Property Testing

A draft experience report on the use of StreamData for

Property Testing

A draft experience report on the use of StreamData for Property Testing in Elixir

A draft experience report on the use of StreamData for

Property Testing

in Elixir at my last job

Rob Howard
@damncabbage
http://robhoward.id.au

FuzzYard

What is Property Testing?

(and StreamData)

We just don't know

```
test "word-count" do
  text = "This is a test. And this too!"

  assert WordCount.count(text) == 7
end
```

```
test "word-count" do
  words = [
    "This", "is", "a", "test.",
    "And", "this", "too!",
  ]
  text = Enum.join(" ", words)

  assert WordCount.count(text) == 7
end
```

```elixir
def word do
  Enum.random([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

test "word-count" do
  word_count = Enum.random(1..10)
  text = (1..word_count)
  |> Enum.map(fn _ -> word() end)
  |> Enum.join(" ")
  assert WordCount.count(text) == word_count
end
```

```
def word do
  Enum.random([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

test "word-count" do
  word_count = Enum.random(1..10)
  text = (1..word_count)
  |> Enum.map(fn _ -> word() end)
  |> Enum.join(" ")
  assert WordCount.count(text) == word_count
end
```

```elixir
def word do
  Enum.random([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

test "word-count" do
  word_count = Enum.random(1..10)
  text = (1..word_count)
  |> Enum.map(fn _ -> word() end)
  |> Enum.join(" ")
  assert WordCount.count(text) == word_count
end
```

```
def word do
  member_of([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

property "word-count" do
  check all count <- positive_integer()
            words <- list_of(word, length: count)
  do
    text = words |> Enum.join(" ")
    assert WordCount.count(text) == count
  end
end
```

```elixir
def word do
  member_of([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

property "word-count" do
  check all count <- positive_integer()
            words <- list_of(word, length: count)
  do
    text = words |> Enum.join(" ")
    assert WordCount.count(text) == count
  end
end
```

```
def word do
  member_of([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

property "word-count" do
  check all count <- positive_integer()
            words <- list_of(word, length: count)
  do
    text = words |> Enum.join(" ")
    assert WordCount.count(text) == count
  end
end
```

```
def word do
  member_of([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

property "word-count" do
  check all count <- positive_integer()
            words <- list_of(word, length: count)
  do
    text = words |> Enum.join(" ")
    assert WordCount.count(text) == count
  end
end
```

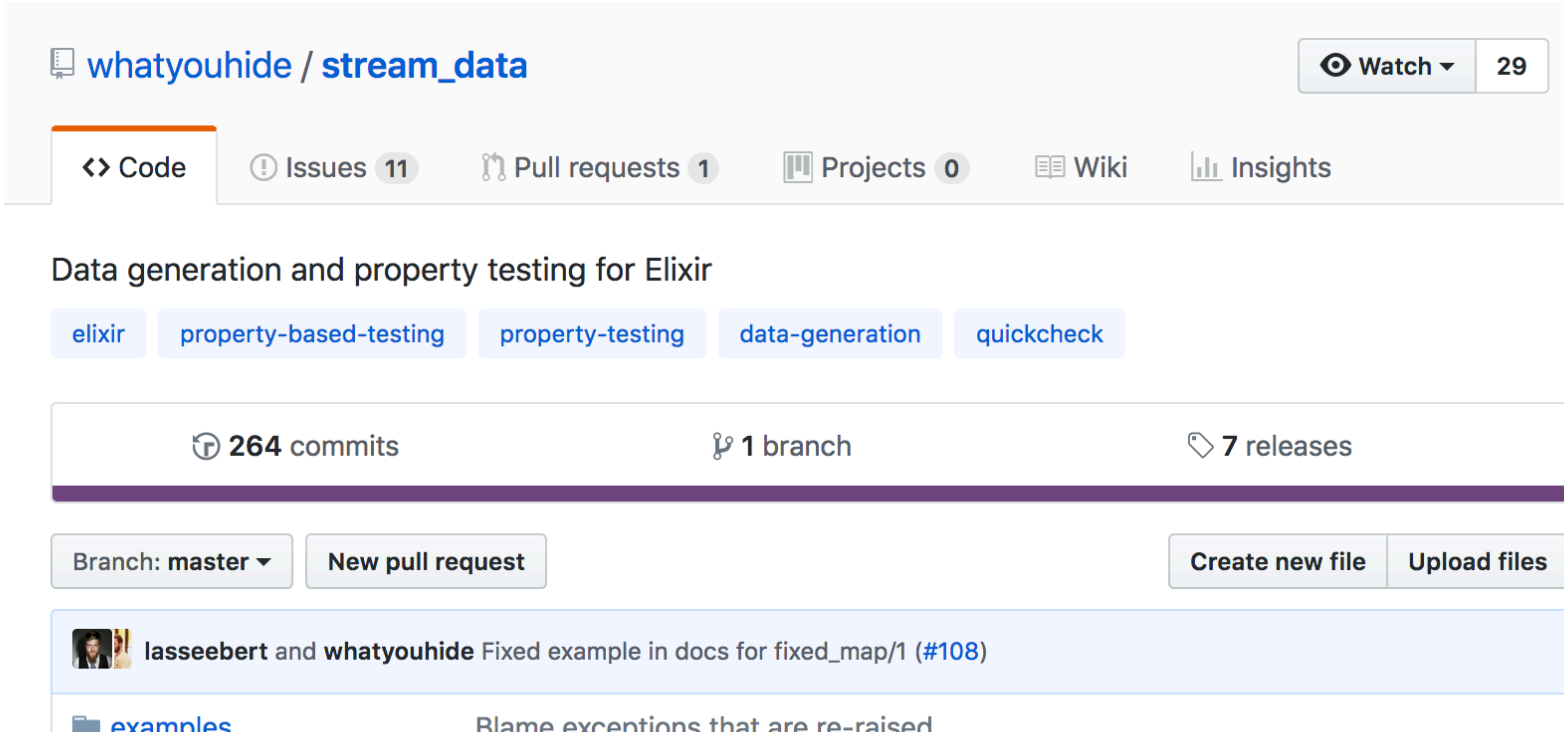
```
def word do
  member_of([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

property "word-count" do
  check all count <- positive_integer()
            words <- list_of(word, length: count)
  do
    text = words |> Enum.join(" ")
    assert WordCount.count(text) == count
  end
end
```

```elixir
def word do
  member_of([
    "This", "is", "a", "test.", "And", "this!"
  ])
end

property "word-count" do
  check all count <- positive_integer()
            words <- list_of(word, length: count)
  do
    text = words |> Enum.join(" ")
    assert WordCount.count(text) == count
  end
end
```

StreamData


whatyouhide / stream_data

Watch 29

Code | Issues 11 | Pull requests 1 | Projects 0 | Wiki | Insights

Data generation and property testing for Elixir

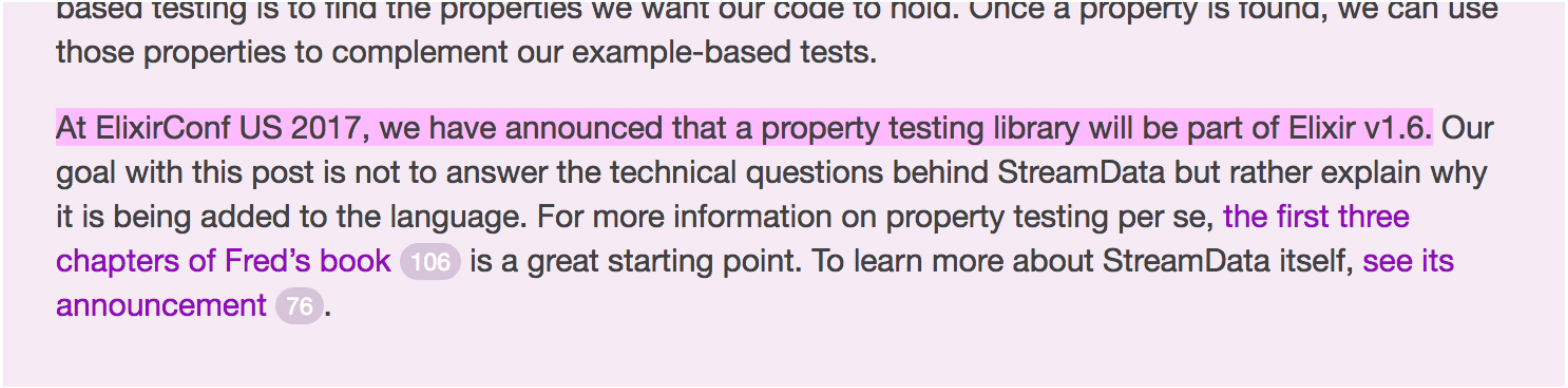
elixir property-based-testing property-testing data-generation quickcheck

264 commits 1 branch 7 releases

Branch: master New pull request Create new file Upload files

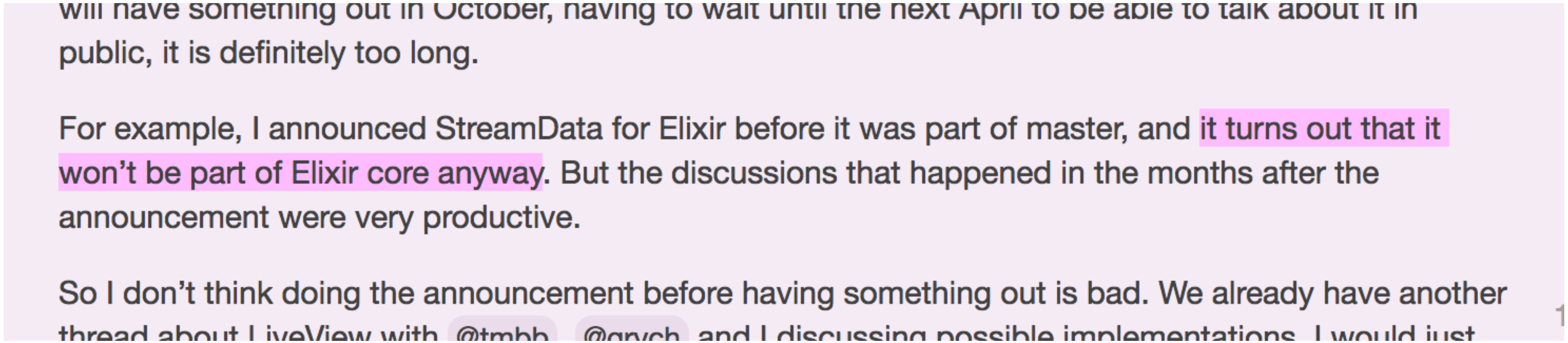
lasseebert and whatyouhide Fixed example in docs for fixed_map/1 (#108)

examples Blame exceptions that are re-raised



based testing is to find the properties we want our code to hold. Once a property is found, we can use those properties to complement our example-based tests.

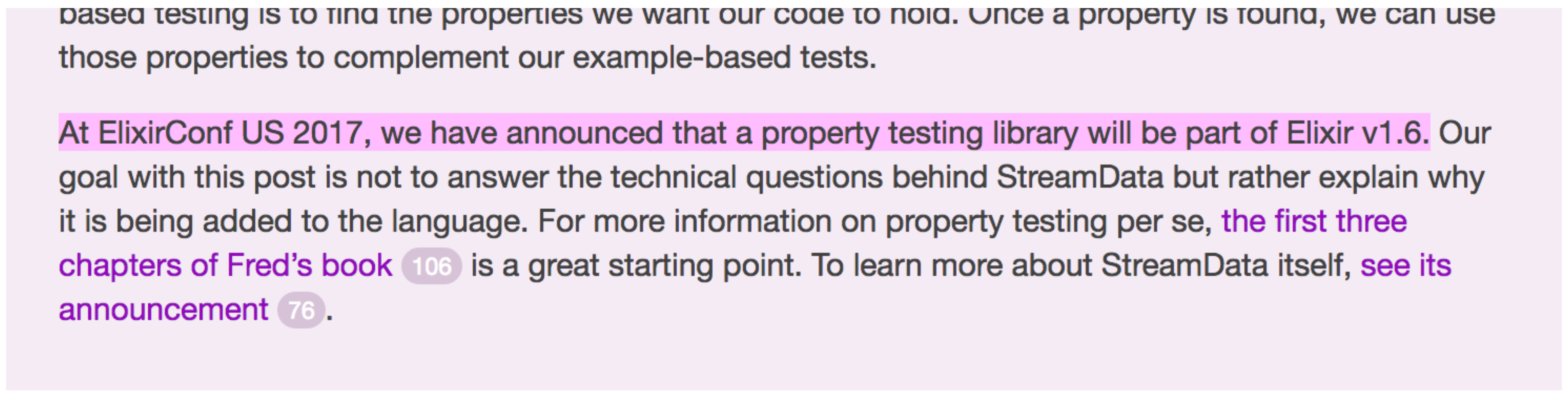
At ElixirConf US 2017, we have announced that a property testing library will be part of Elixir v1.6. Our goal with this post is not to answer the technical questions behind StreamData but rather explain why it is being added to the language. For more information on property testing per se, the first three chapters of Fred's book 106 is a great starting point. To learn more about StreamData itself, see its announcement 76.



will have something out in October, having to wait until the next April to be able to talk about it in public, it is definitely too long.

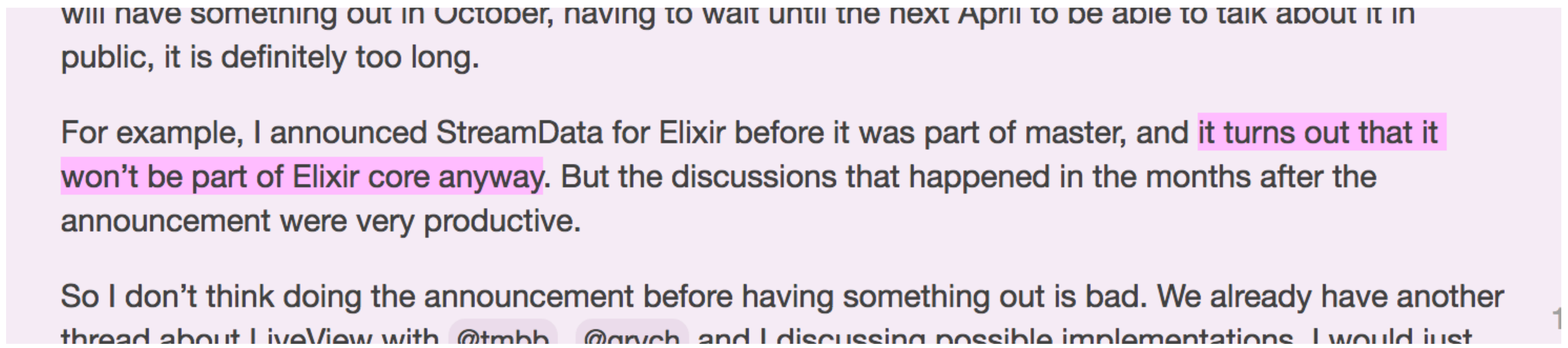
For example, I announced StreamData for Elixir before it was part of master, and it turns out that it won't be part of Elixir core anyway. But the discussions that happened in the months after the announcement were very productive.

So I don't think doing the announcement before having something out is bad. We already have another thread about LiveView with @tmbb, @grych and I discussing possible implementations. I would just



based testing is to find the properties we want our code to hold. Once a property is found, we can use those properties to complement our example-based tests.

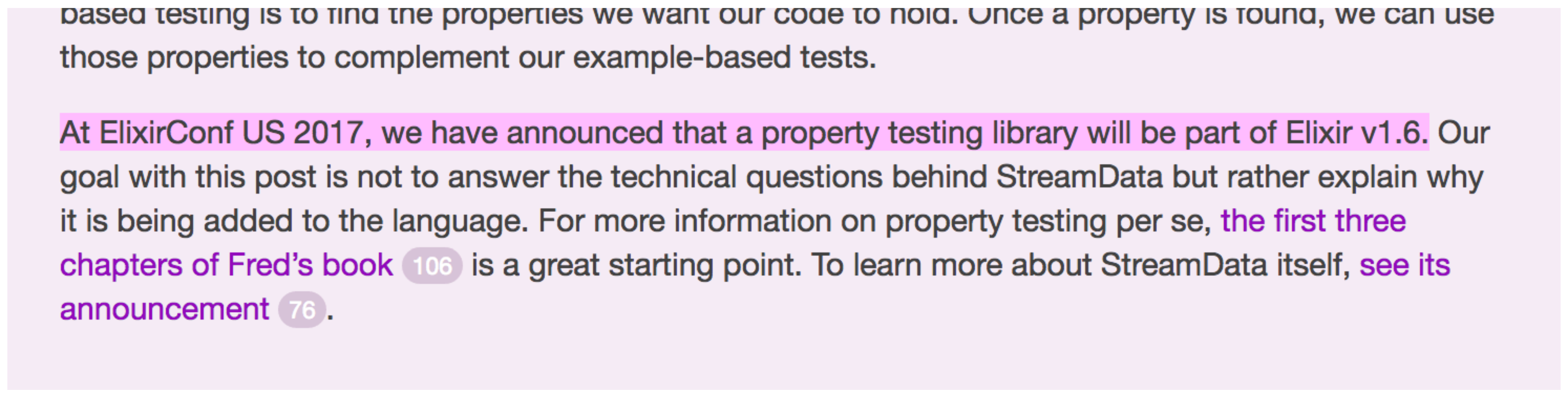
At ElixirConf US 2017, we have announced that a property testing library will be part of Elixir v1.6. Our goal with this post is not to answer the technical questions behind StreamData but rather explain why it is being added to the language. For more information on property testing per se, the first three chapters of Fred's book 106 is a great starting point. To learn more about StreamData itself, see its announcement 76.



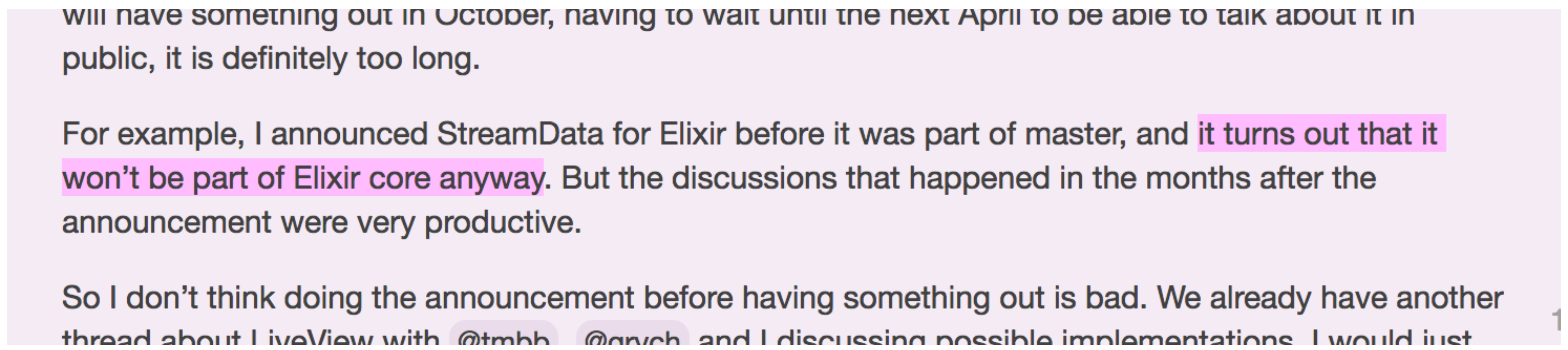
will have something out in October, having to wait until the next April to be able to talk about it in public, it is definitely too long.

For example, I announced StreamData for Elixir before it was part of master, and it turns out that it won't be part of Elixir core anyway. But the discussions that happened in the months after the announcement were very productive.

So I don't think doing the announcement before having something out is bad. We already have another thread about LiveView with @tmbb , @grych and I discussing possible implementations. I would just

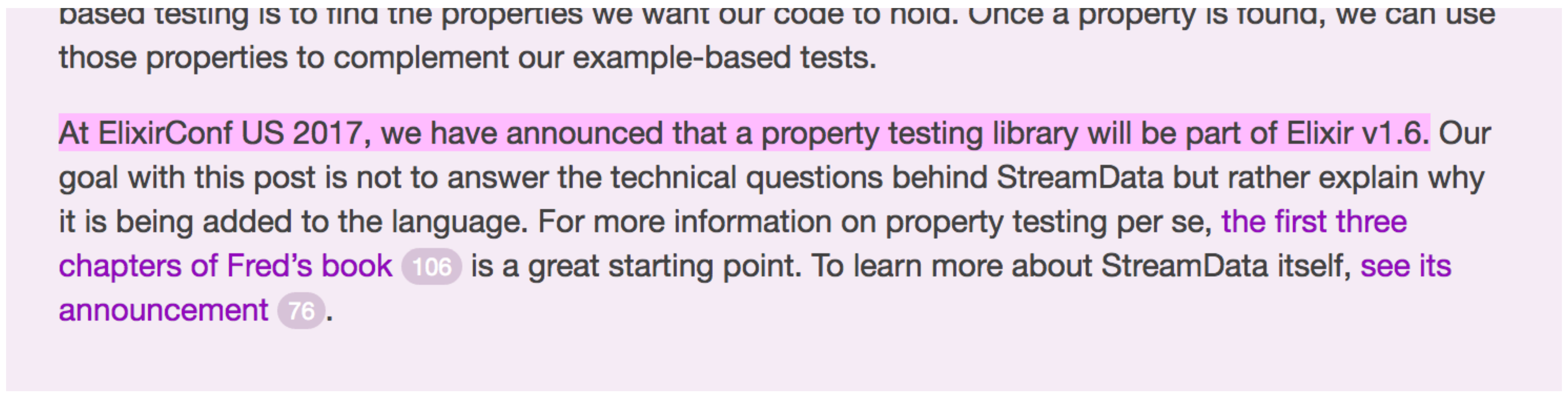

based testing is to find the properties we want our code to hold. Once a property is found, we can use those properties to complement our example-based tests.

At ElixirConf US 2017, we have announced that a property testing library will be part of Elixir v1.6. Our goal with this post is not to answer the technical questions behind StreamData but rather explain why it is being added to the language. For more information on property testing per se, the first three chapters of Fred's book 106 is a great starting point. To learn more about StreamData itself, see its announcement 76.


will have something out in October, having to wait until the next April to be able to talk about it in public, it is definitely too long.

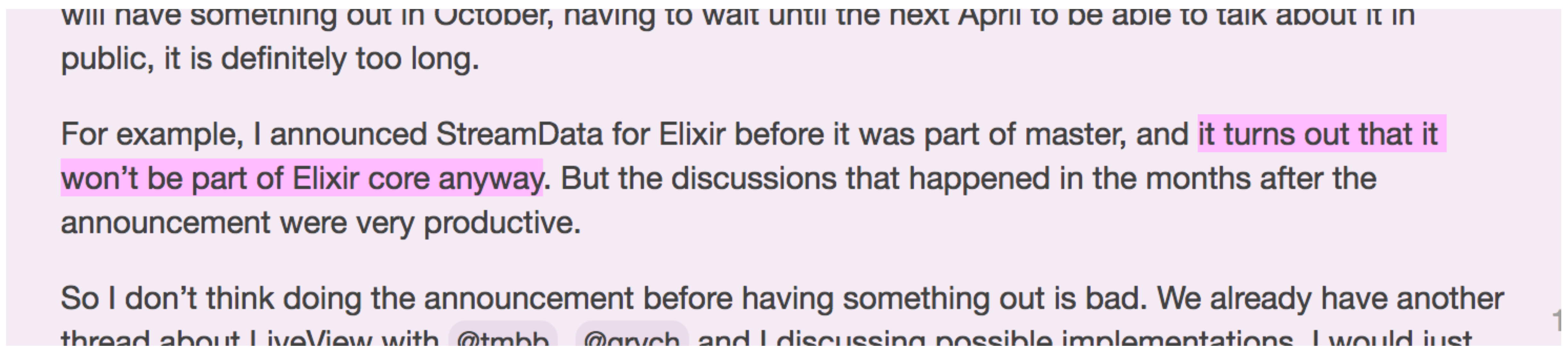
For example, I announced StreamData for Elixir before it was part of master, and it turns out that it won't be part of Elixir core anyway. But the discussions that happened in the months after the announcement were very productive.

So I don't think doing the announcement before having something out is bad. We already have another thread about LiveView with @tmbb , @grych and I discussing possible implementations. I would just



based testing is to find the properties we want our code to hold. Once a property is found, we can use those properties to complement our example-based tests.

At ElixirConf US 2017, we have announced that a property testing library will be part of Elixir v1.6. Our goal with this post is not to answer the technical questions behind StreamData but rather explain why it is being added to the language. For more information on property testing per se, the first three chapters of Fred's book 106 is a great starting point. To learn more about StreamData itself, see its announcement 76.



will have something out in October, having to wait until the next April to be able to talk about it in public, it is definitely too long.

For example, I announced StreamData for Elixir before it was part of master, and it turns out that it won't be part of Elixir core anyway. But the discussions that happened in the months after the announcement were very productive.

So I don't think doing the announcement before having something out is bad. We already have another thread about LiveView with @tmbb , @grych and I discussing possible implementations. I would just

Generators

```
import StreamData

example = constant("Hello, World")







check all greeting <- example
do
  assert greeting == "Hello, World"
end
```

```
import StreamData

example = constant("Hello, World")







check all greeting <- example
do
  assert greeting == "Hello, World"
end
```

```
import StreamData

example = constant("Hello, World")
# example :: StreamData.t(String.t)





check all greeting <- example
do
  assert greeting == "Hello, World"
end
```

```
import StreamData

example = constant("Hello, World")
# example :: StreamData.t(String.t)






check all greeting <- example
do
  assert greeting == "Hello, World"
end
```

```
import StreamData

example = constant("Hello, World")







check all greeting <- example
do
  assert greeting == "Hello, World"
end
```

```
import StreamData

example = constant("Hello, World")




check all greeting <- example
do
  assert greeting == "Hello, World"
end
```

```
import StreamData

example = constant("Hello, World")







check all greeting <- example
do
  assert greeting == "Hello, World"
end
```

```
import StreamData

word = member_of([
  "Hello",
  "World",
  "Tokyo",
  "Perth",
])

check all a_word <- word
do
  assert String.length(a_word) == 5
end
```

```
import StreamData

word = member_of([
  "Hello",
  "World",
  "Tokyo",
  "Perth",
])

check all a_word <- word
do
  assert String.length(a_word) == 5
end
```

```
import StreamData

word = member_of([...])
words = list_of(word)







check all original <- words
do
  ...
end
```

```
import StreamData

word = member_of([...])
words = list_of(word)




check all original <- words
do
  ...
end
```

```
import StreamData

word = member_of([...])
words = list_of(word)
sentence = list_of(words) |> ...





check all original <- words
do
  ...
end
```

```
import StreamData

word = member_of([...])
words = list_of(word)
sentence = list_of(words) |> ...




check all original <- words
do
  ...
end
```

```
import StreamData

def word, do: member_of([...])
def words, do: list_of(word())
def sentence do
  ...
end
```

```elixir
import StreamData

def word, do: member_of([...])
def words, do: list_of(word())
def sentence do
  gen all words <- words()
  do
    words
    |> Enum.join(" ")
    |> String.capitalize()
    |> Kernel.<>(".")
  end
end
```

```
import StreamData

def word, do: member_of([...])
def words, do: list_of(word())
def sentence do
  gen all words <- words()
  do
    words
    |> Enum.join(" ")
    |> String.capitalize()
    |> Kernel.<>(".")
  end
end
```

```
import StreamData

def word, do: member_of([...])
def words, do: list_of(word())
def sentence do
  gen all words <- words()
  do
    words
    |> Enum.join(" ")
    |> String.capitalize()
    |> Kernel.<>(".")
  end
end
```

```
import StreamData

def word, do: member_of([...])
def words, do: list_of(word())
def sentence do
  gen all words <- words()
  do
    words
    |> Enum.join(" ")
    |> String.capitalize()
    |> Kernel.<>(".")
  end
end
```

```
import StreamData

def word, do: member_of([...])
def words, do: list_of(word())
def sentence do
  gen all words <- words()
  do
    words
    |> Enum.join(" ")
    |> String.capitalize()
    |> Kernel.<>(".")
  end
end
```

```
import StreamData

def word, do: member_of([...])
def words, do: list_of(word())
def sentence do
  gen all words <- words()
          ending <- member_of(~W[. ! ? ‽])
  do
    words
    |> Enum.join(" ")
    |> String.capitalize()
    |> Kernel.<>(ending)
  end
end
```

```
import StreamData

def word, do: member_of([...])
def words, do: list_of(word())
def sentence do
  gen all words <- words()
          ending <- member_of(~W[. ! ? ‽])
  do
    words
    |> Enum.join(" ")
    |> String.capitalize()
    |> Kernel.<>(ending)
  end
end
```

```
def naive_datetime() do
  gen all date <- date(),
          time <- time()
  do
    {:ok, ndt} = NaiveDateTime.new(date, time) do
  end
end

def date do
  gen all year <- integer(2010..2040),
          {:ok, jan1} = Date.new(year, 1, 1),
          days <- integer(0..(if Date.leap_year?(jan1), do: 365, else: 364))
  do
    Date.add(jan1, days)
  end
end

def time do
  gen all hour <- integer(0..23),
          min <- integer(0..59),
          sec <- integer(0..59),
          usec <- integer(0..999_999)
  do
    {:ok, time} = Time.new(hour, min, sec, usec)
    time
  end
end
```

StreamData
v0.4.2

search

PAGES
MODULES
EXCEPTIONS

ExUnitProperties
Top
Summary
+ Functions

StreamData
Top
Summary
+ Types
+ Functions

Filters the given generator `data` *according to the given* `predicate` *function*

`fixed_list(datas)`

Generates a list of fixed length where each element is generated from the corresponding generator in `data`

`fixed_map(data)`

Generates maps with fixed keys and generated values

`float(options \\ [])`

Generates floats according to the given `options`

`frequency(frequencies)`

Generates values from different generators with specified probability

`integer()`

Generates integers bound by the generation size

`integer(range)`

Generates an integer in the given `range`

`iodata()`

Generates iodata

`iolist()`

Generates iolists

`keyword_of(value_data)`

Generates keyword lists where values are generated by `value_data`

`list_of(data)`

Generates lists where each values is generated by the given `data`

`list_of(data, options)`

Generates lists where each values is generated by the given `data`

`map(data, fun)`

Positives, Pains, and Sundry General Experiences

Generator Pitfalls

1)
Size of Generated Data

```
word = member_of([...])
words = list_of(word)
sentence = list_of(words)


check all data <- sentence,
          text = List.flatten(data)
                 |> Enum.join(" ")
do
  ...
end
```

```elixir
sentence =
  list_of(
    list_of(member_of([...]))
  )

check all data <- sentence,
          text = List.flatten(data)
                 |> Enum.join(" ")
do
  ...
end
```

```
sentence =
  list_of(
    list_of(member_of([...]))
  )

check all data <- sentence,
          text = List.flatten(data)
                 |> Enum.join(" ")
do
  ...
end
```

$O(n^2)$

Success

```
def smallish(generator) do
  generator
  |> SD.scale(fn size ->
       trunc(:math.log(size))
     end)
end

# ...

words    = smallish(list_of(word()))
sentence = smallish(list_of(words))
```

```
def smallish(generator) do
  generator
  |> SD.scale(fn size ->
       trunc(:math.log(size))
     end)
end

# ...

words    = smallish(list_of(word()))
sentence = smallish(list_of(words))
```

```
def smallish(generator) do
  generator
  |> SD.scale(fn size ->
       trunc(:math.log(size))
     end)
end

# ...

words    = smallish(list_of(word()))
sentence = smallish(list_of(words))
```

```
def smallish(generator) do
  generator
  |> SD.scale(fn size ->
       trunc(:math.log(size))
     end)
end

def biggish(generator) do
  generator
  |> SD.scale(fn size ->
       size * 99
     end)
end
```

2)
Reconstructing information after the fact is tough. So don't try to do that.

```
def word do
  Enum.random(["Hello", "World"])
end
def sentence do
  (1..Enum.random(1..10))
  |> Enum.map(fn _ -> word() end)
  |> Enum.join(" ")
  |> (&(&1 <> ".")).() # help
end

test "word-count" do
  sentence = sentence() # random sentence!
  assert WC.count(sentence) == ... # now what?
end
```

```
def word do
  Enum.random(["Hello", "World"])
end
def sentence do
  (1..Enum.random(1..10))
  |> Enum.map(fn _ -> word() end)
  |> Enum.join(" ")
  |> (&(&1 <> ".")).() # help
end

test "word-count" do
  sentence = sentence() # random sentence!
  assert WC.count(sentence) == ... # now what?
end
```

```
def word do
  member_of(["This", "is", "a", ...])
end




property "word-count" do
  check all count <- positive_integer()
            words <- list_of(word, length: count)
  do
    text = words |> Enum.join(" ")
    assert WordCount.count(text) == count
  end
end
```

```
def word do
  member_of(["This", "is", "a", ...])
end



property "word-count" do
  check all count <- positive_integer()
            words <- list_of(word, length: count)
  do
    text = words |> Enum.join(" ")
    assert WordCount.count(text) == count
  end
end
```

3)
Timex and/or Timezones

```
def naive_datetime() do
  gen all date <- date(),
          time <- time()
  do
    {:ok, ndt} = NaiveDateTime.new(date, time) do
  end
end

def date do
  gen all year <- integer(2010..2040),
          {:ok, jan1} = Date.new(year, 1, 1),
          days <- integer(0..(if Date.leap_year?(jan1), do: 365, else: 364))
  do
    Date.add(jan1, days)
  end
end

def time do
  gen all hour <- integer(0..23),
          min <- integer(0..59),
          sec <- integer(0..59),
          usec <- integer(0..999_999)
  do
    {:ok, time} = Time.new(hour, min, sec, usec)
    time
  end
end
```

```
def naive_datetime() do
  gen all date <- date(),
          time <- time()
  do
    {:ok, ndt} = NaiveDateTime.new(date, time) do
  end
end

def potentially_ambiguous_datetime() do
  gen all naive <- naive_datetime()
  do
    Timex.to_datetime(naive, "America/Los_Angeles")
  end
end
```

```
Failed with generated values (after 100 runs):
...

left:
  %{timestamp: #DateTime<
    2024-11-03 01:20:57-07:00 PDT
    America/Los_Angeles
    >}

right:
  %{timestamp: #<Ambiguous(
    #DateTime<
      2024-11-03 01:20:57-07:00 PDT
      America/Los_Angeles
    > ~ #DateTime<
      2024-11-03 01:20:57-08:00 PST
      America/Los_Angeles
    >)>
   }
```

Round-Tripping

```elixir
alias Example.Health.Json

describe "JSON round-trips" do
  property "without IDs" do
    check all reports <- list_of(Gen.Health.health_report(nil))
    do
      assert Json.parse(Json.generate(reports)) == {:ok, reports}
    end
  end

  property "with IDs" do
    report_gen = Gen.Health.health_report(nil) |> Gen.Id.with_id()
    check all reports <- list_of(report_gen)
    do
      assert Json.parse(Json.generate(reports)) == {:ok, reports}
    end
  end
end
```

```elixir
def parse(list_of_maps) do
  ...
  # {:ok, [%HealthReport{}, %HealthReport{}, ...]}
  # or
  # {:error, ...}
end

def generate(reports) do
  ...
  # [%{"id" => ..., "attributes" => ...}, ...]
end
```

Idempotency

1. Generate data to send.
2. Make controller request.
3. Make controller request.
4. Retrieve models.
5. ... Clean up! ⚠️ (Remove DB rows)
6. Then `assert` your expected result.

1. Generate data to send.
2. Make 1+ controller requests.

3. Retrieve models.
4. … Clean up! ⚠️ (Remove DB rows)
5. Then `assert` your expected result.

1. Generate data to send.
2. **Make 1+ controller requests.**

3. Retrieve models.
4. … Clean up! ⚠️ (Remove DB rows)
5. Then `assert` your expected result.

1. Generate data to send.
2. Make 1+ controller requests.

3. Retrieve models.
4. **… Clean up! ⚠️ (Remove DB rows)**
5. Then `assert` your expected result.

(Shrinking)

Shrinking replays a test, backtracking on the input data, until the test starts passing again.

But ExUnitProperties runs <u>inside</u> a test case. DB data is not reset between runs.

Merde

1. Generate data to send.
2. Make 1+ controller requests.

3. Retrieve models.
4. **... Clean up! ⚠️ (Remove DB rows)**
5. Then `assert` your expected result.

Shrinking Woes

```
created_at: nil, day_starting_at: #DateTime<2021-04-01 23:53:29.047627Z>, exercise_seconds: nil, id: nil,
runwalk_meters: 45576, steps_count: nil, updated_at: nil, user:
#Ecto.Association.NotLoaded<association :user is not loaded>, user_id: nil, weights:
[%Example.HealthReport.WeightMeasurement{__meta__: #Ecto.Schema.Metadata<:built, "health_report_weights">,
created_at: nil, health_report: #Ecto.Association.NotLoaded<association :health_report is not loaded>,
health_report_id: nil, id: nil, timestamp: #DateTime<2023-10-01 02:50:52.461705+11:00 AEDT Australia/
Sydney>, updated_at: nil, weight_grams: 2992005}, %Example.HealthReport.WeightMeasurement{__meta__:
#Ecto.Schema.Metadata<:built, "health_report_weights">, created_at: nil, health_report:
#Ecto.Association.NotLoaded<association :health_report is not loaded>, health_report_id: nil, id: nil,
timestamp: #DateTime<2013-05-31 06:38:36.865466Z>, updated_at: nil, weight_grams: 2286711}]},
%Example.HealthReport{__meta__: #Ecto.Schema.Metadata<:built, "health_reports">, burnt_joules: 142,
created_at: nil, day_starting_at: #DateTime<2037-10-27 20:32:03.518323Z>, exercise_seconds: nil, id: nil,
runwalk_meters: 24141, steps_count: nil, updated_at: nil, user:
#Ecto.Association.NotLoaded<association :user is not loaded>, user_id: nil, weights: []},
%Example.HealthReport{__meta__: #Ecto.Schema.Metadata<:built, "health_reports">, burnt_joules: 20942,
created_at: nil, day_starting_at: #DateTime<2034-08-20 10:59:11.639846Z>, exercise_seconds: 11893, id: nil,
runwalk_meters: 17910, steps_count: 24013, updated_at: nil, user:
#Ecto.Association.NotLoaded<association :user is not loaded>, user_id: nil, weights:
[%Example.HealthReport.WeightMeasurement{__meta__: #Ecto.Schema.Metadata<:built, "health_report_weights">,
created_at: nil, health_report: #Ecto.Association.NotLoaded<association :health_report is not loaded>,
health_report_id: nil, id: nil, timestamp: #DateTime<2022-09-25 04:24:59.049975-07:00 PDT America/
Los_Angeles>, updated_at: nil, weight_grams: 2891106}, %Example.HealthReport.WeightMeasurement{__meta__:
#Ecto.Schema.Metadata<:built, "health_report_weights">, created_at: nil, health_report:
#Ecto.Association.NotLoaded<association :health_report is not loaded>, health_report_id: nil, id: nil,
timestamp: #DateTime<2019-03-12 21:30:26.737857-07:00 PDT America/Los_Angeles>, updated_at: nil,
weight_grams: 1042956}, %Example.HealthReport.WeightMeasurement{__meta__: #Ecto.Schema.Metadata<:built,
"health_report_weights">, created_at: nil, health_report:
#Ecto.Association.NotLoaded<association :health_report is not loaded>, health_report_id: nil, id: nil,
timestamp: #DateTime<2029-02-22 20:36:40.061115Z>, updated_at: nil, weight_grams: 34965},
%Example.HealthReport.WeightMeasurement{__meta__: #Ecto.Schema.Metadata<:built, "health_report_weights">,
created_at: nil, health_report: #Ecto.Association.NotLoaded<association :health_report is not loaded>,
health_report_id: nil, id: nil, timestamp: #DateTime<2019-04-14 03:56:08.321487-07:00 PDT America/
Los_Angeles>, updated_at: nil, weight_grams: 458541}]}, %Example.HealthReport{__meta__:
#Ecto.Schema.Metadata<:built, "health_reports">, burnt_joules: nil, created_at: nil, day_starting_at:
```

Factories
vs
Generators
vs
... both?

Macros

Macros

are

Composition-Resistant

```
check all thing <- thingy(:thing),
          max_runs: 10
do
  expensive!(thing)
  assert thing == thing
end
```

```
check all thing <- thingy(:thing),
          max_runs: 10
do
  expensive!(thing)
  assert thing == thing
end
```

```
slowcheck thing <- thingy(:thing)

do
  expensive!(thing)
  assert thing == thing
end
```

```
defmacro slowcheck(...), do: ...
```

```
slowcheck thing <- thingy(:thing)

do
  expensive!(thing)
  assert thing == thing
end
```

Bugs Caught

this one is a bit tough to give a useful answer for

It's a design tool.

1)
JSON round-tripping

(Parsing bug, generator bug, parsing bug, ...)

2)
Idempotent actions

(Finding bugs with the conflict-resolution for inserting.)

3)
Ruby example: zips

Round-Tripping

```
property_of { char, integer }.check { |char,size|
  file = File.join(tmpdir, "testfile-#{size}.bin")
  zip  = File.join(tmpdir, "testfile-#{size}.zip")

  data_write = char * size  # size-length string, all char.
  filename   = char * size

  File.open(file, 'wb') { |f| f.write(data_write) }
  Zip::File.open(zip, CREATE) {|f| f.add(filename, file) }

  data_read = nil
  Zip::File.open(zip) {|f|
    data_read = f.first.get_input_stream.read
  }

  expect(data_write).to == data_read
}
```

Round-Tripping

```
Size: 65535 - Gen'd, Written,
Zipped, Unzipped. Written data
equals read data.
```

```
Size: 65536 - Gen'd, Written,
Zipped, /Users/rhoward/code/
experiments/p7zip/rubyzip/lib/
zip/inflater.rb:44:in `inflate':
invalid stored block lengths
(Zlib::DataError)
```

Round-Tripping

```
$ 7z x testfile-65536.zip
7-Zip [64] …

Processing archive: testfile-65536.zip

Errors: Headers Error
Errors: Unconfirmed start of archive
Warnings: There are data after the end of
archive

Extracting testfile-65536: Segmentation fault
```

Round-Tripping

```
$ 7z x testfile-65536.zip
7-Zip [64] …

Processing archive: testfile-65536.zip

Errors: Headers Error
Errors: Unconfirmed start of archive
Warnings: There are data after the end of
archive

Extracting testfile-65536: Segmentation fault
```

so what's the answer then

<u>Yes</u>, if you have someone on your team who's already used property testing before.

Tentative yes, if you don't have that person. Keep it to isolated cases, to try it out, so you can rip it out later.

No, if you don't want to go off the beaten path.

Start with things that are easy to generate data for, or you want to test the crap out of.

How to draw an owl

1.

2.

1. Draw some circles

2. Draw the rest of the fucking owl

A draft experience report
on the use of StreamData for

Property Testing

in Elixir
at my last job

Rob Howard
@damncabbage
http://robhoward.id.au

Things To Read

- Proper Testing
  https://propertesting.com/
- StreamData docs
  https://hexdocs.pm/stream_data/StreamData.html