Extracted from:

Testing Elixir

Effective and Robust Testing for Elixir and its Ecosystem

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Testing Elixir

Effective and Robust Testing for Elixir and its Ecosystem

Andrea Leopardi and Jeffrey Matthias

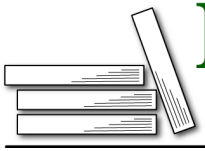
edited by Jacquelyn Carter

Testing Elixir

Effective and Robust Testing for Elixir and its Ecosystem

Andrea Leopardi
Jeffrey Matthias

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Molly McBeath
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-782-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2021

Isolating Code

We can employ multiple strategies to test our code in a way that removes outside variables, controlling the situation in which our code under test must perform and allowing us to expect a specific outcome. We'll cover the easiest method—injecting dependencies and creating basic substitutes (test doubles)—in order to add another option to your testing tool belt.

Dependency Injection

Before we can leverage dependency injection to isolate the behavior of our code under test, let's take a moment to define dependency injection. A dependency is any code that your code relies on. Dependency injection (often abbreviated as DI) is a fancy name for any system that allows your code to utilize a dependency without hard-coding the name of the dependency, allowing any unit of code that meets a contract to be used. In Elixir, we have two common ways to inject a dependency: as a parameter to a function and through the application environment. Utilizing DI in our tests allows us to create replacement dependencies that behave in predictable ways, allowing the tests to focus on the logic inside the code under test.

Passing a dependency as a parameter is the more common way to inject dependencies in unit testing, which is the style that we'll focus on first. In later chapters, we'll cover dependency injection via the application environment, as well as Mox, a tool that aids in creating test doubles.

Test Double Terminology

In this chapter, we'll refer to stand-ins for production code for the purposes of testing as "test doubles." This covers code that can return a prescribed result and even assert that it was called, with or without specific parameters. We'll dive deeper into different kinds of doubles in the next chapter.

Passing a Dependency as a Parameter

Passing a dependency as a parameter is as straightforward as it sounds and is often the simplest solution. We can choose to either pass a function or a module as a parameter. When you inject a module as shown in the following code, it must meet an implicit contract. Otherwise calling the code will raise an exception. The `SoggyWaffle` module has been written to allow a weather forecast function to be passed in:

unit_tests/soggy_waffle/lib/soggy_waffle.ex

```
Line 1  defmodule SoggyWaffle do
        alias SoggyWaffle.WeatherAPI
        def rain?(city, datetime, weather_fn \\ &WeatherAPI.get_forecast/1) do
          with {:ok, response} <- weather_fn.(city) do
    5       {:ok, weather_data} =
              SoggyWaffle.WeatherAPI.ResponseParser.parse_response(response)

            SoggyWaffle.Weather.imminent_rain?(weather_data, datetime)
          end
   10   end
        end
```

Here, at line 3, we're providing a function reference as the default parameter for the injected dependency. This pattern allows us to not have to pass in the real dependency during execution while allowing us to pass in a double during testing. Injecting a function is a tool to reach for when you have a complicated dependency that might return inconsistent results even within the same test setup. Injecting a double allows you to remove unknown factors and give your code under test a consistent environment in which it is tested. Injecting a whole module, which we'll explore in the next chapter, can be useful when your code will call more than one function on that module.

In our code, the other two dependencies, SoggyWaffle.WeatherAPI.ResponseParser and SoggyWaffle.Weather, are purely functional and are separately well tested. As a result, we can save ourselves extra work by not worrying about injecting doubles for those dependencies. They'll never produce a different result. The only part of the code that we need to remove from the test scenario, which is why we call this code isolation, is the call to the weather API at line 4. It very easily could return different results depending on when the code is executed, preventing our test from being able to expect exact results.

Now that we added a mechanism to inject the dependency, let's write a test that can leverage that mechanism to create a scenario where the result should always be the same. Add the following test file (test/soggy_waffle_test.exs) to your application.

unit_tests/soggy_waffle_examples/test/soggy_waffle_test.exs

```
Line 1  defmodule SoggyWaffleTest do
        use ExUnit.Case

        describe "rain?/2" do
    5     test "success: gets forecasts, returns true for imminent rain" do
            now = DateTime.utc_now()
            future_unix = DateTime.to_unix(now) + 1
            expected_city = Enum.random(["Denver", "Los Angeles", "New York"])
            test_pid = self()
```

```
10
-       weather_fn_double = fn city ->
-         send(test_pid, {:get_forecast_called, city})
-
-         data = [
15          %{
-             "dt" => future_unix,
-             "weather" => [%{"id" => _drizzle_id = 300}]
-           }
-         ]
20
-         {:ok, %{"list" => data}}
-       end
-
-       assert SoggyWaffle.rain?(expected_city, now, weather_fn_double)
25
-       assert_received {:get_forecast_called, ^expected_city},
-                       "get_forecast/1 was never called"
-     end
-   end
30 end
```

The major feature of this test is the definition and use of a function-style test double, weather_fn_double, at line 11. It matches the contract of SoggyWaffle.WeatherAPI.get_forecast/1, but, unlike the real function, it'll always return the exact same response. Now our test can assert on specific values, helping us to know that the module under test, SoggyWaffle, is behaving correctly.

Notice, though, that the test double doesn't just meet the contract, it also has a mechanism to report that the function was called and to send the parameter passed to it back to the test, seen at line 12. When the double is called, the module will send the test process the {:get_forecast_called, city} message.

The test code at line 26 makes sure that the function was called, but it also ensures that the correct value was passed to it. We call this style of testing an expectation. This way, if that dependency is never called, the test will fail. This kind of expectation is mostly useful when there's an expected side effect of your code and your tests are concerned with making sure it happens. Admittedly our test is just making a query via the function call, so the expectation isn't totally necessary in this case, but it still illustrates well how to add an expectation to a test double.

Some other features of this test are worth noting before we move on. The code under test uses time comparisons in its business logic, actually found in a functional dependency, SoggyWaffle.Weather. Because we aren't isolating SoggyWaffle from that dependency, knowledge of the time comparison has now bled up into the knowledge needed to test SoggyWaffle. This ties back to our earlier

discussion of defining the unit, or black box, for our test. Because SoggyWaffle.Weather is well tested, there is little downside to taking this approach, as it will prove to be easier to understand and maintain in the long run, a very important priority when we design our tests.

The anonymous function is defined inside of the test at line 11, as opposed to a named function in the same file, because it needs to have access to the value bound to future_unix. Additionally, we have to define the test's process ID (or PID) *outside* of the function because self() won't be evaluated until the function is executed, inside the process of the code under test, which could have a different PID than our test. Because anonymous functions are closures in Elixir, the value of future_unix is part of the function when it gets executed. The call to self() is evaluated at the definition of the anonymous function, with the value being the PID of that test process. Leveraging closures is one of the advantages of function-style dependency injection. We'll examine other DI tools later that give us similar power.

One last, notable feature of this test that isn't related to dependency injection is the use of randomized data at line 8. When test data is incidental, meaning it shouldn't change the behavior of your code, try to avoid hard-coded values.

Don't worry about this issue when you're making your first pass on a test. But before you consider that test complete, we suggest you look for places where you can switch from hard-coded values to randomized data. While you aren't looking to try to test your code with every possible value that could be passed to it (that's more in line with property-based testing, covered in Chapter 7, Property-Based Testing, on page ?), it's possible to accidentally write code that only passes with a single specific input. That's obviously not ideal, so it's nice to reach for the low-hanging fruit and add a little variation to what's passed in. In this case, there's no benefit to testing all of the city options because the value of the string itself won't change anything: it's just being passed to the weather function. Our test's concern is that the correct value is passed to the double we injected. While we have a hard-coded list of possible cities, there are libraries that can help generate random data, like Faker.[5] But even handling it locally like we did here will net you the benefits without having to pull in a new dependency.

Be careful, though. The pursuit of dynamic tests like this can go too far, leaving your test hard to understand and hard to maintain. Additionally, the test *must* have a way to know what data it's using, and the assertions should be able to take that into account.

5. https://hex.pm/packages/faker

Finer Control over Dependency Injection

An alternative to injecting a whole function is to pass in a single value. An example you might see in code is when the outside dependency is the system time. If your code under test needs to do something with system time, it's very difficult for your tests to assert a known response or value unless you can control system time. While controlling your code's concept of system time is easy in some languages, it isn't in Elixir. That makes this scenario a perfect candidate for injecting a single value. The following code allows for an injected value, but it defaults to the result of a function call if no parameter is passed:

unit_tests/soggy_waffle/lib/soggy_waffle/weather.ex

```
Line 1  defmodule SoggyWaffle.Weather do
        @type t :: %__MODULE__{}

        defstruct [:datetime, :rain?]
      5
        @spec imminent_rain?([t()], DateTime.t()) :: boolean()
        def imminent_rain?(weather_data, now \\ DateTime.utc_now()) do
          Enum.any?(weather_data, fn
            %__MODULE__{rain?: true} = weather ->
     10       in_next_4_hours?(now, weather.datetime)

            _ ->
              false
          end)
     15   end

        defp in_next_4_hours?(now, weather_datetime) do
          four_hours_from_now =
            DateTime.add(now, _4_hours_in_seconds = 4 * 60 * 60)
     20
          DateTime.compare(weather_datetime, now) in [:gt, :eq] and
            DateTime.compare(weather_datetime, four_hours_from_now) in [:lt, :eq]
        end
      end
```

Looking at our function signature in line 7, we can see that without a value passed in, the result of DateTime.utc_now/0 will be bound to the variable datetime. Our tests will pass a value in, overriding the default, but the code will use the current time when running in production. By allowing our function to take a known time, we can remove the unknown. When running in production, your code will never be passed another value, but testing it just got a lot easier. Our tests can now create a controlled environment in which to test our code.

unit_tests/soggy_waffle_examples/test/soggy_waffle/weather_test.exs

```
Line 1  defmodule SoggyWaffle.WeatherTest do
```

```
  use ExUnit.Case
  alias SoggyWaffle.Weather

  describe "imminent_rain?/2" do
    test "returns true when it will rain in the future" do
      now = datetime_struct(hour: 0, minute: 0, second: 0)
      one_second_from_now = datetime_struct(hour: 0, minute: 0, second: 1)

      weather_data = [weather_struct(one_second_from_now, :rain)]

      assert Weather.imminent_rain?(weather_data, now) == true
    end
```

Our code under test is all time-based, with "the future" being very important. By passing in a value, we're able to strictly control the conditions in which our code is executed, isolating our code under test from its dependency, system time (via DateTime.utc_now/0). This is especially important because we need alignment between "now," when the code is executing, and the time in the weather data passed into the function.

Notice the one-second difference between "now" and the weather data. This is a practice called boundary testing. Our code is looking for values in the future, and we have made the future as close to the boundary as we can so that we can be certain that any time more recent than that will also pass the test. While technically our data could have been one microsecond in the future, the data our application will get from the weather API is granular only down to the second, so we'll stick with a unit of time that most people are more familiar with, the second. Whenever you're writing tests for code that does comparisons, you should strive to test right at the boundaries. This is even more important if you're dealing with any time zone–based logic, as testing too far from a boundary can hide bad time-zone logic.

In our test, there are calls to two helper functions, datetime_struct/1 and weather_struct/2. They can be explained fairly easily: datetime_struct/1 returns a %DateTime{} struct where all the values are the same each time except the overrides for hour, minute, and second, while weather_struct/2 returns SoggyWaffle.Weather structs as defined in the module of the same name in our application. These allow us to easily construct test data in a way that improves the readability of the test. Let's see the definitions for these helper functions:

unit_tests/soggy_waffle_examples/test/soggy_waffle/weather_test.exs

```
defp weather_struct(datetime, condition) do
  %Weather{
    datetime: datetime,
    rain?: condition == :rain
  }
end
```

```
defp datetime_struct(options) do
  %DateTime{
    calendar: Calendar.ISO,
    day: 1,
    hour: Keyword.fetch!(options, :hour),
    microsecond: {0, 0},
    minute: Keyword.fetch!(options, :minute),
    month: 1,
    second: Keyword.fetch!(options, :second),
    std_offset: 0,
    time_zone: "Etc/UTC",
    utc_offset: 0,
    year: 2020,
    zone_abbr: "UTC"
  }
end
```

Be careful, though, because helper functions like this can become a maintenance nightmare. When writing helper functions for your tests, try to keep them defined in the same file as the tests using them, and try to keep them simple in functionality, with clear, explanatory names. If you design the helper function's signature to enhance the readability of your tests, all the better. In the case of our datetime_struct/1, the signature takes a keyword list, letting the call itself highlight what's important about the return value.

The keys aren't necessary since plain values would have sufficed, but they make it fairly easy to understand the difference in the return values at lines 7 and 8 of the test body from the SoggyWaffle.WeatherTest code sample on page 5. All the other values will be the same. By contrast, weather_struct/2 only takes plain values; but intentional parameter naming, both of the variable name and that atom, still keep the call easy to understand.

For unit testing, injecting dependencies through the API will help keep your code clean and your tests easy and controlled. As you move on to dealing with integration testing, you'll need to explore other methods of dependency injection since you won't necessarily have the ability to pass a dependency in from your tests like you would with a unit test. Both of these ways of injecting dependencies are simple and easy to understand. Ultimately, that's the best argument for using them.