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

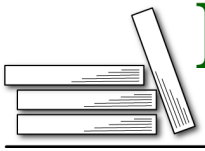
edited by Jacquelyn Carter

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

Example-Based Tests

When writing tests, we usually write something that we could call *example-based* tests. In this section, we'll have a new look at them from the perspective of the inputs that we feed to our code. We'll define terminology to identify example-based tests, which will help us later compare them to property-based tests.

Let's start with an example. Let's say we want to test that the Enum.sort/1 function correctly sorts lists. We could call the following an example-based test because it verifies that the code we want to test (Enum.sort/1 in this case) works by giving a few examples of how it should work:

property_based_testing/sorting/test/example_based_sort_test.exs

```
defmodule ExampleBasedSortTest do
  use ExUnit.Case

  test "Enum.sort/1 sorts lists" do
    assert Enum.sort([]) == []
    assert Enum.sort([1, 2, 3]) == [1, 2, 3]
    assert Enum.sort([2, 1, 3]) == [1, 2, 3]
  end
end
```

In this example, we're verifying that sorting an empty list returns an empty list, that sorting an ordered list leaves it unchanged, and that sorting an unsorted list returns the sorted version of that list. We chose three examples that we thought would cover a representative sample of all the possible inputs to Enum.sort/1. This works, but you can see that there are a lot more inputs we could test, such as negative numbers, lists with duplicates in them, and so on.

Sometimes, we try to test more inputs and simplify the test at the same time by extracting the inputs and corresponding expected outputs and then running the test on those inputs and outputs:

property_based_testing/sorting/test/tabular_sort_test.exs

```
defmodule TabularSortTest do
  use ExUnit.Case

  test "Enum.sort/1 sorts lists" do
    inputs_and_outputs = [
      {[], []},
      {[1, 2, 3], [1, 2, 3]},
      {[2, 1, 3], [1, 2, 3]},
      {[2, 1, 2], [1, 2, 2]},
      {[0, -1, -2], [-2, -1, 0]}
    ]

    for {input, expected_output} <- inputs_and_outputs do
      assert Enum.sort(input) == expected_output
```

```
    end
  end
end
```

When using this kind of approach, tests look like *assertion tables*. An assertion table is a table of inputs and outputs, and the tests assert that running the code on the input of each row of the table results in the output on that same row. For this reason, an alternative name for example-based tests is *tabular tests*.

This kind of test has many benefits. First of all, tests like these are easy to write since we know how our code works, and it's usually straightforward to come up with inputs for the code we're testing. Another benefit of these tests is that, since we're specifying all inputs, we can choose to test corner cases that we suspect might be problematic for our code. In our trivial sorting example, we know that the empty list is a corner case because it's a peculiar list, so we can just go ahead and test our code on it every time we run the test.

However, these tests have some downsides as well. Testing the same known inputs on every test run means that it's hard to discover unknown corner cases because, well, they're unknown. At the same time, it's hard to discover inputs that our code doesn't support or that it should support because we're the ones writing the examples in the test. Let's see how we can improve the situation.

Introducing Randomness and Property-Based Testing

We can solve some of the problems that example-based tests suffer from by introducing a bit of chaos in our tests. Using randomness to generate inputs will allow us to test a wider range of inputs against our code and potentially create inputs that trigger edge cases.

In our sorting example, what we really want to test is that the output of `Enum.sort/1` is a sorted version of the input. For any random input, we can think of a few properties that the output will always retain. For example, the output list always has the same length as the input list. Another property is that the output list is always sorted, which is something that we can check in a pretty straightforward way by checking that each element is smaller than or equal to the following one. Now that we've thought of these properties, we could change our test so that we generate random lists and test these properties on the output of our code instead of checking *what* the output is. Let's see how to do that:

property_based_testing/sorting/test/randomized_sort_test.exs

```
Line 1 defmodule RandomizedSortTest do
     - use ExUnit.Case
```

```
-
-   test "Enum.sort/1 sorts lists" do
5     for _ <- 1..10 do
-       random_list = random_list()
-       sorted_list = Enum.sort(random_list)
-
-       assert length(random_list) == length(sorted_list)
10      assert sorted?(sorted_list)
-     end
-   end
-
-   defp random_list do
15    Stream.repeatedly(fn -> Enum.random(-100..100) end)
-     |> Enum.take(_length = Enum.random(0..10))
-   end
-
-   defp sorted?([first, second | rest]),
20    do: first <= second and sorted?([second | rest])
-
-   defp sorted?(_other), do: true
- end
```

The random_list/0 function creates an infinite stream of random numbers between -100 and 100 and then picks a random number of elements from the stream using Enum.take/2. The number of elements we pick from the stream is the length of the random list, which we keep between 0 and 10 elements. The sorted?/1 function checks that the first two elements of the list are sorted and then recursively checks the rest of the list until it arrives at an empty or one-element list, which is always sorted. On line 9, we check our first property, that the sorted list has the same number of elements as the input list. On line 10, we check the second property, that the sorted list is sorted.

This approach to testing has a few benefits. One of the most obvious is that it can potentially test on a lot more inputs than example-based testing can. In our example, if we want to change the number of tested lists to a hundred or a thousand, we can just change the right end of the range on line 5. However, the usefulness of testing on many inputs is limited unless the inputs vary.

The Role of Randomness

This is where randomness comes into play. By having a lot of inputs generated at random, our hope is to cover a decent part of the possible inputs to our code and at the same time cover a good *variety* of inputs. Essentially, we want a good sample of inputs that represents the *input space*, which is the set of all possible inputs. In our example, we're still covering a tiny part of our input space (all lists of numbers), but covering the whole input space is often unfeasible. Random generation gives us a nice compromise, especially consid-

ering that every time the tests are run, possibly different lists are generated. Generating random elements also helps us to uncover potential corner cases that we didn't anticipate.

You might be asking yourself how much randomness is *enough*, that is, how many inputs you need to generate or how many times you need to run these tests to have confidence that they cover enough of the input space. In many cases, the input space is infinite or too vast to cover, but only you will know how far to push it based on the specific use case.

The test we wrote for `Enum.sort/1` is an example of a kind of test called *property-based tests*. They are called that because of the method we used to come up with this kind of test: we think of *properties* that our code holds regardless of the input we feed to it, provided the input is valid.

The benefits of property-based testing don't end with what we've just discussed. Coming up with valid inputs and properties is a huge part of property-based testing, but it's also a helpful design tool. If you have to write down in clear terms what the valid inputs of your code are, you could end up expanding or shrinking the space of valid inputs. Coming up with properties, instead, forces you to think about what your code should do regardless of the specific input you feed to it, which might help with the design or implementation of the code. In the list-sorting example, the functionality is trivial, so it's hard to see the design benefits of property-based testing; but in more complex contexts, it can be useful to think about these things.

Property-based testing is rarely done in a hand-rolled way like we did in our example, as there's a plethora of frameworks (for all kinds of programming languages) that facilitate the implementation of property-based tests. Usually, property-based testing frameworks provide powerful ways of generating data and an infrastructure for verifying properties against that generated data. There's also one important feature that makes using a property-based testing framework a clear advantage over rolling out your own randomness-based tests: frameworks *simplify* the randomly generated inputs when a failure occurs, and they present error messages that tend to be significantly easier to understand and address than if you handwrite tests with random data like we did.

For Elixir, the property-based testing framework we're going to use from now on is called stream_data.[3]

3. https://github.com/whatyouhide/stream_data

Why Use stream_data?

The Elixir and Erlang ecosystems have good support for property-based testing through well-established libraries such as Quviq's QuickCheck and PropEr for Erlang,[a] [b] and PropCheck,[c] Quixir,[d] and stream_data for Elixir.

However, we're biased toward stream_data since Andrea wrote the original library, which means we know it well and are comfortable with it. In any case, the property-based testing concepts we're going to illustrate work well with all libraries.

We feel like having some context on why stream_data was created in the first place, even if other property-based testing frameworks were already available, could be helpful to readers. One reason was that originally the plan was to include stream_data in the Elixir standard library, which meant having to write something from scratch to make sure licensing wasn't a problem and that the Elixir core team would be able to maintain the code. The team later realized that stream_data worked well enough as a library and so it didn't end up in Elixir itself. Another reason was that all existing property-based testing frameworks would only generate random data in the context of property-based testing, without taking advantage of Elixir streams to make data generation a general-purpose tool.

a. http://www.quviq.com/products/erlang-quickcheck/
b. https://github.com/proper-testing/proper
c. https://github.com/alfert/propcheck
d. https://github.com/pragdave/quixir

Introducing stream_data

stream_data is a property-based testing framework for Elixir. It provides two main functionalities, data generation and a framework for writing and running properties. The data generation aspect of the library is usable outside of property-based testing as a standalone feature, but it's the backbone of the whole framework and is also used extensively when writing properties.

To follow along in the next few sections, create a new Mix project with `$ mix new sorting` and then add `:stream_data` as a dependency in your `mix.exs` file:

property_based_testing/sorting/mix.exs
```
defp deps do
  [{:stream_data, ">= 0.0.0", only: [:dev, :test]}]
end
```

Now, run `$ mix deps.get` to fetch the dependency. As you can see in the code, we've only added `:stream_data` in the `:test` environment since we'll only be using the library when testing.

Before diving into the framework, let's rewrite the RandomizedSortTest test we hand-rolled earlier to use the tools that stream_data provides:

property_based_testing/sorting/test/randomized_sort_stream_data_test.exs

```
Line 1  defmodule FirstStreamDataPropertySortTest do
        use ExUnit.Case
        use ExUnitProperties
        
5       property "Enum.sort/1 sorts lists" do
          check all list <- list_of(integer()) do
            sorted_list = Enum.sort(list)
        
            assert length(list) == length(sorted_list)
10          assert sorted?(sorted_list)
          end
        end
        
        defp sorted?([first, second | rest]),
15        do: first <= second and sorted?([second | rest])
        
        defp sorted?(_other), do: true
      end
```

Don't worry about the new things you see in this test. We'll cover all of them in this chapter. The goal here is to show you what stream_data looks like. For now, run mix test in the project where you added this file and see the beautiful green dots.

As it turns out, the underlying shape of the test is quite similar to RandomizedSortTest. Instead of using the test macro to define a test, we use property (on line 5). Then we use a new construct, check all, on line 6. This replaces the for comprehension we had. On the same line, we have list <- list_of(integer()). That's exactly one of the most important features of a property-based framework: *data generators*. Here stream_data takes care of generating random data (with cool characteristics we'll see later) for you. Now that we have an idea of what a stream_data test looks like, let's move on to dissecting its components in a more detailed way.

In the next sections, we're going to start exploring from the data generation aspect of stream_data and then move on to designing and running properties. To follow along, run iex -S mix to fire up an IEx session from the root of the project that includes stream_data as a dependency.

You might be wondering why we won't illustrate these concepts on one of the applications we developed in the previous chapters (such as Soggy Waffle). Well, the reason is that we would have to bend those applications in weird ways to be able to show these ideas effectively. Instead, we decided to use

simple, small, and self-contained examples so that we can focus on property-based testing concepts and tools.