Taming Side Effects

Claudio Ortolina - Code BEAM STO 2019

Hello!

Claudio Ortolina

Senior engineer at PSPDFKit GmbH

@cloud8421

Testing side effects

- Non deterministic
- Non idempotent
- Sometimes brittle (requires a lot of setup)
- Potentially slow (e.g. hitting an external http api)

```
defmodule App.Shop.Mailer do
  def send_greeting(user_id) do
    case App.Store.get(user_id) do
      nil ->
        {:error, :user_not_found}
      user ->
        email = App.Shop.Emails.greeting(user)
        send_email(email)
    end
  end
end
```

```
defmodule App.Shop.Mailer do
  def send_greeting(user_id) do
    case App.Store.get(user_id) do
      nil ->
        {:error, :user_not_found}
      user ->
        email = App.Shop.Emails.greeting(user)
        send_email(email)
    end
  end
end
```

```
defmodule App.Shop.Mailer do
  def send_greeting(user_id) do
    case App.Store.get(user_id) do
      nil ->
        {:error, :user_not_found}
      user ->
        email = App.Shop.Emails.greeting(user)
        send_email(email)
    end
  end
end
```

Current solutions

Mocks

- Provide isolation
- Partial compile time guarantees
- Implementation code doesn’t need to change
- Might become obsolete without the developer noticing

Library specific

- e.g. Ecto’s ownership system
- Isolate state for every test
- Different dependencies require different strategies
- Rely on dependency to offer isolation

Goals

- Unified test strategy (avoid context switching between libraries)
- Easier maintenance of test cases
- Clear separation of unit tests and system tests

Strategy

- Group related side effects in modules
- Pass such modules as argument to functions that need to perform side effects
- Organise side-effect modules in a data structure that simplifies their management
- Carefully evaluate if specific mock is needed

Grouping side effects

```
defmodule App.Shop.PostSignup do
  def send_greeting(user_id) do
    case App.Repo.get(user_id) do
      nil ->
        {:error, :user_not_found}
      user ->
        email = App.Shop.Emails.greeting(user)
        App.Mailer.send_email(email)
    end
  end
end
```

```
defmodule App.Backup.Dropbox do
  @spec upload_file(name(), contents()) :: path()
  @spec list_files() :: [path()]
  @spec download_file(path()) :: contents()
  @spec exists?(name()) :: boolean()
end
```

```
defmodule App.Backup.DropboxTest do
  use ExUnit.Case, async: true
  @moduletag :external_dependency

  describe "list_files/0" do
    test "it starts with no files" do
      assert [] == App.Backup.Dropbox.list_files()
    end
  end
end
```

Grouping side effects

- Allows straightforward replacement for tests
- Pushes towards identifying related types
- Allows isolated testing of effect module

Pass modules

```
defmodule App.Shop.PostSignup do
  def send_greeting(user_id, ctx) do
    case ctx.store.get(user_id) do
      nil ->
        {:error, :user_not_found}
      user ->
        email = App.Shop.Emails.greeting(user)
        ctx.mailer.send_email(email)
    end
  end
end
```

```
# requires the caller to make an explicit decision
repo = Map.fetch!(ctx, :repo)

# use a sensible default
repo = Map.get(ctx, :repo, App.Repo)
```

```
defmodule App.ShopPostSignupTest do
  use ExUnit.Case, async: true

  alias App.{Email, User, Shop.PostSignup}

  defmodule Store do
    def get(“example-id") do
      %User{email: “user@example.com"}
    end
    def get(_other_id), do: nil
  end
```

```
  defmodule Mailer do
    def send_email(%Email{to: "user@example.com"}) do
     :ok
    end

    def send_email(_other_email) do
      {:error, :invalid_credentials}
    end
  end

  test "send_greeting/2" do
    ctx = %{store: Store, mailer: Mailer}
    assert :ok == PostSignup.send_greeting("example-id")
  end
end
```

Pass modules

- Establishes convention for side-effect rich modules
- Allows extending over time
- Allows having defaults
- Can we call it context? Naming is hard

Organise with a data structure

```
defmodule App.Ctx do
  defstruct store: App.Store,
            mailer: App.Mailer.Sendgrid

  def default, do: %__MODULE__{}
  def with_override(initial \\ default(), k, v) do
   %{initial | k => v}
  end
end
```

```
defmodule App.Shop.PostSignup do
  def send_greeting(user_id, ctx \\ App.Ctx.default()) do
    case ctx.store.get(user_id) do
      nil ->
        {:error, :user_not_found}
      user ->
        email = App.Shop.Emails.greeting(user)
        ctx.mailer.send_email(email)
    end
  end
end
```

```
defmodule App.ShopPostSignupTest do
  use ExUnit.Case, async: true

  alias App.{Email, User, Shop.PostSignup}

  defmodule Store do
    …
  end

  defmodule Mailer do
    …
  end
```

```
test "send_greeting/2" do
  ctx = Ctx.default()
        |> Ctx.with_override(:store, Store)
        |> Ctx.with_override(:mailer, Mailer)
  assert :ok == PostSignup.send_greeting("example-id", ctx)
end
```

```
defmodule App.Application do
  …

  def start(_type, _args) do
    ctx = App.Ctx.default()

    children = [
      App.Repo,
      {App.HTTP.Listener, port: 3000, ctx: ctx}
    ]
    opts = [strategy: :one_for_one, name: App.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Organise with a data structure

- Introduce a Context module expressed by a struct
- While overkill at first, promotes organic growth of the codebase (strong convention)
- Provides default values
- Initialise and use as high up as possible, ideally at the application start level. Pass down as needed

Do we need mocks at all?

- External services we don’t control (e.g. vendor apis)
- Record 3rd party responses (a.k.a. as VCR) as preferential strategy (with auto-expiry to guarantee that eventually we catch up)
- Master branch on CI should not use recorded responses

How do we test the entire system?

- If possible, have a test harness that stresses all components
- Stress golden path that has to work at all times
- Consider complementary approaches, e.g. asserting facts in production

Goodbye!

Claudio Ortolina

Senior engineer at PSPDFKit GmbH

@cloud8421