           MyApp.update_user(user, age: 30)

  assert updated_user.age == 30
end
```

Faker can help catch this github.com/elixirs/faker

Tip: Use @tmp_dir tag

ExUnit gives you an `@tag :tmp_dir`

```
@tag :tmp_dir
test "Use temp directory", %{tmp_dir: tmp_dir} do
  assert File.dir?(tmp_dir) == true
end
```

Directory is cleared before every test

Hat tip to Lars Wikman for the reminder!

Tip: Use custom tags for in setup blocks

CODE BEAM AMERICA

```
@tag admin?: true
test "admins can delete posts", %{user: admin} do
  ...
  assert can_delete?(post, admin) == true
end

@tag admin?: false
test "non-admins cannot delete posts", %{user: user} do
  ...
  assert can_delete?(post, user) == false
end
```

Tip: Use custom tags in setup blocks

CODE BEAM AMERICA

```
setup context do
  admin? = context[:admin?]
  user = create_user(%{}, admin?: admin?)
  %{user: user}
end


@tag admin?: true
test "admins can delete posts", %{user: admin} do
  ...
end
```

Recommendation: use sparingly (and only for the primary entity)

Given / When / Then

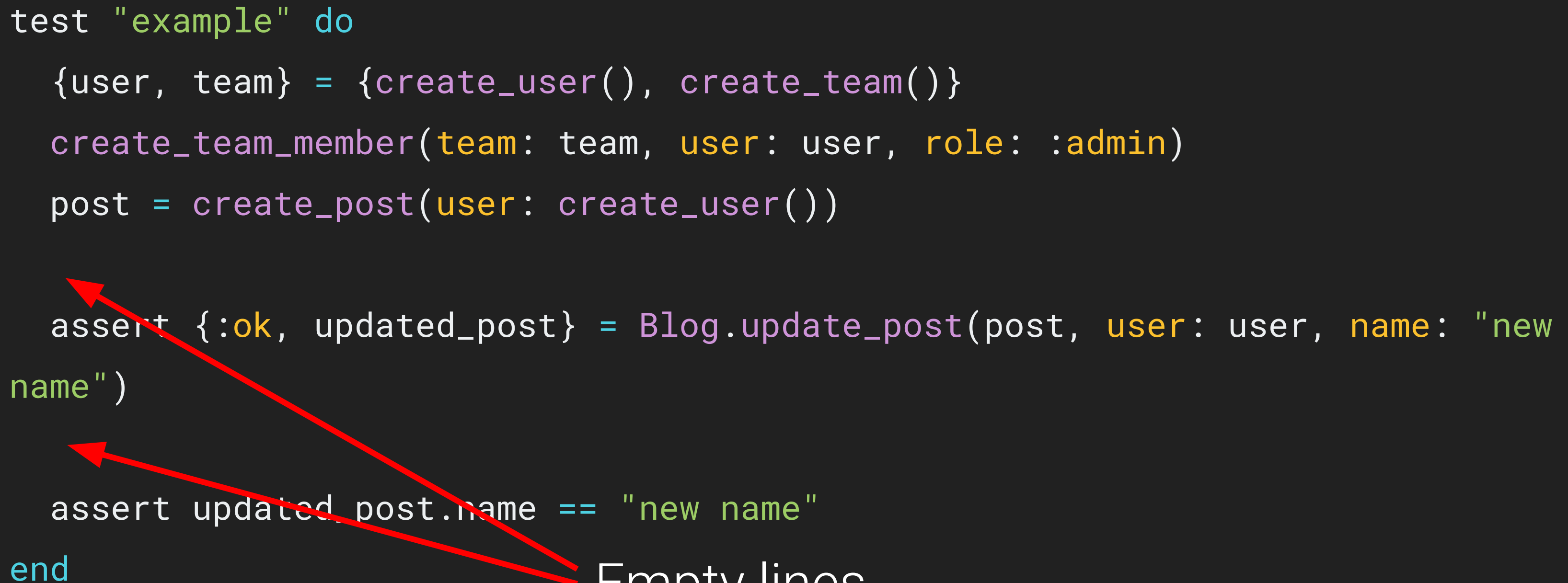
CODE BEAM AMERICA

```
test "example" do
  # GIVEN an admin user
  {user, team} = {create_user(), create_team()}
  create_team_member(team: team, user: user, role: :admin)
  post = create_post(user: create_user())

  # WHEN they update other's posts
  assert {:ok, updated_post} = Blog.update_post(post, user: user, name: "new name")

  # THEN the update succeed
  assert updated_post.name == "new name"
end
```

martinfowler.com/bliki/GivenWhenThen.html

Given / When / Then

CODE BEAM AMERICA



martinfowler.com/bliki/GivenWhenThen.html

Running your tests

- Writing Tests
- Running Tests
- Anti-patterns
- Libraries

mix test.watch

CODE BEAM AMERICA

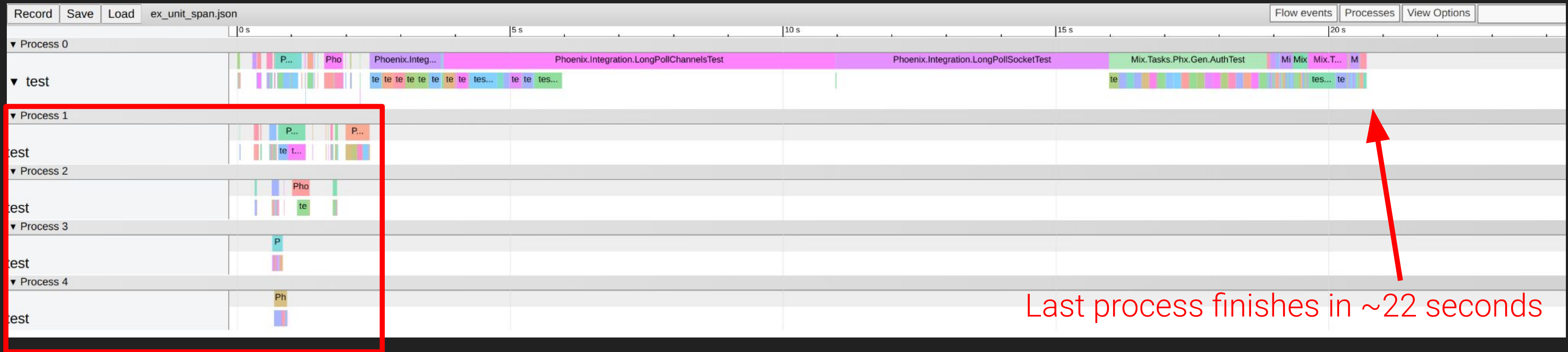
- github.com/lpil/mix-test.watch
- Allows you to run `mix test.watch [pattern]`
  - When you change your source code, the test re-runs
  - "It just works!" -someone probably

ExUnitSpan visualizes your test suite timing

CODE BEAM AMERICA

ExUnitSpan github.com/ananthakumaran/ex_unit_span

Produces a trace of your tests, helps to visualize test concurrency



First 4 processes finish in ~2 seconds

Run tests from your editor

CODE BEAM AMERICA

- There's many extensions for this
  - Emacs: github.com/ananthakumaran/exunit.el
  - Vim: github.com/vim-test/vim-test
  - VSCode: github.com/samuelpordeus/vscode-elixir-test
- It saves time over copying the file name and line numbers manually

Tip: Run a specific single test

CODE BEAM AMERICA

- There's two ways to run a single test
- Option 1: by line number
  - `mix test test/some_test.exs:12`
  - This runs only the test on line 12

Tip: Run a specific single test

- Option 2: by test name
  - given this test:

```
test "admins can delete others posts" do
  ...
end
```

  - run `mix test --only 'test:test admins can delete others posts'`

tag

test name

CODE BEAM AMERICA

Anti-patterns in Testing

Even more opinions ahead!

- Writing Tests
- Running Tests
- Anti-patterns
- Libraries

Anti-patterns in Testing

⚠️ Opinions ahead! ⚠️

Slavishly chasing 100% test coverage

- Aiming for 100% test coverage encourages the wrong behaviors
  - The aim becomes test coverage instead of effective tests
  - Your test suite becomes brittle

1 Assertion Per Test (1APT)

CODE BEAM AMERICA

1APT is when every test case includes **exactly one** assertion

ANTI-PATTERNS

1 Assertion Per Test (1APT)

CODE BEAM AMERICA

```
test "logs the user out and redirects to /", %{conn: conn, user: user} do
  conn = conn |> log_in_user(user) |> delete(~p"session")
  assert redirected_to(conn) == "/"
end

test "logs the user out and removes the token", %{conn: conn, user: user} do
  conn = conn |> log_in_user(user) |> delete(~p"session")
  refute get_session(conn, :user_token)
end

test "logs the user out and shows a flash message", %{conn: conn, user: user} do
  conn = conn |> log_in_user(user) |> delete(~p"session")
  assert get_flash(conn, :info) =~ "Logged out successfully"
end
```

1 Assertion Per Test (1APT)

CODE BEAM AMERICA

Instead you can write it in a single test

```
test "logs the user out", %{conn: conn, user: user} do
  conn = conn |> log_in_user(user) |> delete(~p"session")

  assert redirected_to(conn) == "/"
  refute get_session(conn, :user_token)
  assert get_flash(conn, :info) =~ "Logged out successfully"
end
```

Using meck-based libraries

- Meck-based libraries:
  - meck, mock, patch, espec
- Meck works by replacing modules
- Not `async: true` friendly
- Replacing some modules creates *very* hard to track down bugs in the test suite

Testing Libraries to be aware of

Writing Tests

Running Tests

Anti-patterns

Libraries

Previously mentioned

- Bypass github.com/PSPDFKit-labs/bypass
- ProcessTree github.com/jbsf2/process-tree
- Machete github.com/mtrudel/machete
- mix test.watch github.com/lpil/mix-test.watch

Parameterized Test

ParameterizedTest github.com/s3cur3/parameterized_test

Create test cases from markdown (or json) tables

Parameterized Test

Instead of writing three separate tests:

```
test "editors can view and edit" do
  user = create_user(:editor)
  assert Posts.can_view?(user) == true
  assert Posts.can_edit?(user) == true
end

test "viewers can view but not edit" do
  user = create_user(:viewer)
  ...
end

test "anonymous viewers cannot view or edit" do
    ...
end
```

Parameterized Test

```
param_test "users with editor permissions or better can edit posts",
           """
           | permissions | can_view? | can_edit? |
           |-------------|-----------|-----------|
           | :editor     | true      | true      |
           | :viewer     | true      | false     |
           | nil         | false     | false     |
           """,
  %{permissions: permissions, can_edit?: can_edit?, can_view?: can_view?}
do
  user = create_user_with_permission(permissions)
  assert Posts.can_view?(user) == can_view?
  assert Posts.can_edit?(user) == can_edit?
end
```

Mneme

Mneme github.com/zachallaun/mneme
Snapshot testing library that helps write and update your assertions

```
test "mneme example" do
  auto_assert my_function()
end
```

Mneme Example

```
$ mix test
[1] New · test basic example (MnemeTest)
test/mneme_test.exs:10

────────────────────────────────────────────
  -  auto_assert my_function()
────────────────────────────────────────────
  +  auto_assert %MyAwesomeValue{so: :cool} <- my_function()
────────────────────────────────────────────

Accept new assertion?
y yes  n no  s skip    y
```

Mneme Example

Your new test:

```
test "mneme example" do
  auto_assert %MyAwesomeValue{so: :cool} <- my_function()
end
```

Property Based Testing

CODE BEAM AMERICA

- Fancy testing technique ✨
- You no longer write test cases individually
- Generates random test data to verify code properties
- Discovers edge cases example based tests might miss

Property Based Testing

- Libraries:
  - Erlang: PropEr
  - Elixir: StreamData

```
property "list reversing twice returns original list" do
  check all list <- list_of(integer()) do
    assert Enum.reverse(Enum.reverse(list)) == list
  end
end
```

Thank you!

Any Questions?