QuviQ

Testing Web Services with QuickCheck

Thomas Arts

Why is testing hard?

O(n) test cases

3—4 tests per feature

Why is testing hard?

$O(n^2)$ test cases

pairs of features

Why is testing hard?

$O(n^3)$ test cases

triples of features

Don’t write tests!

Generate them

Specify properties

Specify a property of your software:

QuickCheck automatically generates random test cases to verify the property.

```
defmodule StringTests do
   use ExUnit.Case
   use EQC.ExUnit

   property "Strings starting" do
     forall {s1, s2} <- {utf8(), utf8()} do
       String.starts_with?(s1 <> s2, s1)
     end
   end

end
```

Running QuickCheck

Add :eqc_ex to your project

```
> mix eqc.install --mini
> mix eqc test/string_eqc.ex
.....................................................
.....................................
OK, passed 100 tests
.

Finished in 1.0 seconds
1 property, 0 failures
```

Properties

```
property "reverse strings" do
  forall s <- Type.string() do
    ensure String.reverse(String.reverse(s)) == s
  end
end
```

QuickCheck

```
.....................................................
....Failed! After 60 tests.
not ensured: "扜🔥W🪐)ӟ" == "🔥扜W🪐)ӟ"
Shrinking xx.xxxxxxxxxxxxxx..xxxxxxxx.....xxxxxxxxxxxx
xxxxxxxx(8 times)
not ensured: " 🔥" == "🔥 "
```

```
  1) property reverse strings (SimpleTests)
     test/simple_eqc.exs:10
     forall(s <- Type.string()) do
       ensure(String.reverse(String.reverse(s)) == s)
     end
     Failed for "🔥 "

Finished in 1.0 seconds
1 property, 1 failure
```

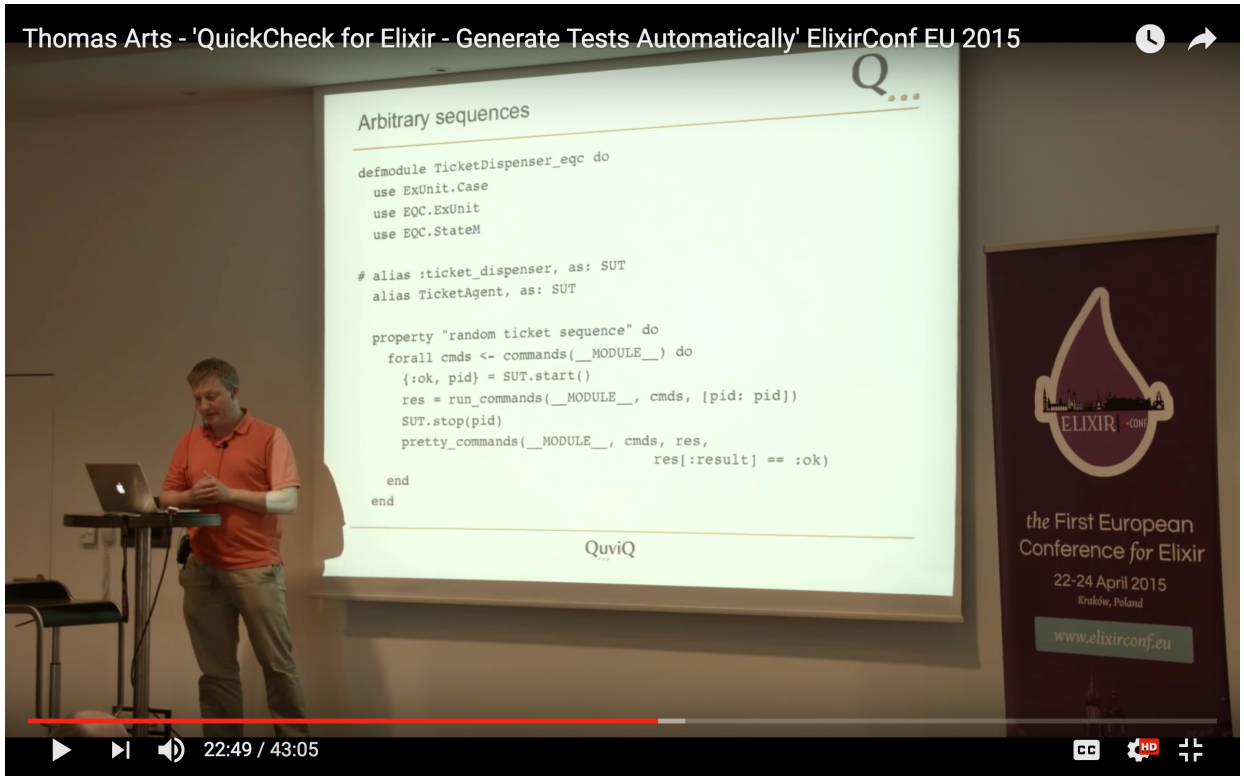
Property Based Testing

Use specifications

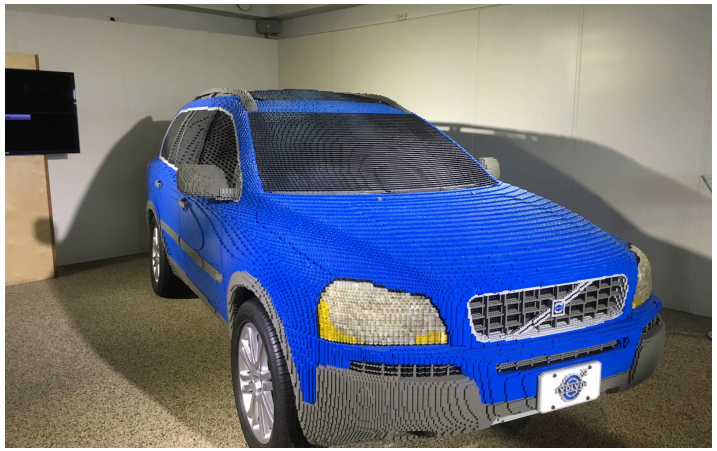
(properties)

Generate tests from specifications

What about real systems?



Quviq



3,000 pages of specifications

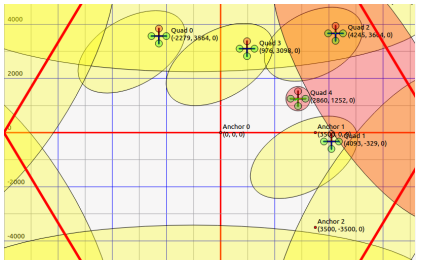
20,000 lines of QuickCheck

1,000,000 LOC, 6 suppliers

200 problems

100 problems in the standard

10x shorter test code



Dropbox File Store

Riak

basho

klarna

ejabberd

JSON schema

use JSON schema to define data generators:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "currency": {
            "type": "string",
            "enum": [ "EUR", "USD", "SEK" ]
        },
        "amount": {
            "type": "number",
            "minimum": 0
        }
    },
    "required": [ "currency", "amount" ]
}
```

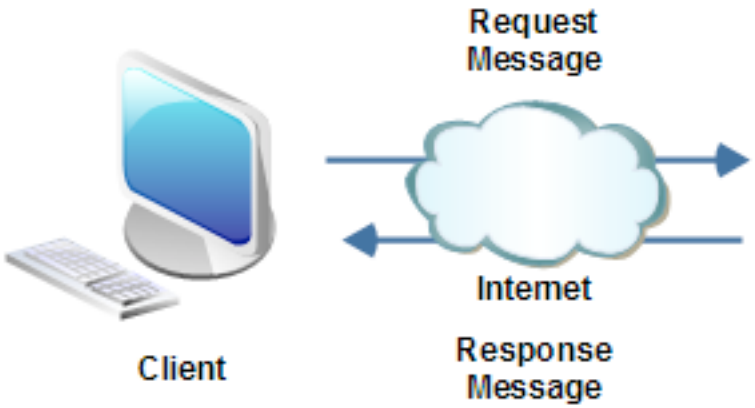
JSON schema generators

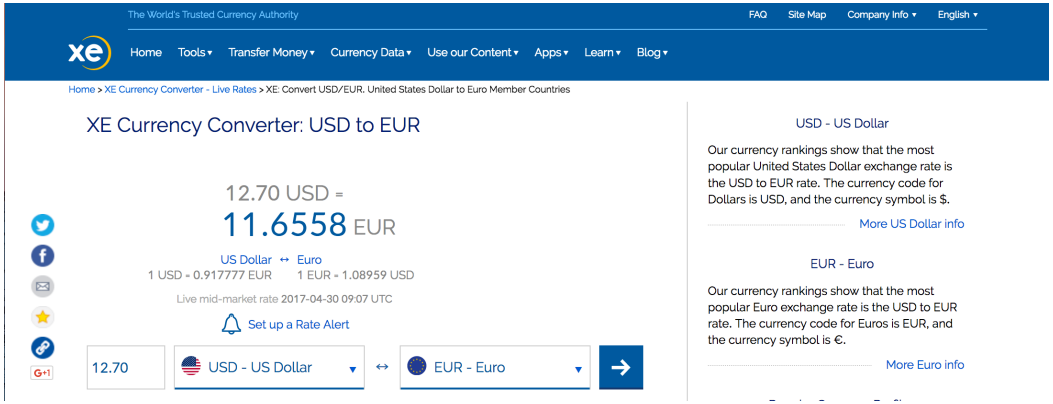
```
setup do
  module = SchemaGen.from_file("valuta.json")
  [ generator: module.generate() ]
end


property "Currency positive", %{generator: currency} do
  forall %{ "amount" => v1 } <- currency do
    ensure v1 >= 0
  end
end
```

```
....................
....................
OK, passed 100 tests
.

Finished in 1.5 seconds
1 property, 0 failures
```

```
"{\"currency\":\"USD\",\"amount\":854}"
"{\"currency\":\"USD\",\"amount\":75529}"
"{\"currency\":\"USD\",\"amount\":48}"
"{\"currency\":\"USD\",\"amount\":14917.0}"
"{\"currency\":\"SEK\",\"amount\":942}"
"{\"currency\":\"SEK\",\"amount\":649.0}"
"{\"currency\":\"SEK\",\"amount\":19.857142857142}"
"{\"currency\":\"EUR\",\"amount\":71.0}"
"{\"currency\":\"EUR\",\"amount\":159}"
"{\"currency\":\"EUR\",\"amount\":1.0}"
.....
```

Test a currency web service!





Web Service

JSON API

JSON schema for return value

```
{
      "$schema": "http://json-schema.org/draft-04/schema#",
      "type": "object",
      "properties": {
          "from": { "type": "string",
                    "enum": [ "EUR", "USD", "SEK" ]
          },
          "to": { "type": "string",
                  "enum": [ "EUR", "USD"       ]
          },
          "amount": { "type": "number",
                      "minimum": 0
          },
          "time": { "type": "string" }
      },
      "required": [ "from", "to", "amount" ]
}
```

introduce an error!

Type correctness

```
setup_all do
  module = SchemaGen.from_file("valuta.json")
  resp   = SchemaGen.from_file("currency_api.json")
  [ generator: module.generate(),
    valid: fn(x) -> resp.validate(x) end ]
end

property "exchange", %{generator: currency, valid: validate} do
  forall %{ "currency" => cur1, "amount" => v1 } <- currency do
    forall %{ "currency" => cur2 } <- currency do

      return = convert(cur1, cur2, v1)
      ensure validate.(return) == :ok
    end
  end
end
```

performs the web service call

Found the type error!

```
Request get http://www.xe.com/currencyconverter/convert
  query:  %{"Amount" => 1.0, "From" => "USD", "To" => "EUR"}
...

Request get http://www.xe.com/currencyconverter/convert
query:  %{"Amount" => 0.5, "From" => "SEK", "To" => "SEK"}
Failed! After 11 tests.
not ensured: {:error, [{"Value \"SEK\" is not allowed in enum.", "#/to"}]} == :ok


Shrinking .
Request get http://www.xe.com/currencyconverter/convert
  query:  %{"Amount" => 0.0, "From" => "EUR", "To" => "SEK"}

not ensured: {:error, [{"Value \"SEK\" is not allowed in enum.", "#/to"}]} == :ok


  1) property exchange (TypeTests)
     test/type_eqc.exs:23
     forall(%{"currency" => cur1, "amount" => v1} <- currency) do
       forall(%{"currency" => cur2} <- currency) do
         return = convert(cur1, cur2, v1)
         ensure(validate.(return) == :ok)
       end
     end
     Failed for %{"amount" => 0.0, "currency" => "EUR"}
                %{"amount" => 0.0, "currency" => "SEK"}
```

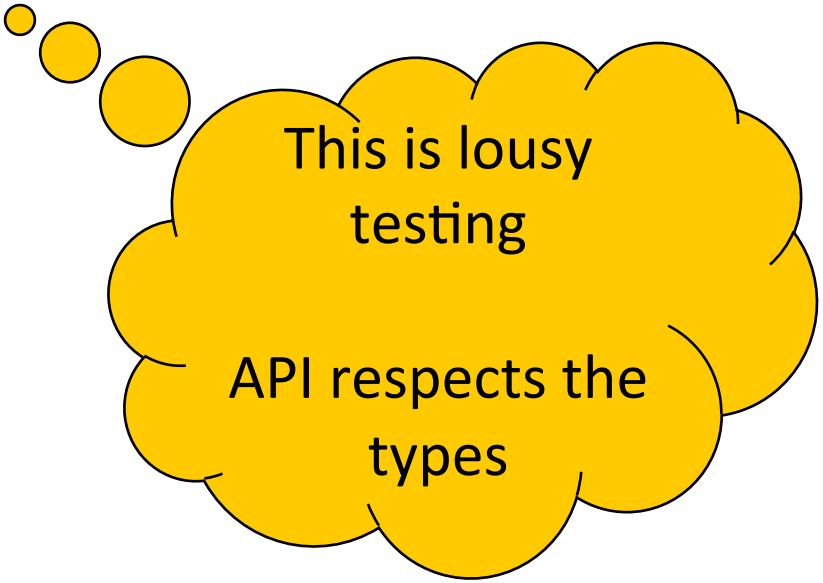
JSON schema for return value

```
{
      "$schema": "http://json-schema.org/draft-04/schema#",
      "type": "object",
      "properties": {
          "from": { "type": "string",
                    "enum": [ "EUR", "USD", "SEK" ]
          },
          "to": { "type": "string",
                  "enum": [ "EUR", "USD", "SEK" ]
          },
          "amount": { "type": "number",
                      "minimum": 0
          },
          "time": { "type": "string" }
      },
      "required": [ "from", "to", "amount" ]
}
```

fix the specification!

Type specification tested

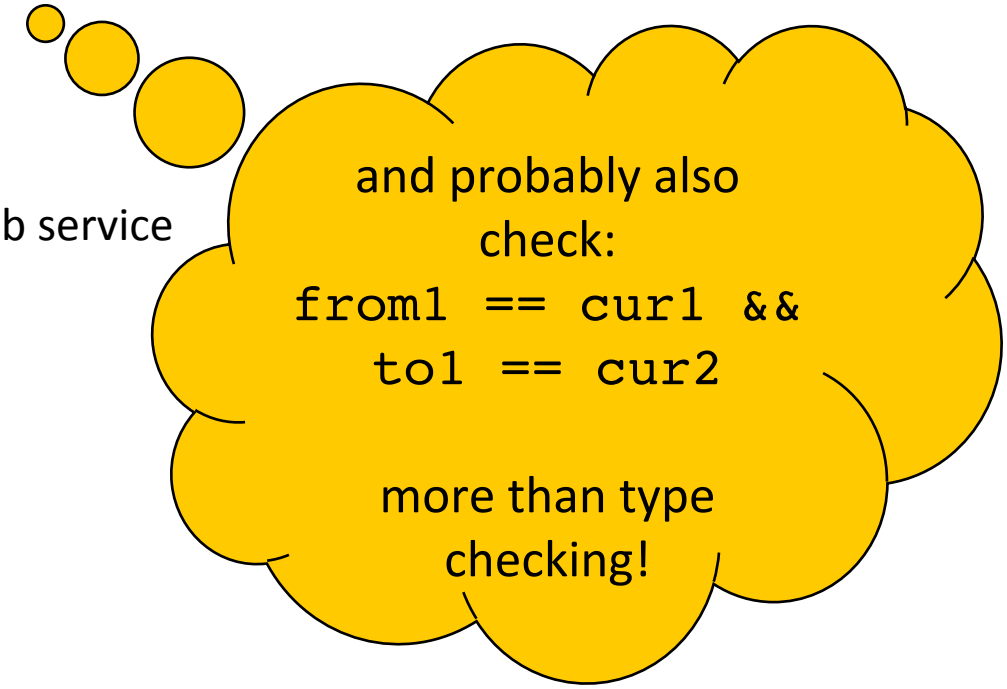
```
.....................................................................
.........................

OK, passed 100 tests

.


Finished in 46.1 seconds

1 property, 0 failures
```



Test a currency web service!

```
property "exchange", %{generator: currency} do
  forall %{ "currency" => cur1, "amount" => v1 } <- currency do
    forall %{ "currency" => cur2 } <- currency do
       %{ "amount" => v2} = convert(cur1, cur2, v1)
       %{ "amount" => v}  = convert(cur2, cur1, v2)
       ensure v == v1
    end
  end
end
```

The function `convert` queries the web service



Zero is not allowed?

```
Request get http://www.xe.com/currencyconverter/convert
  query:  %{"Amount" => 0.0, "From" => "USD", "To" => "EUR"}
Request get http://www.xe.com/currencyconverter/convert
  query:  %{"Amount" => 0.918685, "From" => "EUR", "To" =>
"USD"}
Failed! After 1 tests.
not ensured: 1.0 == 0.0

Shrinking .(1 times)


Request get http://www.xe.com/currencyconverter/convert
  query:  %{"Amount" => 0.0, "From" => "EUR", "To" => "EUR"}
Request get http://www.xe.com/currencyconverter/convert
  query:  %{"Amount" => 1.0, "From" => "EUR", "To" => "EUR"}
not ensured: 1.0 == 0.0
```

Zero is not allowed!

Only allow values > 0

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "currency": {
            "type": "string",
            "enum": [ "EUR", "USD", "SEK" ]
        },
        "amount": {
            "type": "number",
            "minimum": 0,
            "exclusiveMinimum": true
        }
    },
   "required": [ "currency", "amount" ]
}
```

Zero is not allowed?

```
....

not ensured: 1.0 == 1.0e-9


1) property exchange (ValutaTests)
   test/valuta_eqc.exs:60
   forall
      ...
   end
   Failed for %{"amount" => 1.0e-9, "currency" => "USD"},
              %{"amount" => 1.0e-9, "currency" => "EUR"}
```

Precision seems 10^{-9} but need different currencies

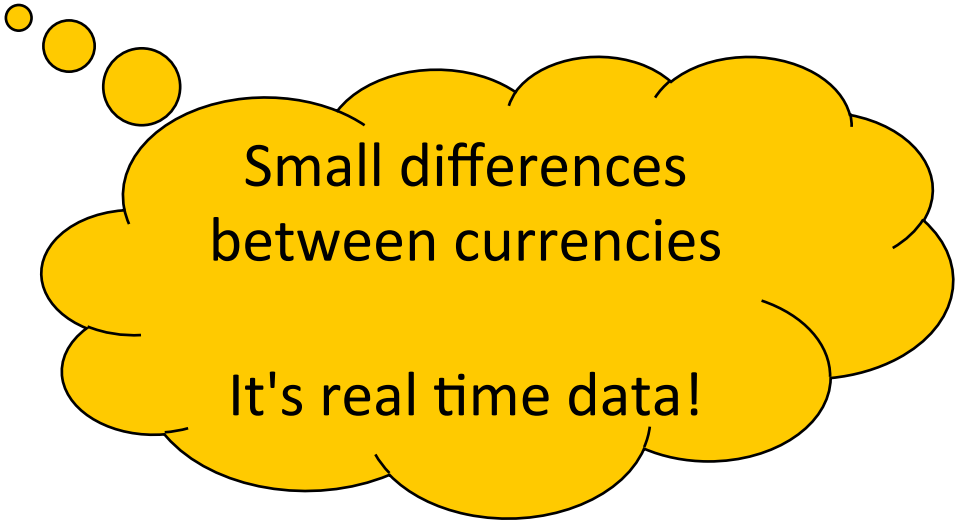
Only allow values > 0

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "currency": {
            "type": "string",
            "enum": [ "EUR", "USD", "SEK" ]
        },
        "amount": {
            "type": "number",
            "minimum": 1
        }
    },
    "required": [ "currency", "amount" ]
}
```

Ok, more than 1

Precision or real time data?

```
Request get http://www.xe.com/currencyconverter/convert
  query:  %{"Amount" => 1.0, "From" => "USD", "To" => "EUR"}
Request get http://www.xe.com/currencyconverter/convert
  query:  %{"Amount" => 0.917897, "From" => "EUR", "To" =>
"USD"}
Failed! After 2 tests.
not ensured: 1.00006 == 1.0
```



Run the same check again

```
@check usd1: [ %{"amount" => 1.0, "currency" => "USD"},
               %{"currency" => "EUR" }]

property "exchange", %{generator: currency} do
  forall %{ "currency" => cur1, "amount" => v1 } <- currency do
    forall %{ "currency" => cur2 } <- currency do
       %{ "amount" => v2} = convert(cur1, cur2, v1)
       %{ "amount" => v}  = convert(cur2, cur1, v2)
       ensure v == v1
    end
  end
end
```

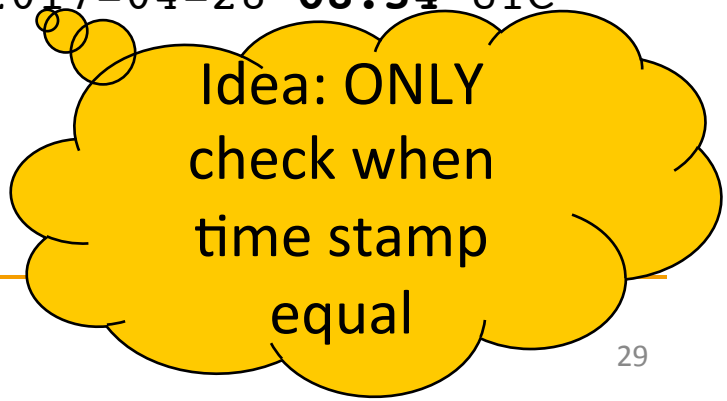

```
mix eqc --only check
```

- OK, passed the test.
- not ensured: 1.00013 == 1.0

Show the time

```
property "exchange", %{generator: currency} do
  forall %{ "currency" => cur1, "amount" => v1 } <- currency do
    forall %{ "currency" => cur2 } <- currency do
      %{ "amount" => v2, "time" => t1} = convert(cur1, cur2, v1)
      %{ "amount" => v,  "time" => t2} = convert(cur2, cur1, v2)
      when_fail IO.puts "at time #{t1} and #{t2}" do
        ensure v == v1
      end
    end
  end
end
```

```
Failed!
at time 2017-04-28 08:53 UTC and 2017-04-28 08:54 UTC
not ensured: 0.999975 == 1.0
```



Skip if time not the same

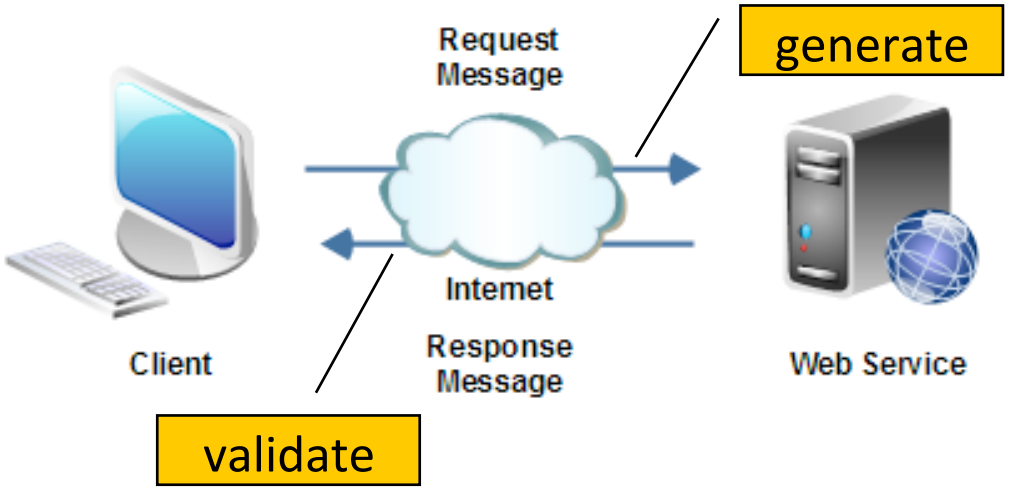
```
property "exchange", %{generator: currency} do
  forall %{ "currency" => cur1, "amount" => v1 } <- currency do
    forall %{ "currency" => cur2 } <- currency do
      %{"amount" => v2, "time" => t1} = convert(cur1, cur2, v1)
      %{"amount" => v,  "time" => t2} = convert(cur2, cur1, v2)
      when_fail IO.puts "at time #{t1} and #{t2}" do
        implies t1 == t2 do
          ensure v == v1
        end
      end
    end
  end
end
```

Show the time

```
at time 2017-04-28 09:10 UTC and 2017-04-28 09:10 UTC
not ensured: 0.999999 == 1.0

at time 2017-04-28 09:13 UTC and 2017-04-28 09:13 UTC
not ensured: 1.3333333333333 == 1.3333333333333333

at time 2017-04-28 09:33 UTC and 2017-04-28 09:33 UTC
not ensured: 0.999996 == 1.0
```

Web service testing

- Specify data in JSON Schema
- Generate tests for arbitrary (shrinking) data
- Use Elixir / QuickCheck to execute tests



Web service testing

But... arbitrary data can be too "arbitrary"

After JSON schema, there quickly is demand for more precise specifications.

QuickCheck = Domain Specific Language
for defining random data

Currency at a given time

Add a timestamp to the query

"What is the exchange rate at tim

```
"type": "string",
"pattern": "(19|20)[0-9][0-9]-(0
(0|1)[0-9]:[0-5][0-9]"
```

```
"2071-10-07 00:49"
"2026-11-33 16:16"
"2024-04-22 16:06"
"2021-11-16 14:02"
"2007-02-20 19:44"
"1940-04-06 09:18"
"1939-12-22 15:58"
"1932-08-19 12:17"
"1917-07-32 07:38"
"1912-01-07 13:52"
```

SEK to USD in 1980 makes sense

SEK to EUR in 1980 makes NO sense

Dependencies hard to encode in JSON schema

Combine JSON and QuickCheck

```
property "currency at time T", %{generator: currency} do
  forall cur <- currency do
    forall t <- timestamp(cur) do

          ....

    end
  end
end
```

Combine JSON and QuickCheck

```
def date("EUR") do
  Type.Date.generate([start_date: ~D[1999-01-01],
                      end_date: ~D[2017-05-04] ])
end
def date(_) do
  Type.Date.generate([start_date: ~D[1980-01-01],
                      end_date: ~D[2017-05-04] ])
end

property "currency at time T", %{generator: currency} do
  forall cur <- currency do
    forall {d, t} <- {date(cur["currency"]), Type.Time.generate()} do
      {hm, _} = t |> Time.to_string |> String.split_at(5)
      timestamp = Date.to_string(d) <> " " <> hm

      .....
    end
  end
end
```

Conclusion

JSON schema, QuickCheck and Elixir powerful trio

Elixir good way in for testing web services

people like it!