

---

## Norm

Norm is a system for specifying the structure of data. It can be used for validation and for generation of data. Norm does not provide any set of predicates and instead allows you to re-use any of your existing validations.

```
1 import Norm
2
3 iex> conform!(123, spec(is_integer() and &(&1 > 0)))
4 123
5
6 iex> conform!(-50, spec(is_integer() and &(&1 > 0)))
7 ** (Norm.MismatchError) Could not conform input:
8 val: -50 fails: &(&1 > 0)
9
10 iex> user_schema = schema(%{
11 ...>   user: schema(%{
12 ...>     name: spec(is_binary()),
13 ...>     age: spec(is_integer() and &(&1 > 0))
14 ...>   })
15 ...> })
16 iex> input = %{user: %{name: "chris", age: 30, email: "c@keathley.io"}}
17 iex> conform!(input, user_schema)
18 %{user: %{name: "chris", age: 30, email: "c@keathley.io"}}
19 iex> generated_users =
20 ...>   user_schema
21 ...>   |> gen()
22 ...>   |> Enum.take(3)
23 iex> for g <- generated_users, do: g.user.age > 0 && is_binary(g.user.
    name)
24 [true, true, true]
```

Norm can also be used to specify contracts for function definitions:

```
1 defmodule Colors do
2   use Norm
3
4   def rgb(), do: spec(is_integer() and &(&1 in 0..255))
5
6   def hex(), do: spec(is_binary() and &String.starts_with?(&1, "#"))
7
8   @contract rgb_to_hex(r :: rgb(), g :: rgb(), b :: rgb()) :: hex()
9   def rgb_to_hex(r, g, b) do
10     # ...
11   end
12 end
```

---

## Validation and conforming values

Norm validates data by “conforming” the value to a specification. If the values don’t conform then a list of errors is returned. There are 2 functions provided for this `conform/2` and `conform!/2`. If you need to return a list of well defined errors then you should use `conform/2`. Otherwise `conform!/2` is generally more useful. The input data is always passed as the 1st argument to `conform` so that calls to conform are easily chainable.

## Predicates and specs

Norm does not provide a special set of predicates and instead allows you to convert any predicate into a spec with the `spec/1` macro. Predicates can be composed together using the `and` and `or` keywords. You can also use anonymous functions to create specs.

```
1 spec(is_binary())
2 spec(is_integer() and &(&1 > 0))
3 spec(is_binary() and fn str -> String.length(str) > 0 end)
```

The data is always passed as the first argument to your predicate so you can use predicates with multiple values like so:

```
1 iex> defmodule Predicate do
2 ...>   def greater?(x, y), do: x > y
3 ...> end
4 iex> conform!(10, spec(Predicate.greater?(5)))
5 10
6 iex> conform!(3, spec(Predicate.greater?(5)))
7 ** (Norm.MismatchError) Could not conform input:
8 val: 3 fails: Predicate.greater?(5)
```

## Tuples and atoms

Atoms and tuples can be matched without needing to wrap them in a function.

```
1 iex> :atom = conform!(:atom, :atom)
2 :atom
3 iex> {1, "hello"} = conform!({1, "hello"}, {spec(is_integer()), spec(
4   is_binary())})
5 {1, "hello"}
6 iex> conform!({1, 2}, {:one, :two})
7 ** (Norm.MismatchError) Could not conform input:
8 val: 1 in: 0 fails: is not an atom.
9 val: 2 in: 1 fails: is not an atom.
```

---

Because Norm supports matching on bare tuples we can easily validate functions that return `{:ok, term()}` and `{:error, term()}` tuples. These specifications can be combined with `one_of/1` to create union types.

```
1 iex> defmodule User do
2 ...>   defstruct [:name, :age]
3 ...>
4 ...>   def get_name(id) do
5 ...>     case id do
6 ...>       1 -> {:ok, "Chris"}
7 ...>       2 -> {:ok, "Alice"}
8 ...>       _ -> {:error, "user does not exist"}
9 ...>     end
10 ...>   end
11 ...> end
12 iex> result_spec = one_of([
13 ...>   {:ok, spec(is_binary())},
14 ...>   {:error, spec(fn _ -> true end)},
15 ...> ])
16 iex> {:ok, _name} = conform!(User.get_name(1), result_spec)
17 {:ok, "Chris"}
18 iex> {:ok, "Alice"} = conform!(User.get_name(2), result_spec)
19 {:ok, "Alice"}
20 iex> {:error, _error} = conform!(User.get_name(-42), result_spec)
21 {:error, "user does not exist"}
```

## Collections

Norm can define collections of values using `coll_of`.

```
1 iex> conform!([1,2,3], coll_of(spec(is_integer)))
2 [1, 2, 3]
```

Collections can take a number of options:

- `:kind` - predicate function the kind of collection being conformed
- `:distinct` - boolean value for specifying if the collection should have distinct elements
- `:min_count` - Minimum element count
- `:max_count` - Maximum element count
- `:into` - The output collection the input will be conformed into. If not specified then the input type will be used.

```
1 iex> conform!([:a, :b, :c], coll_of(spec(is_atom), into: MapSet.new()))
2 #MapSet<[:a, :b, :c]>
```

---

## Schemas

Norm provides a `schema/1` function for specifying maps and structs:

```
1 iex> user_schema = schema(%{
2 ...>   user: schema(%{
3 ...>     name: spec(is_binary()),
4 ...>     age: spec(is_integer() and & &1 > 0),
5 ...>   })
6 ...> })
7 iex> conform!(%{user: %{name: "chris", age: 31}}, user_schema)
8 %{user: %{name: "chris", age: 31}}
9 iex> conform!(%{user: %{name: "chris", age: -31}}, user_schema)
10 ** (Norm.MismatchError) Could not conform input:
11 val: -31 in: :user/:age fails: &(&1 > 0)
```

Schema's are designed to allow systems to grow over time. They provide this functionality in two ways. The first is that any unspecified fields in the input are passed through when conforming the input. The second is that all keys in a schema are optional. This means that all of these are valid:

```
1 iex> user_schema = schema(%{
2 ...>   name: spec(is_binary()),
3 ...>   age: spec(is_integer()),
4 ...> })
5 iex> conform!(%{}, user_schema)
6 %{}
7 iex> conform!(%{age: 31}, user_schema)
8 %{age: 31}
9 iex> conform!(%{foo: :foo, bar: :bar}, user_schema)
10 %{foo: :foo, bar: :bar}
```

If you're used to more restrictive systems for managing data these might seem like odd choices. We'll see how to specify required keys when we discuss Selections.

**Structs** You can also create specs from structs:

```
1 defmodule User do
2   defstruct [:name, :age]
3
4   def s, do: schema(%__MODULE__ {
5     name: spec(is_binary()),
6     age: spec(is_integer())
7   })
8 end
```

This will ensure that the input is a `User` struct with the key that match the given specification. Its convention to provide a `s()` function in the module that defines the struct so that schema's can be shared throughout your system.

---

You don't need to provide specs for all the keys in your struct. Only the spec'ed keys will be conformed. The remaining keys will be checked for presence.

```
1 defmodule Norm.User do
2   defstruct [:name, :age]
3 end
4 iex> user_schema = schema(%Norm.User{})
5 iex> conform!(%Norm.User{name: "chris"}, user_schema)
```

**Key semantics** Atom and string keys are matched explicitly and there is no casting that occurs when conforming values. If you need to match on string keys you should specify your schema with string keys.

Schemas accommodate growth by disregarding any unspecified keys in the input map. This allows callers to start sending new data over time without coordination with the consuming function.

### Selections and optionality

We said that all of the fields in a schema are optional. In order to specify the keys that are required in a specific use case we can use a Selection. The Selection takes a schema and a list of keys - or keys to lists of keys - that must be present in the schema.

```
1 iex> user_schema = schema(%{
2 ...>   user: schema(%{
3 ...>     name: spec(is_binary()),
4 ...>     age: spec(is_integer()),
5 ...>   })
6 ...> })
7 iex> just_age = selection(user_schema, [user: [:age]])
8 iex> conform!(%{user: %{name: "chris", age: 31}}, just_age)
9 %{user: %{age: 31, name: "chris"}}
10 iex> conform!(%{user: %{name: "chris"}}, just_age)
11 ** (Norm.MismatchError) Could not conform input:
12 val: %{name: "chris"} in: :user/:age fails: :required
```

If you need to mark all fields in a schema as required you can elide the list of keys like so:

```
1 iex> user_schema = schema(%{
2 ...>   user: schema(%{
3 ...>     name: spec(is_binary()),
4 ...>     age: spec(is_integer()),
5 ...>   })
6 ...> })
7 iex> conform!(%{user: %{name: "chris", age: 31}}, selection(user_schema
8   ))
9 %{user: %{name: "chris", age: 31}}
```

---

Selections are an important tool because they give control over optionality back to the call site. This allows callers to determine what they actually need and makes schema's much more reusable.

## Patterns

Norm provides a way to specify alternative specs using the `alt/1` function. This is useful when you need to support multiple schema's or multiple alternative specs.

```
1 iex> create_event = schema(%{type: spec(&(&1 == :create))})
2 iex> update_event = schema(%{type: spec(&(&1 == :update))})
3 iex> event = alt(create: create_event, update: update_event)
4 iex> conform!(%{type: :create}, event)
5 {:create, %{type: :create}}
6 iex> conform!(%{type: :update}, event)
7 {:update, %{type: :update}}
8 iex> conform!(%{type: :delete}, event)
9 ** (Norm.MismatchError) Could not conform input:
10 val: :delete in: :create/:type fails: &(&1 == :create)
11 val: :delete in: :update/:type fails: &(&1 == :update)
```

## Generators

Along with validating that data conforms to a given specification, Norm can also use specifications to generate examples of good data. These examples can then be used for property based testing, local development, seeding databases, or any other use case.

```
1 iex> user_schema = schema(%{
2 ...>   name: spec(is_binary()),
3 ...>   age: spec(is_integer() and &(&1 > 0))
4 ...> })
5 iex> generated =
6 ...>   user_schema
7 ...>   |> gen()
8 ...>   |> Enum.take(3)
9 iex> for user <- generated, do: user.age > 0 && is_binary(user.name)
10 [true, true, true]
```

Under the hood Norm uses StreamData for its data generation. This means you can use your specs in tests like so:

```
1 input_data = schema(%{"user" => schema(%{"name" => spec(is_binary())})})
2
3 property "users can update names" do
4   check all input <- gen(input_data) do
5     assert :ok == update_user(input)
```

---

```
6   end
7   end
```

## Built in generators

Norm will try to infer the generator to use from the predicate defined in `spec`. It looks specifically for the guard clauses used for primitive types in elixir. Not all of the built in guard clauses are supported yet. PRs are very welcome ;).

## Guiding generators

You may have specs like `spec(fn x -> rem(x, 2) == 0 end)` which check to see that an integer is even or not. This generator expects integer values but there's no way for Norm to determine this. If you try to create a generator from this spec you'll get an error:

```
1 gen(spec(fn x -> rem(x, 2) == 0 end))
2 ** (Norm.GeneratorError) Unable to create a generator for: fn x -> rem(
   x, 2) == 0 end
3 (norm) lib/norm.ex:76: Norm.gen/1
```

You can guide Norm to the right generator by specifying a guard clause as the first predicate in a spec. If Norm can find the right generator then it will use any other predicates as filters in the generator.

```
1 Enum.take(gen(spec(is_integer() and fn x -> rem(x, 2) == 0 end)), 5)
2 [0, -2, 2, 0, 4]
```

But its also possible to create filters that are too specific such as this:

```
1 gen(spec(is_binary() and &(&1 =~ ~r/foobarbaz/)))
```

Norm can determine the generators to use however its incredibly unlikely that Norm will be able to generate data that matches the filter. After 25 consecutive unseccessful attempts to generate a good value Norm (StreamData under the hood) will return an error. In these scenarios we can create a custom generator.

## Overriding generators

You'll often need to guide your generators into the interesting parts of the state space so that you can easily find bugs. That means you'll want to tweak and control your generators. Norm provides an escape hatch for creating your own generators with the `with_gen/2` function:

---

```
1 age = spec(is_integer() and &(&1 >= 0))
2 reasonable_ages = with_gen(age, StreamData.integer(0..105))
```

Because `gen/1` returns a `StreamData` generator you can compose your generators with other `StreamData` functions:

```
1 age = spec(is_integer() and &(&1 >= 0))
2 StreamData.frequency([
3   {3, gen(age)},
4   {1, StreamData.binary()}],
5 ])
6
7 gen(age) |> StreamData.map(&Integer.to_string/1) |> Enum.take(5)
8 ["1", "1", "3", "4", "1"]
```

This allows you to compose generators however you need to while keeping your generation co-located with the specification of the data.

## Adding contracts to functions

You can `conform` data wherever it makes sense to do so in your application. But one of the most common ways to use Norm is to validate a functions arguments and return value. Because this is such a common pattern, Norm provides function annotations similar to `@spec`:

```
1 defmodule Colors do
2   use Norm
3
4   def rgb(), do: spec(is_integer() and &(&1 in 0..255))
5
6   def hex(), do: spec(is_binary() and &String.starts_with?(&1, "#"))
7
8   @contract rgb_to_hex(r :: rgb(), g :: rgb(), b :: rgb()) :: hex()
9   @doc "Convert an RGB value to its CSS-style hexadecimal notation."
10  def rgb_to_hex(r, g, b) do
11    # ...
12  end
13 end
```

If the arguments for `rgb_to_hex` don't conform to the specification or if `rgb_to_hex` does not return a value that conforms to `hex` then an error will be raised.

Note `@contract` must be placed *before* `@doc` as above for `ExDoc` and `ExUnit.DocTest` to continue working as intended.

---

## Installation

Add `norm` to your list of dependencies in `mix.exs`. If you'd like to use Norm's generator capabilities then you'll also need to include `StreamData` as a dependency.

```
1 def deps do
2   [
3     {:stream_data, "~> 0.4"},
4     {:norm, "~> 0.13"}
5   ]
6 end
```

## Should I use this?

Norm is still early in its life so there may be some rough edges. But we're actively using this at my current company (Bleacher Report) and working to make improvements.

## Contributing and TODOS

Norm is being actively worked on. Any contributions are very welcome. Here is a limited set of ideas that are coming soon.

- ☐ More streamlined specification of keyword lists.
- ☐ Support "sets" of literal values
- ☐ specs for functions and anonymous functions
- ☐ easier way to do dispatch based on schema keys