elixir

@elixirlang / elixir-lang.org

- **Productivity**

- Productivity
- Extensibility

- Productivity
- Extensibility
- Compatibility

{ 1, 2, 3 } - tuples

```
{ 1, 2, 3 } - tuples
[ 1, 2, 3 ] - lists
```

```
{ 1, 2, 3 } - tuples
[ 1, 2, 3 ] - lists
"hello"     - binary
```

```
{ 1, 2, 3 } - tuples
[ 1, 2, 3 ] - lists
"hello"     - binary
:atom       - atoms
```

```
{ 1, 2, 3 } - tuples
[ 1, 2, 3 ] - lists
"hello"     - binary
:atom       - atoms
Module      - modules
```

elixir

```elixir
defmodule Hello do
  def world do
    IO.puts "hello world"
  end
end

Hello.world
```

Productivity

Everything is an expression

erlang

```
-module(hello).

world() ->
   io:format("hello world~n").
```

erlang

```
-module(hello).

io:format("compiling~n").

world() ->
  io:format("hello world~n").
```

elixir

```
defmodule Hello do
  IO.puts "compiling"

  def world() do
    IO.puts "hello world"
  end
end
```

```
$ elixirc hello.ex
compiling
Compiled hello.ex

$ elixir -e Hello.world
hello world
```

Compile time work

elixir

```
def binary_to_integer(b) do
  b |> binary_to_list
    |> list_to_integer
end
```

elixir

```elixir
defmodule MyLib do
  bif = function_exported?(
    :erlang,
    :binary_to_integer,
    1
  )
  if bif do
    # delegate to erlang
  else
    # stub
  end
end
```

Modules on console

iex

```
iex> defmodule Hello do
...> def world() do
...> IO.puts "hello world"
...> end
...> 1+2
...> end
```

```
{
   :module,
   Hello,
   <<70,79,82,49,...>>,
   3
}
```

Scripts

```
$ cat hello.exs
IO.puts "Hello world"

$ elixir hello.exs
Hello world
```

Macros

“Lisps traditionally
empowered developers
because you can
eliminate anything that's tedious through macros,
and that power is really what
people keep going back for”

- Rich Hickey

lisp

```
(+ 1 2)
```

elixir

elixir

```
1 + 2
```

elixir

```
1 + 2
```

```
{ :+, [], [1,2] }
```

function metadata args

elixir

elixir

```
is_atom(:foo)
```

elixir

```
is_atom(:foo)

{ :is_atom, [], [:foo] }
```

function

metadata

args

```
defmacro unless(expr, opts) do
  quote do
    if(!unquote(expr), unquote(opts))
  end
end

unless(true, do: exit())
```

Domain Specific Languages

erlang

```
handle(
  'GET',
  [<<"posts">>, ID],
  Req
) ->
```

elixir

```
def handle(
  :GET,
  ["posts", id],
  req
) do
```

elixir

```
get "posts/:id", req do
```

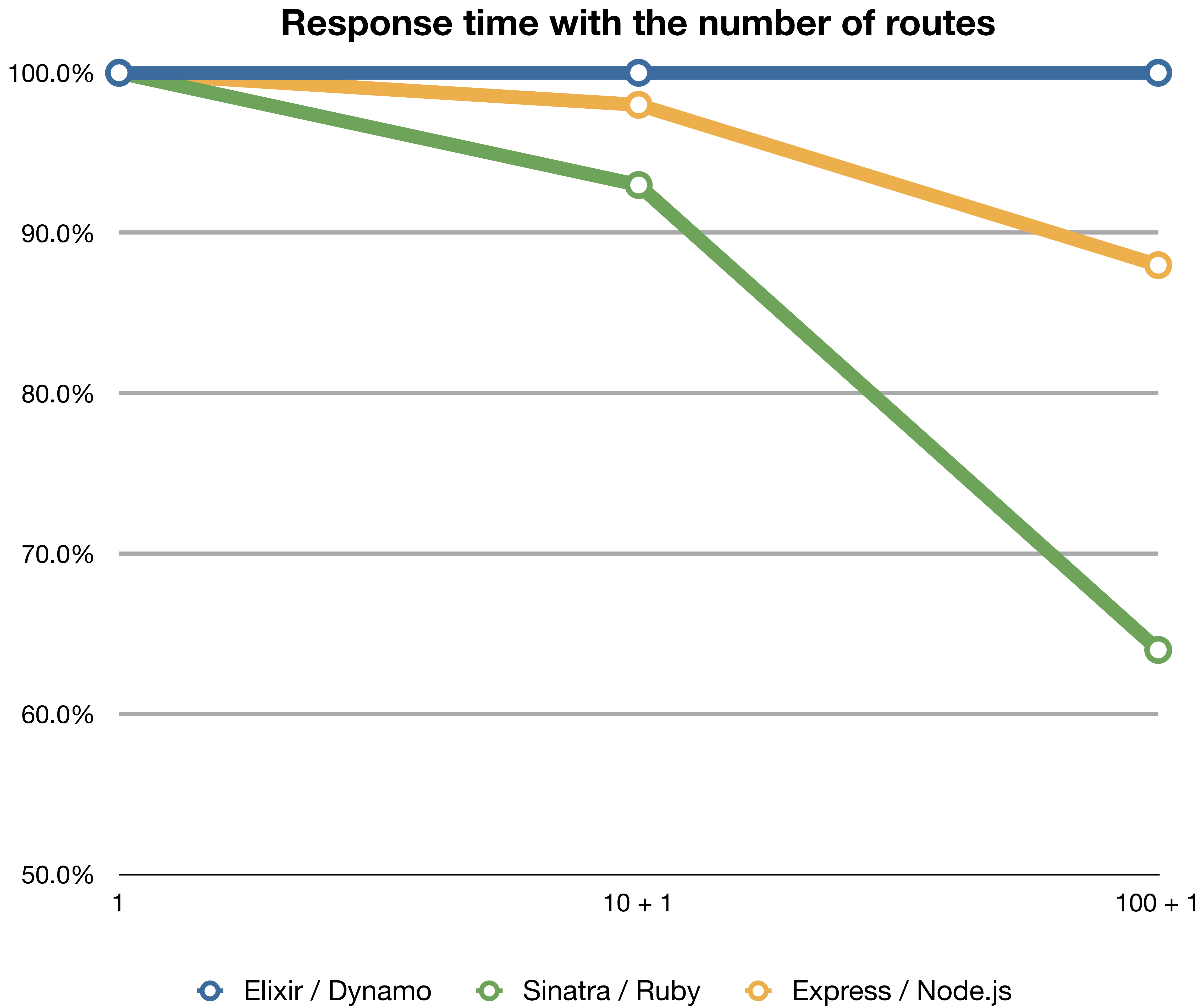
elixir

```
get "posts", req do

get "posts/:id", req do
```

↓

```
def handle(verb, path, req) do
```


Response time with the number of routes
100.0%
90.0%
80.0%
70.0%
60.0%
50.0%
1
10 + 1
100 + 1
Elixir / Dynamo
Sinatra / Ruby
Express / Node.js

elixir

```elixir
defmodule MathTest do
  use ExUnit.Case

  test "basic operations" do
    assert 1 + 1 == 2
  end
end
```

elixir

```
defmodule MathTest do
  use ExUnit.Case

  def test_basic_operations do
    assert 1 + 1 == 2
    :ok
  end
end
```

```
# assert 1 + 1 == 2

defmacro assert({ :=~, line, [l,r] }) do
  # ...
end

defmacro assert({ :==, line, [l,r] }) do
  # ...
end

defmacro assert(default) do
  # ...
end
```

beam.smp bash

```
/Volumes/Work/github/elixir[master+]$ make test_eex
==> eex (exunit)
................F.F..............................

Failures:

  1) test evaluates with assigns (EEx.SmartEngineTest)
     ** (ExUnit.ExpectationError)
                  expected: "1"
       to be equal to (==): "2"
     at test/eex/smart_engine_test.exs:11

  2) test simple chars lists (EEx.TokenizerTest)
     ** (ExUnit.ExpectationError)
                  expected: [{:text,2,"foo"}]
       to be equal to (==): [{:text,1,"foo"}]
     at test/eex/tokenizer_test.exs:8

Finished in 0.07 seconds
48 tests, 2 failures
make: *** [test_eex] Error 1
/Volumes/Work/github/elixir[master+*]$
```

Extensibility

The expression problem

	add	delete	count
List			
Array			
Set			

OO

	add	delete	count
List			
Array			
Set			
MyList	Your class implementation		

FP

	add	delete	count	reduce
List				Your new function
Array				
Set				

erlang

```erlang
-module(json).

to_json(Item) when is_list(Item) ->
  % ...
to_json(Item) when is_binary(Item) ->
  % ...
to_json(Item) when is_number(Item) ->
  % ...
```

Protocols

elixir

```
defprotocol JSON do
  def to_json(item)
end

JSON.to_json(item)
```

elixir

```
defimpl JSON, for: List do
  # ...
end

defimpl JSON, for: Binary do
  # ...
end

defimpl JSON, for: Number do
  # ...
end
```

elixir

```elixir
defimpl JSON, for: Set do
  # ...
end
```

Enum API

elixir

```
Enum.map [1,2,3], fn(x) ->
  x * 2
end
#=> [2,4,6]
```

elixir

```
Enum.map 1..5, fn(x) ->
  x * 2
end
#=> [2,4,6,8,10]
```

Inspect API

erlang

```
1> dict:from_list([{a,1}]).
{dict,1,16,16,8,80,48,
      {[],[],[],[],[],[],[],[],[],[],
       [],[],[],[],[],[]},
      {{[],
        [[a|1]],
        [],[],[],[],[],[],[],[],[],[],
        [],[],[],[]}}}
```

elixir

```
iex> HashDict.new(a: 1)
#HashDict<[a: 1]>
```

Compatibility

ERLANG

DISTRIBUTED
FAULT-TOLERANT
APPLICATIONS
WITH HOT-CODE
SWAPPING

GENOMU

A CONCURRENCY-ORIENTED K/V DATABASE

Why?

There is no conversion cost for calling Erlang from Elixir and vice-versa

Elixir's standard library is meant to be compact enough to show case Elixir's features

:application
:gen_server
:gen_event
:supervisor

:lists -> Enum
:string -> String
:dict / :orddict -> HashDict

String (unicode)

- Basic primitives (codepoints, graphemes)
- Basic operations (length, at, split)
- Compilation-time work:
  200LOC / 200kb BEAM file

String (unicode)

```
def downcase("É" <> t) do
  "é" <> t
end

def downcase("Á" <> t) do
  "á" <> t
end
```

String (unicode)

- Compilation-time work:
  200LOC / 200kb BEAM file
- Reasonably fast, no bottlenecks

Scripting

- No special mode (escript)
- .exs files
- Path (expand, split, basename)
- File (cp, mkdir_p, cp_r, mv)

Scripting

```
> { :ok, out } = File.read("/unknown")

** (MatchError) no match of right
hand side value: {:error,:enoent}
```

Scripting

```
> out = File.read!("/unknown")

** (File.Error) could not read
file /unknown: no such file or
directory
```

Mix

- Project generation
- Dependencies management
- Useful every day tasks:
  - project compilation
  - running tests
  - etc

HOME GETTING STARTED DOCS BLOG SOURCE

elixir

```
defprotocol String.Inspect
  only: [BitString, List,

defimpl String.Inspect, fo
  def inspect(false), do:
  def inspect(true),  do:
  def inspect(nil),   do:
  def inspect(:""),   do:

  def inspect(atom) do
```

Elixir is a functional meta-programming aware language built on top of the Erlang VM. It is a dynamic language with flexible and homoiconic syntax that leverages Erlang's abilities to build concurrent, distributed, fault-tolerant applications with hot code upgrades.

Elixir also supports polymorphism via protocols (similar to Clojure's), dynamic records, aliases and first-class support to associative data structures (usually known as dicts or hashes in other programming languages).

Finally, Elixir and Erlang share the same bytecode and data types. This means you can invoke Erlang code from Elixir (and vice-versa) without any conversion or performance hit. This allows a developer to mix the expressiveness of Elixir with the robustness and performance of Erlang.

To install Elixir or learn more about it, check our getting started guide. We also have online documentation available and a Crash Course for Erlang developers.

Highlights

Everything is an expression

```
defmodule Hello do
  IO.puts "Defining the function world"

  def world do
    IO.puts "Hello World"
  end

  IO.puts "Function world defined"
```

News: Elixir v0.6.0 released

Search...

IMPORTANT LINKS

- #elixir-lang on freenode IRC
- Twitter
- Mailing list
- Issues Tracker
- Textmate Bundle
- Vim Elixir
- Crash Course for Erlang developers

NirvanaPlatform/couchie

```
Couchie.open(:default)
#=> {:ok,#PID<0.63.0>}

Couchie.set(:default, "3-18-13-6-57", "value")
Couchie.get(:default, "3-18-13-6-57")
#=> {"3-18-13-6-57",16538602597327634432,"value"}

Couchie.open(:cache, 10, 'localhost:8091', 'cache')
#=> {:ok,#PID<0.77.0>}
```

datahogs/tirexs

```
import Tirexs.Bulk
settings = Tirexs.ElasticSearch.Config.new()

Tirexs.Bulk.store [index: "articles", refresh:
true], settings do
  create id: 1, title: "One", tags: ["elixir"]
  create id: 2, title: "Two", tags: ["ruby"]
  create id: 3, title: "Three", tags: ["java"]
  create id: 4, title: "Four", tags: ["erlang"]
end
```

sasa1977/exactor

```elixir
defmodule Actor do
  use ExActor

  defcast inc(x), state do
    new_state(state + x)
  end

  defcall get, state do
    state
  end
end
```

```elixir
{:ok,act} = Actor.start(1)
Actor.get(act) # 1

Actor.inc(act, 2)
Actor.get(act) # 3
```

elixir

@elixirlang / elixir-lang.org