Extracted from:

# Programming Elixir 1.3



The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

# Programming Elixir 1.3

Functional
|> Concurrent
|> Pragmatic
|> Fun

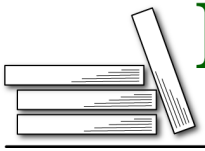
Dave Thomas

Foreword by
José Valim,
Creator of Elixir

# Programming Elixir 1.3

Dave Thomas

The Pragmatic Bookshelf
Raleigh, North Carolina





Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

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

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



Printed in the United States of America.
ISBN-13: 978-1-68050-200-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2016

You'd expect that a relatively new language would come with a fairly minimal set of tools—after all, the development team will be having fun playing with the language.

Not so with Elixir. Tooling was important from the start, and the core team have spent a lot of time providing a world-class environment in which to develop code.

In this short chapter, we'll look at some aspects of this.

This chapter is not the full list. We've already seen the ex_doc tool, which creates beautiful documentation for your code. Later, when we look at OTP Applications, on page ? we'll experiment with the Elixir Release manager, a tool for managing releases while your application continues to run.

For now, let's look at testing, code exploration, and server monitoring tools.

# Testing

We already used the ExUnit framework to write tests for our Issues Tracker app. But that chapter only scratched the surface of Elixir testing. Let's dig deeper.

## Testing the Comments

When I document my functions, I like to include examples of the function being used—comments saying things such as, "Feed it these arguments, and you'll get this result." In the Elixir world, a common way to do this is to show the function being used in an iex session.

Let's look at an example from our Issues app. The TableFormatter formatter module defines a number of self-contained functions that we can document.

project/5/issues/lib/issues/table_formatter.ex

```
defmodule Issues.TableFormatter do
  import Enum, only: [ each: 2, map: 2, map_join: 3, max: 1 ]

  @doc """
  Takes a list of row data, where each row is a Map, and a list of
  headers. Prints a table to STDOUT of the data from each row
  identified by each header. That is, each header identifies a column,
  and those columns are extracted and printed from the rows.
  We calculate the width of each column to fit the longest element
  in that column.
  """
  def print_table_for_columns(rows, headers) do
    with data_by_columns = split_into_columns(rows, headers),
         column_widths   = widths_of(data_by_columns),
         format          = format_for(column_widths)
```

```
  do
      puts_one_line_in_columns(headers, format)
      IO.puts(separator(column_widths))
      puts_in_columns(data_by_columns, format)
  end
end

@doc """
Given a list of rows, where each row contains a keyed list
of columns, return a list containing lists of the data in
each column. The `headers` parameter contains the
list of columns to extract

## Example

    iex> list = [Enum.into([{"a", "1"},{"b", "2"},{"c", "3"}], %{}),
    ...>         Enum.into([{"a", "4"},{"b", "5"},{"c", "6"}], %{})]
    iex> Issues.TableFormatter.split_into_columns(list, [ "a", "b", "c" ])
    [ ["1", "4"], ["2", "5"], ["3", "6"] ]

"""
def split_into_columns(rows, headers) do
  for header <- headers do
    for row <- rows, do: printable(row[header])
  end
end

@doc """
Return a binary (string) version of our parameter.
## Examples
    iex> Issues.TableFormatter.printable("a")
    "a"
    iex> Issues.TableFormatter.printable(99)
    "99"
"""
def printable(str) when is_binary(str), do: str
def printable(str), do: to_string(str)

@doc """
Given a list containing sublists, where each sublist contains the data for
a column, return a list containing the maximum width of each column

## Example
    iex> data = [ [ "cat", "wombat", "elk"], ["mongoose", "ant", "gnu"]]
    iex> Issues.TableFormatter.widths_of(data)
    [ 6, 8 ]
"""
def widths_of(columns) do
  for column <- columns, do: column |> map(&String.length/1) |> max
end

@doc """
Return a format string that hard codes the widths of a set of columns.
We put `" | "` between each column.
```

```
  ## Example
      iex> widths = [5,6,99]
      iex> Issues.TableFormatter.format_for(widths)
      "~-5s | ~-6s | ~-99s~n"
  """
  def format_for(column_widths) do
    map_join(column_widths, " | ", fn width -> "~-#{width}s" end) <> "~n"
  end

  @doc """
  Generate the line that goes below the column headings. It is a string of
  hyphens, with + signs where the vertical bar between the columns goes.

  ## Example
        iex> widths = [5,6,9]
        iex> Issues.TableFormatter.separator(widths)
        "------+--------+----------"
  """
  def separator(column_widths) do
    map_join(column_widths, "-+-", fn width -> List.duplicate("-", width) end)
  end

  @doc """
  Given a list containing rows of data, a list containing the header selectors,
  and a format string, write the extracted data under control of the format string.
  """
  def puts_in_columns(data_by_columns, format) do
    data_by_columns
    |> List.zip
    |> map(&Tuple.to_list/1)
    |> each(&puts_one_line_in_columns(&1, format))
  end

  def puts_one_line_in_columns(fields, format) do
    :io.format(format, fields)
  end
end
```

Note how some of the documentation contains sample iex sessions. I like doing this. It helps people who come along later understand how to use my code. But, as importantly, it lets *me* understand what my code will feel like to use. I typically write these sample sessions before I start on the code, changing stuff around until the API feels right.

But the problem with comments is that they just don't get maintained. The code changes, the comment gets stale, and it becomes useless. Fortunately, ExUnit has `doctest`, a tool that extracts the iex sessions from your code's `@doc` strings, runs it, and checks that the output agrees with the comment.

To invoke it, simply add one or more

```
doctest «ModuleName»
```

lines to your test files. You can add them to existing test files for a module (such as table_formatter_test.exs) or create a new test file just for them. That's what we'll do here. Let's create a new test file, test/doc_test.exs, containing this:

project/5/issues/test/doc_test.exs

```
defmodule DocTest do
  use ExUnit.Case
  doctest Issues.TableFormatter
end
```

We can now run it:

```
$ mix test test/doc_test.exs
......
Finished in 0.00 seconds
5 tests, 0 failures
```

And, of course, these tests are integrated into the overall test suite:

```
$ mix test
..............
Finished in 0.01 seconds
13 tests, 0 failures
```

Let's force an error to see what happens:

```
@doc """
Return a binary (string) version of our parameter.

## Examples

    iex> Issues.TableFormatter.printable("a")
    "a"
    iex> Issues.TableFormatter.printable(99)
    "99.0"
"""

def printable(str) when is_binary(str), do: str
def printable(str), do: to_string(str)
```

And run the tests again:

```
$  mix test test/doc_test.exs
.........
  1) test doc at Issues.TableFormatter.printable/1 (3) (DocTest)
     Doctest failed
     code: " Issues.TableFormatter.printable(99) should equal \"99.0\""
     lhs:  "\"99\""
     stacktrace:
       lib/issues/table_formatter.ex:52: Issues.TableFormatter (module)
6 tests, 1 failures
```

## Structuring Tests

You'll often find yourself wanting to group your tests at a finer level than per module. For example, you might have multiple tests for a particular function, or multiple functions that work on the same test data. ExUnit has you covered.

Let's test this simple module:

tooling/pbt/lib/stats.ex

```
defmodule Stats do
  def sum(vals),     do: vals |> Enum.reduce(0, &+/2)
  def count(vals),   do: vals |> length
  def average(vals), do: sum(vals) / count(vals)
end
```

Our tests might look something like this:

tooling/pbt/test/describe.exs

```
defmodule TestStats do
  use ExUnit.Case

  test "calculates sum" do
    list = [1, 3, 5, 7, 9]
    assert Stats.sum(list) == 25
  end

  test "calculates count" do
    list = [1, 3, 5, 7, 9]
    assert Stats.count(list) == 5
  end

  test "calculates average" do
    list = [1, 3, 5, 7, 9]
    assert Stats.average(list) == 5
  end
end
```

There are a couple of issues here. First, these tests only pass in a list of integers. Presumably we'd want to test with floats, too. So let's use the describe feature of ExUnit to document that these are the integer versions of the tests:

tooling/pbt/test/describe.exs

```
defmodule TestStats0 do
  use ExUnit.Case

  describe "Stats on lists of ints" do
    test "calculates sum" do
      list = [1, 3, 5, 7, 9]
      assert Stats.sum(list) == 25
    end

    test "calculates count" do
      list = [1, 3, 5, 7, 9]
```

```
      assert Stats.count(list) == 5
    end

    test "calculates average" do
      list = [1, 3, 5, 7, 9]
      assert Stats.average(list) == 5
    end
  end
end
```

If any of these fail, the message would include the description and test name:

```
test Stats on lists of ints calculates sum (TestStats0)
     test/describe.exs:12
     Assertion with == failed
     ...
```

A second issue with our tests is that we're duplicating the test data in each function. In this particular case this is arguably not a major problem. There are times, however, where this data is complicated to create. So let's use the setup feature to move this code into a single place. While we're at it, we'll also put the expected answers into the setup. This means that if we decide to change the test data in future, we'll find it all in one place.

tooling/pbt/test/describe.exs

```
defmodule TestStats1 do
  use ExUnit.Case

  describe "Stats on lists of ints" do

    setup do
      [ list:    [1, 3, 5, 7, 9, 11],
        sum:     36,
        count:   6
      ]
    end

    test "calculates sum", fixture do
      assert Stats.sum(fixture.list) == fixture.sum
    end

    test "calculates count", fixture do
      assert Stats.count(fixture.list) == fixture.count
    end

    test "calculates average", fixture do
      assert Stats.average(fixture.list) == fixture.sum / fixture.count
    end
  end
end
```

The setup function is invoked automatically before each test is run. (There's also a setup_all function that is just invoked once for the test run.) The setup

function returns a keyword list of named test data. In testing circles, this data, which is used to drive tests, is called a *fixture*.

This data is passed to our tests as a second parameter, following the test name. In my tests, I've called this parameter `fixture`. I then access the individual fields using the `fixture.list` syntax.

In the code here I passed a block to `setup`. You can also pass the name of a function (as an atom).

Inside the setup code you can define callbacks using `on_exit`. These will be invoked at the end of the test. They can be used to undo changes made by the test.

There's a lot of depth in ExUnit. I'd recommend spending a little time in the ExUnit docs.[1]

1. http://elixir-lang.org/docs/stable/ex_unit/ExUnit.html