

Extracted from:

Programming Elixir ≥ 1.6

Functional |> Concurrent |> Pragmatic |> Fun

This PDF file contains pages extracted from *Programming Elixir ≥ 1.6*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

Programming Elixir ≥ 1.6

Functional

|> Concurrent

|> Pragmatic

|> Fun

Dave Thomas

Foreword by

José Valim,

Creator of Elixir



Programming Elixir $\geq$ 1.6

Functional |> Concurrent |> Pragmatic |> Fun

Dave Thomas

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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:

Publisher: Andy Hunt

VP of Operations: Janet Furlow

Managing Editor: Brian MacDonald

Supervising Editor: Jacquelyn Carter

Copy Editor: Candace Cunningham

Indexing: Potomac Indexing, LLC

Layout: Gilson Graphics

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

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

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-13: 978-1-68050-299-2

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—May 2018

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 has 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 ExDoc 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.

Debugging with IEx

You already know that IEx is the go-to utility to play with Elixir code. It also has a secret and dark second life as a debugger. It isn't fancy, but it lets you get into a running program and examine the environment.

You enter the debugger when running Elixir code hits a *breakpoint*. There are two ways of creating a breakpoint. One works by adding calls into the code you want to debug. The other is initiated from inside IEx. We'll look at both using the following (buggy) code:

```
tooling/buggy/lib/buggy.ex
defmodule Buggy do
  def parse_header(
    <<
      format::integer-16,
      tracks::integer-16,
```

```

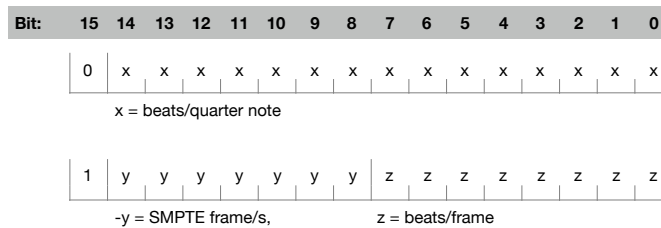
➤ division::integer-16
  >>
) do
  IO.puts "format: #{format}"
  IO.puts "tracks: #{tracks}"
  IO.puts "division: #{decode(division)}"
end

def decode(<< 1::1, beats::15 >>) do
  "J = #{beats}"
end

def decode(<< 0::1, fps::7, beats::8 >>) do
  "#{fps} fps, #{beats}/frame"
end
end
end

```

This code is supposed to decode the data part of a MIDI header frame. This contains three 16-bit fields: the format, the number of tracks, and the time division. This last field comes in one of two formats:



The `parse_header/1` function splits the overall header into the three fields, and the `decode/1` function works out which type of time division we have.

Let's run it, using a sample header I extracted from a MIDI file.

```

$ iex -S mix
iex> header = << 0, 1, 0, 8, 0, 120 >>
<<0, 1, 0, 8, 0, 120>>
iex> Buggy.parse_header header
format: 1
tracks: 8
** (FunctionClauseError) no function clause matching in Buggy.decode/1
iex>

```

Oh no! That was totally unexpected. It looks like we're not passing the correct value to decode. Let's use the debugger to find out what's going on.

Injecting Breakpoints Using IEx.pry

We can add a breakpoint to our source code using the `pry` function. For example, to stop our code just before we call `decode` we could write this:

```

def parse_header(
  <<
    format::integer-16,
    tracks::integer-16,
    division::integer-16
  >>
) do
➤ require IEx; IEx.pry
  IO.puts "format: #{format}"
  IO.puts "tracks: #{tracks}"
  IO.puts "division: #{decode(division)}"
end

```

(We need the `require` because `pry` is a macro.)

Let's try the code now:

```

$ iex -S mix
iex> Buggy.parse_header << 0, 1, 0, 8, 0, 120 >>
Break reached: Buggy.parse_header/1 (lib/buggy.ex:11)

  9:
➤ 10:   require IEx; IEx.pry
 11:   IO.puts "format: #{format}"

pry> binding
[division: 120, format: 1, tracks: 8]
iex> continue()
format: 1
tracks: 8
** (FunctionClauseError) no function clause matching in Buggy.decode/1

```

We reached the breakpoint, and IEx entered *pry mode*. It showed us the function we were in as well as the source lines surrounding the breakpoint.

At this point, IEx is running in the context of this function, so a call to `binding` shows the local variables. The value in the `division` function is 120, but that isn't matching either of the parameters to `decode`.

Aha! `decode` is expecting a binary, not an integer. Let's fix our code:

```

def parse_header(
  <<
    format::integer-16,
    tracks::integer-16,
➤  division::bits-16
  >>
) do
  ...

```

The `pry` call is still in there, so let's recompile and try again:

```

iex> r Buggy
{:reloaded, Buggy, [Buggy]}
iex> Buggy.parse_header << 0, 1, 0, 8, 0, 120 >>
Break reached: Buggy.parse_header/1 (lib/buggy.ex:12)

 10:   ) do
 11:
 12:     require IEx; IEx.pry
 13:     IO.puts "format: #{format}"
 14:     IO.puts "tracks: #{tracks}"

pry> binding
[division: <<0, 120>>, format: 1, tracks: 8]
pry> continue
format: 1
tracks: 8
division: 0 fps, 120/frame
:ok

```

Now the division is a binary, and when we continue the code runs and outputs the header fields. Except...it's parsing the time division as if it were the SMPTE version, and not the beats/quarter note version.

Setting Breakpoints with Break

The second way to create a breakpoint doesn't involve any code changes. Instead, you can use the `break!` command inside `IEx` to add a breakpoint on any public function. Let's remove the call to `pry` and run the code again. Inside `IEx` we'll add a breakpoint on the `decode` function:

```

iex> require IEx
IEx
iex> break! Buggy.decode/1
1
iex> breaks

  ID  Module.function/arity  Pending stops
  ----
  1    Buggy.decode/1      1

iex> Buggy.parse_header << 0, 1, 0, 8, 0, 120 >>
format: 1
tracks: 8
Break reached: Buggy.decode/1 (lib/buggy.ex:21)

 19:   end
 20:
 21:   def decode(<< 0::1, fps::7, beats::8 >>) do
 22:     "#{-fps} fps, #{beats}/frame"
 23:   end

pry> binding
[division: <<0, 120>>, format: 1, tracks: 8]

```

We hit the breakpoint, and we are indeed matching the wrong version of the decode function when we pass it 0000000001111000. Ah, that's because I'm discriminating based on the value of the top bit, and I got it the wrong way around: the SMPTE version should be

```
def decode(<< 1::1, fps::7, beats::8 >>) do
```

and the beats version should be

```
def decode(<< 0::1, beats::15 >>) do
```

There's lots more functionality in the debugger. You can start by getting help for IEx.break/4.

Does This Seem a Little Artificial?

I have a confession to make. The only time I use the Elixir breakpoint facility is when I work on this section of the book. If I have to add code to the source to break in the middle of a function, then I can just raise an exception there instead to get the information I need. And the fact that I can only break at public functions from inside IEx means that I can't get the kind of granularity I need to diagnose issues, because 90% of my functions are private.

However, I'm an old curmudgeon—my favorite editor is a card punch. Don't let my lack of enthusiasm stop you from trying the debugger.

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 doctests, 0 failures
```

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

```
$ mix test
.....
Finished in 0.1 seconds
5 doctests, 9 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.ex
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 the 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
  end
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 invoked just 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