

Exop on StreamData

Andrey Chernykh

Code BEAM STO 2019

Me: quick facts

C# 5 years ▷ Ruby 4 years ▷ Elixir (since 2016)

Number of services (not all of them are micro) in production.

 <https://github.com/madeinussr>

 <https://medium.com/@andreichernykh>


Currently: Elixir developer at Coingaming Group 

Contents

#CodeBEAMSTO


Evolution of an idea: from
inspiration to the result

Contents

 what is Exop?

 why Exop?

 ElixirConf EU & StreamData

 idea

 result

🤔 What is Exop?

Elixir library that provides a macros which allow you to **encapsulate** business logic and **validate** incoming parameters with predefined **contract**.

since 2016





Encapsulation

```
defmodule IntegersDivision do
  use Exop.Operation

  parameter :a, type: :integer, default: 1, required: false
  parameter :b, type: :integer, numericality: %{greater_than: 0}

  def process(params) do
    result = params[:a] / params[:b]
    IO.inspect "The division result is: #{result}"
  end
end
```



Contract

```
defmodule IntegersDivision do
  use Exop.Operation

  parameter :a, type: :integer, default: 1, required: false
  parameter :b, type: :integer, numericity: %{greater_than: 0}

  def process(params) do
    result = params[:a] / params[:b]
    IO.inspect "The division result is: #{result}"
  end
end
```



Contract

```
parameter :user_email, type: :string, format: ~r/@/
```

```
parameter :items, type: :list, length: %{min: 1}, list_item: %{  
  inner: %{  
    "name" => [type: :string, length: %{min: 1}],  
    "price" => [type: :float, numericality: %{greater_than: 0}],  
    "quantity" => [type: :integer, numericality: %{greater_than: 0}]  
  }  
}
```


Validation & Unified output

```
parameter :a, type: :integer, default: 1, required: false  
parameter :b, type: :integer, numericality: %{greater_than: 0}
```

```
iex> IntegersDivision.run(a: 50, b: 5)  
{:ok, "The division result is: 10"}
```

```
iex> IntegersDivision.run(a: "50", b: 5)  
{:error, {:validation, %{a: ["has wrong type"]}}}
```

Validation & Unified output

```
parameter :a, inner: %{b: [type: :atom], c: [type: :string]}
```

```
iex> YourOperation.run(a: :a)
```

```
{:error, {:validation, %{a: ["has wrong type"]}}}
```

```
iex> YourOperation.run(a: %{})
```

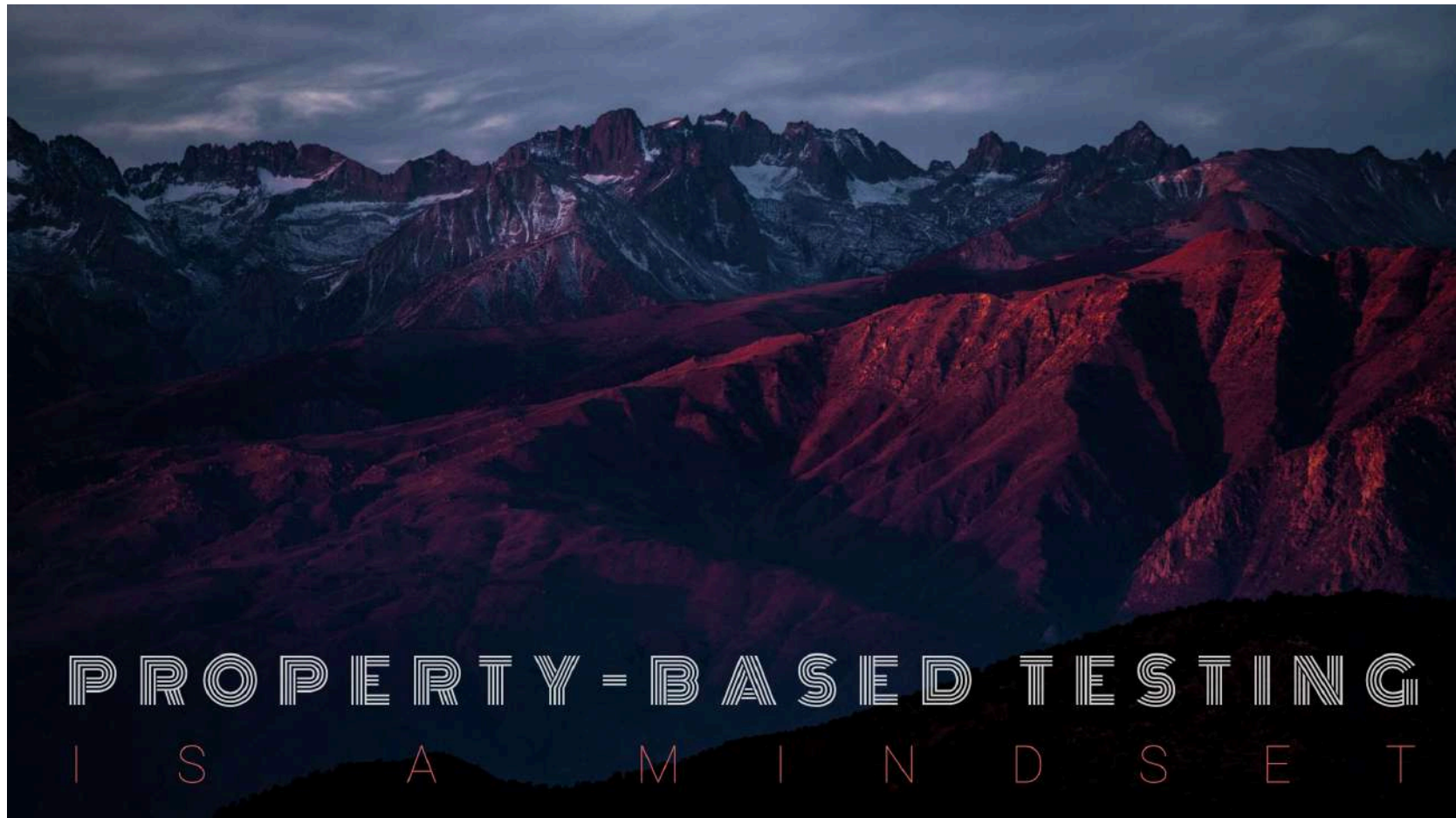
```
{:error, {:validation, %{ "a[:b]" => ["is required"], "a[:c]" => ["is required"]}}}
```

 Playtime

💖 Other Exop features

- Parameter coercion
- Invocation interruption
- Policy check
- Fallbacks
- Operations chain

ElixirConfEU 2018 & StreamData



```
StreamData.integer() ▷ Stream.map(&abs/1) ▷ Enum.take(3)
```

```
#⇒ [1, 0, 2]
```



StreamData

#CodeBEAMSTO

```
require ExUnitProperties
```

```
domains = [
```

```
  "gmail.com",
```

```
  "hotmail.com",
```

```
  "yahoo.com",
```

```
]
```

```
email_generator =
```

```
  ExUnitProperties.gen all name ← StreamData.string(:alphanumeric),
```

```
  name ≠ "",
```

```
  domain ← StreamData.member_of(domains) do
```

```
    name <> "@" <> domain
```

```
  end
```

```
Enum.take(StreamData.resize(email_generator, 20), 2)
```

```
#=> ["efsT6Px@hotmail.com", "swEowmk7mW0VmkJDF@yahoo.com"]
```

StreamData & Property-based testing

```
property "bin1 <> bin2 always starts with bin1" do
  check all bin1 ← binary(),
    | | | | | bin2 ← binary() do
      assert String.starts_with?(bin1 <> bin2, bin1)
    end
  end
end
```

👉 StreamData & Typespecs

stream_data
+
dialyzer



Generators from type

```
@type timeout() :: :infinity | non_neg_integer()
```

```
from_type(timeout())
```

```
one_of([:infinity, map(integer(), &abs/1)])
```


👉 StreamData & Typespecs

stream_data
+
dialyzer

generators from type

```
@type timeout() :: :infinity | non_
```

```
from_type(timeout())
```

```
one_of([:infinity, map(integer(), &abs/1)])
```





...wait

```
defmodule IntegersDivision do
  use Exop.Operation

  parameter :a, type: :integer, default: 1, required: false
  parameter :b, type: :integer, numericality: %{greater_than: 0}

  def process(params) do
    result = params[:a] / params[:b]
    IO.inspect "The division result is: #{result}"
  end
end
```



...wait...wait...wait

```
defmodule IntegersDivision do
  use Exop.Operation

  parameter :a, type: :integer, default: 1, required: false
  parameter :b, type: :integer, numericality: %{greater_than: 0}

  def process(params) do
    result = params[:a] / params[:b]
    IO.inspect "The division result is: #{result}"
  end
end
```



Eureka!

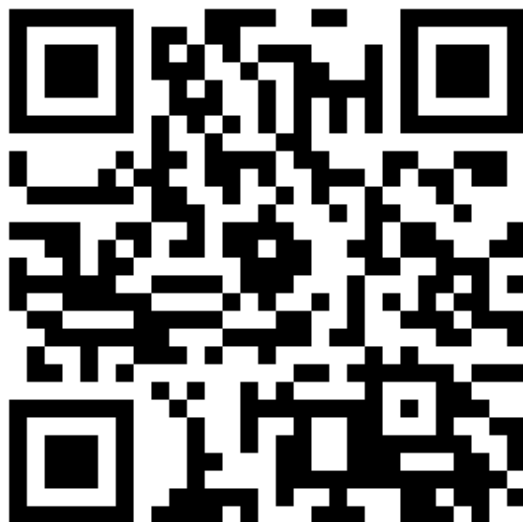
```
defmodule IntegersDivision do
  use Exop.Operation


  parameter :a, type: :integer, default: 1, required: false
  parameter :b, type: :integer, numericality: %{greater_than: 0}

  def process(params) do
    result = params[:a] / params[:b]
    IO.inspect "The division result is: #{result}"
  end
end
```

ExopData

The goal of this library is to help you to write **property-based** tests by utilizing the power of **Exop** (and its contracts) and **StreamData**.



 ExopData: contract

```
[  
  %{name: param_a, opts: [type: :atom, required: false]},  
  %{name: param_b, opts: [type: :integer, numericality: %{min: 0, max: 10}]},  
  # more params here  
]
```



ExopData: data generators

```
contract = [  
  %{name: :a, opts: [type: :integer, numericality: %{greater_than: 0}]},  
  %{name: :b, opts: [type: :integer, numericality: %{greater_than: 10}]}  
]
```

```
#iex> contract ▷ ExopData.generate() ▷ Enum.take(5)
```

```
[  
  %{a: 3808, b: 3328},  
  %{a: 7116, b: 8348},  
  %{a: 3432, b: 7134},  
  %{a: 7024, b: 7941},  
  %{a: 7941, b: 6944}  
]
```



ExopData: data generators

```
defmodule MultiplyService do
  use Exop.Operation

  parameter(:a, type: :integer, numericality: %{greater_than: 0})
  parameter(:b, type: :integer, numericality: %{greater_than: 10})

  def process(%{a: a, b: b} = _params), do: a * b
end
```

```
#iex> MultiplyService ▷ ExopData.generate() ▷ Enum.take(5)
```

```
[
  %{a: 401, b: 2889},
  %{a: 7786, b: 5894},
  %{a: 9187, b: 1863},
  %{a: 3537, b: 1285},
  %{a: 6124, b: 5521}
]
```




ExopData: crazy generators

```

contract = [
  %{
    name: :complex_param,
    opts: [
      type: :map, inner: %{
        a: [type: :integer, numericality: %{in: 10..100}],
        b: [type: :list, length: %{min: 1}, list_item: %{
          type: :map, inner: %{
            c: [type: :list, list_item: %{
              type: :list, list_item: %{
                type: :map, inner: %{
                  d: [type: :string, length: %{is: 12}]
                }
              }
            ]
          }
        ]
      }
    ]
  }
]

```

#CodeBEAMSTO

```

%{
  complex_param: %{
    a: -5,
    b: [
      %{
        Hvj: "g@o2",
        QY: "@",
        Qb: ")\ I(",
        _: "",
        c: [
          [
            %{b: "Bqic.", d: "ZsyVe<ofu$0C", qq: "a"},
            %{b: "09", d: "711\#|DA?#s%", qq: "ZVTY"}
          ],
          [
            %{b: "@w", d: ">f+l ^}Qy!;", qq: "+Q*"},
            %{b: "u{P!", d: "P3t(IJ>`Hn9L", qq: ""},
            %{b: "Z", d: "/%p:A$UNn%GU", qq: "gd+"}
          ],
          [
            %{b: "WfU", d: "CKq2<km-M4L", qq: "3"},
            %{b: "\", d: "87/U3Q2SnqT-", qq: "nEw0h"}
          ],
          [
            %{b: "", d: "12Yw,1E BRyX", qq: ""},
            %{b: "ZNFm", d: "bLb7t| H5}Z ", qq: "F$e"},
            %{b: "zc'U1", d: "3>Gm 0@;0E,1", qq: "\\\\"}
          ]
        ],
        f: "",
        iXU: "",
        nSGR: "*\\Z"
      },
      %{
        Hvj: "46",
        QY: "h4`}Z",
        Qb: "t;90W",
        _: "",
        c: [
          [
            %{b: "[uAW", d: "40e3UD'.N'cw", qq: "c3'"},
            %{b: "", d: "fjHNpd\\1(nCl", qq: "[0'oI"},
            %{b: ")o", d: ";%u0{v5V(h;6", qq: "Dq"}
          ]
        ],
        f: "C\\QB",
        iXU: "",
        nSGR: "m"
      }
    ]
  }
}

```

 Playtime



ExopData: property-based testing

```
defmodule IntegersDivision do
  use Exop.Operation

  parameter :a, type: :integer, default: 1
  parameter :b, type: :integer, required: true,
    numericality: %{:greater_than: 0}

  def process(params) do
    result = params[:a] / params[:b]
    IO.inspect "The division result is: #{result}"
  end
end
```



...properly



ExopData: property-based testing

#CodeBEAMSTO

```
defmodule MultiplyService do
  use Exop.Operation

  parameter(:a, type: :integer, numericality: %{greater_than: 0})
  parameter(:b, type: :integer, numericality: %{greater_than: 10})

  def process(%{a: a, b: b} = _params), do: a * b
end

defmodule ExopPropsTest do
  use ExUnit.Case, async: true
  use ExUnitProperties

  property "Multiply" do
    check all %{a: a, b: b} = params ← ExopData.generate(MultiplyService) do
      {:ok, result} = MultiplyService.run(params)
      expected_result = a * b
      assert result == expected_result
    end
  end
end
```



ExopData: property-based testing

```
property "Multiply" do
  check_operation(MultiplyService, [], fn params →
    assert is_integer(params.a)

    {:ok, params.a * params.b}
  end)
end
```



ExopData: property-based testing

```
property "Multiply" do
  check_operation(MultiplyService, [], &({:ok, &1.a - &1.b}))
end
```

1) property Multiply (ExopPropTest)

test/exop_data/exop_prop_test.exs:14

Failed with generated values (after 0 successful runs):

```
* Clause:    params <- ExopData.generate(operation, opts)
```

```
Generated:  %{a: 1, b: 0}
```

Assertion with == failed

```
code:  assert operation_result == expected_result
```

```
left:  {:ok, 0}
```

```
right: {:ok, 1}
```



ExopData: custom generators

```
property "Multiply" do
  custom_generators = %{
    a: 1, # StreamData.constant(1)
    b: StreamData.integer(11..21)
  }

  check_operation(MultiplyService, [generators: custom_generators], fn params →
    assert params.a == 1
    assert params.b ≥ 11
    assert params.b ≤ 21

    {:ok, params.a * params.b}
  end)
end
```

 Playtime



ExopData: under the hood

```
defmodule ExopData.Generator do
  @moduledoc """
  Defines ExopData generators behaviour.

  An ExopData's generator should define `generate/1` function
  which takes a contract's parameter options with your property test options
  and returns StreamData generator made with respect to the options.
  """

  @callback generate(map(), map()) :: StreamData.t()
end
```



ExopData: under the hood

#CodeBEAMSTO

```
defmodule ExopData.Generators.String do
  @moduledoc """
  Implements ExopData generators behaviour for `string` parameter type.
  """

  @behaviour ExopData.Generator

  def generate(opts \\ %{}, _props_opts \\ %{}) do
    opts ▷ Map.get(:length) ▷ do_generate()
  end

  defp do_generate(%{is: exact}), do: StreamData.string(:ascii, length: exact)

  defp do_generate(%{in: min..max}) do
    StreamData.string(:ascii, min_length: min, max_length: max)
  end

  defp do_generate(%{min: min, max: max}) do
    StreamData.string(:ascii, min_length: min, max_length: max)
  end

  defp do_generate(%{min: min}), do: StreamData.string(:ascii, min_length: min)

  defp do_generate(%{max: max}), do: StreamData.string(:ascii, max_length: max)

  defp do_generate(_), do: StreamData.string(:ascii)
end
```



ExopData: under the hood

```
def generate(contract, props_opts) when is_list(contract) do
  optional_keys = optional_fields(contract)

  contract
  ▷ Enum.into(%{}, &generator_for_param(&1, props_opts))
  ▷ CommonGenerators.map(optional_keys)
end
```



ExopData: under the hood

```
def generator_for_opts(%{equals: value}, _props_opts), do: resolve_exact(value)

def generator_for_opts(%{exactly: value}, _props_opts), do: resolve_exact(value)

def generator_for_opts(%{in: _values} = opts, _props_opts), do: resolve_in_list(opts)

def generator_for_opts(%{format: regex}, _opts), do: resolve_format(regex)

def generator_for_opts(%{regex: regex}, _opts), do: resolve_format(regex)

defp resolve_exact(value), do: StreamData.constant(value)
```



ExopData: under the hood

#CodeBEAMSTO

```
defp run_generator(param_opts, opts) do
  param_type = param_type(param_opts)

  generator_module =
    [
      ExopData.Generators,
      param_type ▷ Atom.to_string() ▷ String.capitalize()
    ]
    ▷ Module.concat()

  if Code.ensure_compiled?(generator_module) do
    generator_module
    ▷ apply(:generate, [param_opts, opts])
    ▷ CommonFilters.filter(param_opts)
  else
    raise("""
    ExopData: there is no generator for params of type :#{param_type},
    you can add your own clause for such params
    """)
  end
end
```



ExopData: under the hood

```
# CommonGenerators.map/2 function makes the final StreamData.fixed_map generator
```

```
def map(data_map, optional_keys) do
  required_keys = Map.keys(data_map) -- optional_keys
  optional_keys_data = sublist(optional_keys)

  new(fn seed, size →
    {seed1, seed2} = split_seed(seed)
    subkeys_tree = call(optional_keys_data, seed1, size)

    data_map
    ▷ Map.take(required_keys ++ subkeys_tree.root)
    ▷ StreamData.fixed_map()
    ▷ call(seed2, size)
    ▷ LazyTree.map(fn fixed_map →
      LazyTree.map(subkeys_tree, fn keys →
        Map.take(fixed_map, required_keys ++ keys)
      end)
    end)
    ▷ LazyTree.flatten()
  end)
end
```



ExopData: under the hood

```
defmodule MultiplyService do
  use Exop.Operation

  parameter :a, type: :integer, numericality: %{gt: 0}
  parameter :b, type: :integer, numericality: %{gt: 10}

  def process(params), do: params.a * params.b
end
```

```
StreamData.fixed_map(%{
  a: StreamData.integer(), # + opts filters
  b: StreamData.integer(), # + opts filters
})
```

Limitations

- `struct` parameter is populated with a structure of random data
- data generating based on regex may be very slow
- complex generators may be slow (list in a map, within a list..)
- property tests of some operations may be slow (for ex. DB)



Limitations

- struct parameter is populated with a structure of random data
 - data generating based on regex may be very slow
 - complex generators may be slow (list in a map, within a list...)
 - property tests of some operations may be slow (for ex. DB)
- Do not try to replace all unit tests
with property-based tests**



Plans

- operations chain support
- parameter's coercion support
- consider PropEr as possible generator
- ...and of course constant improvements based on real usage feedback

🚩 OSS three main words

Learn, Share, Contribute



Exop



ExopData

Special credits to: Aleksandr Fomin ([llxff](#))

Thank ▶ you