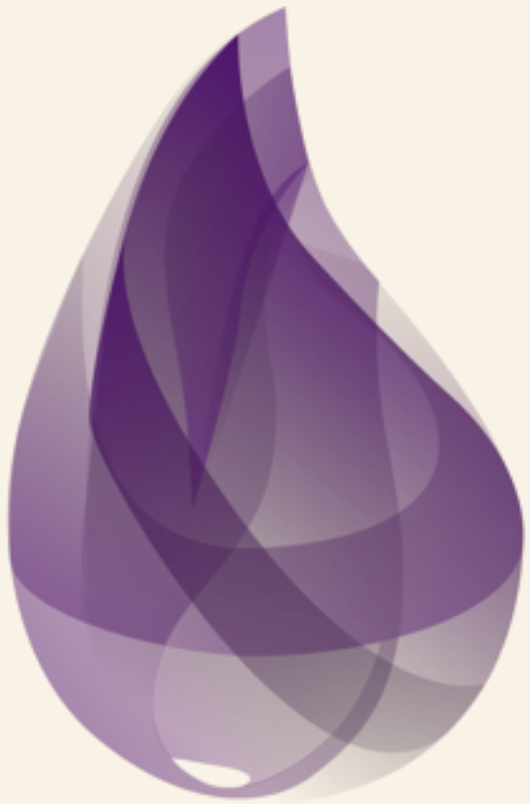
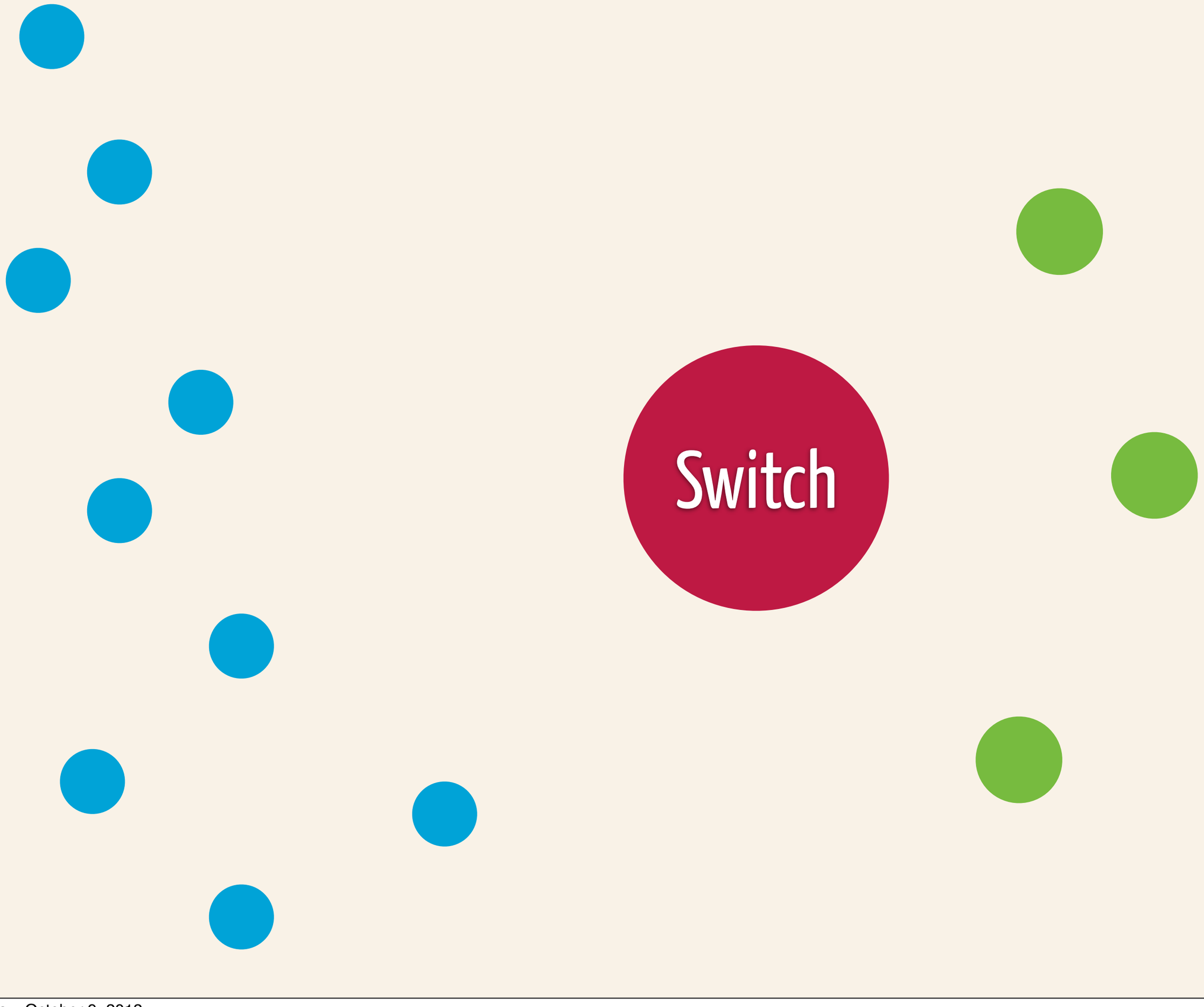


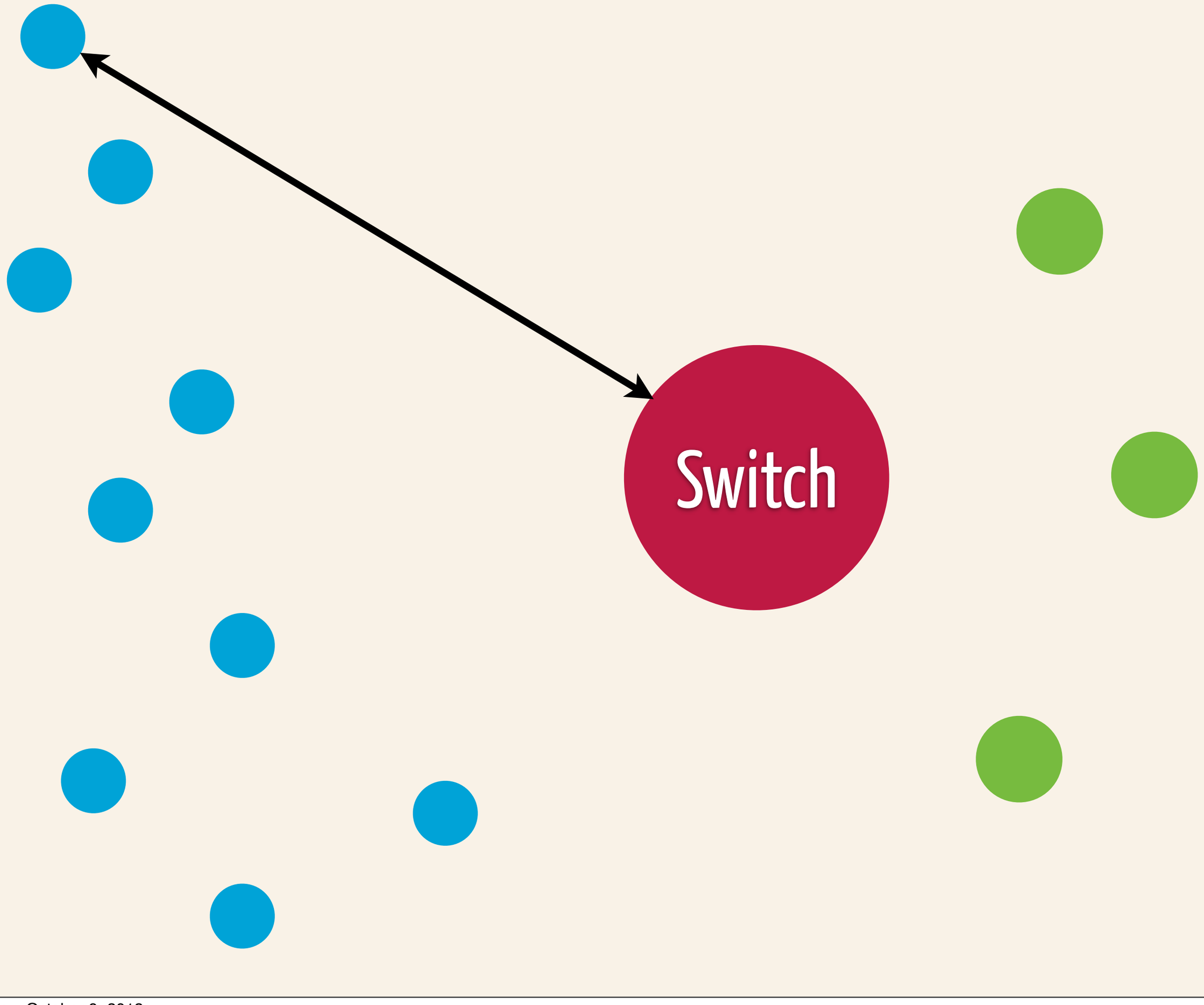
elixir

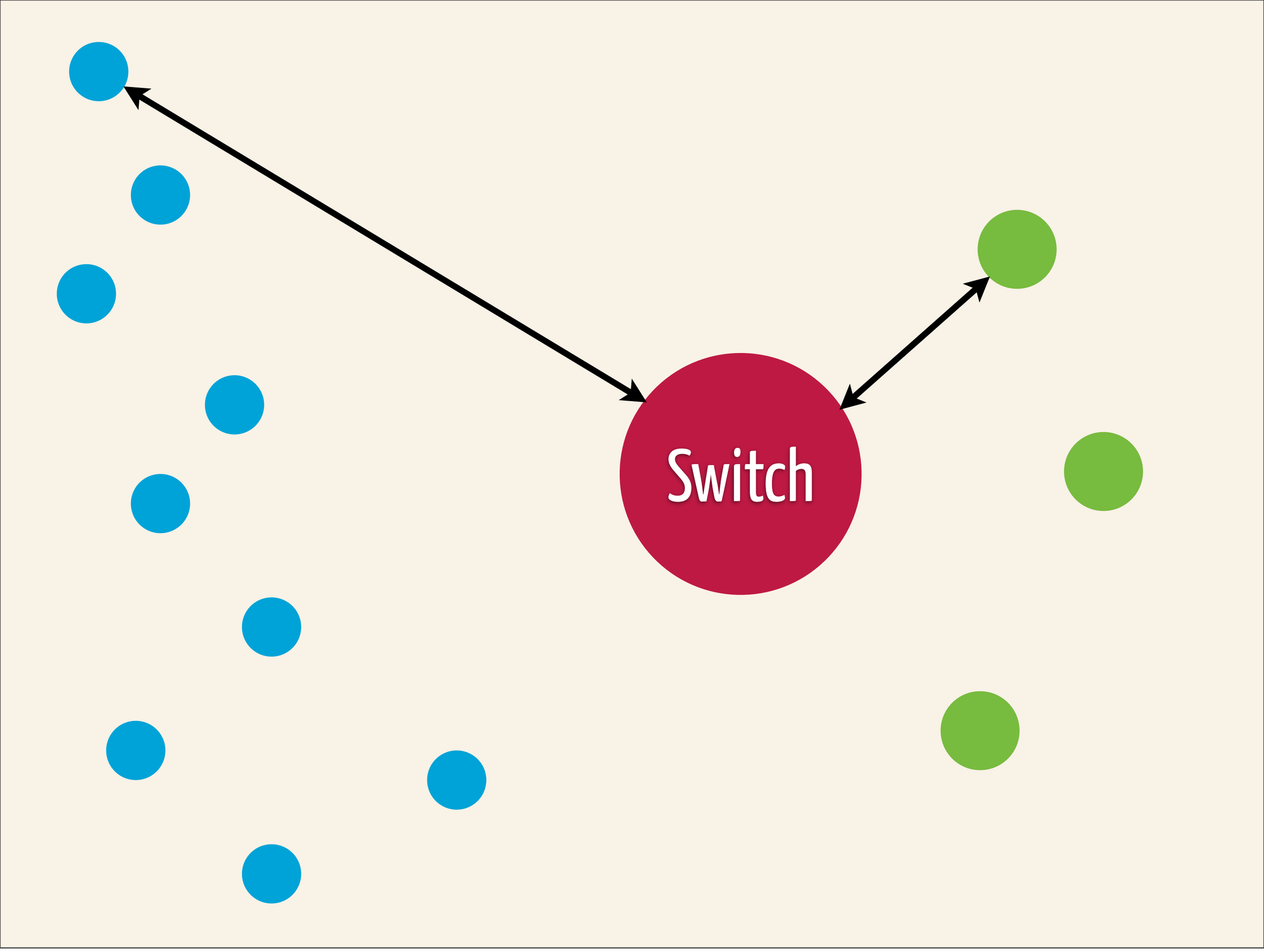
@elixirlang / elixir-lang.org

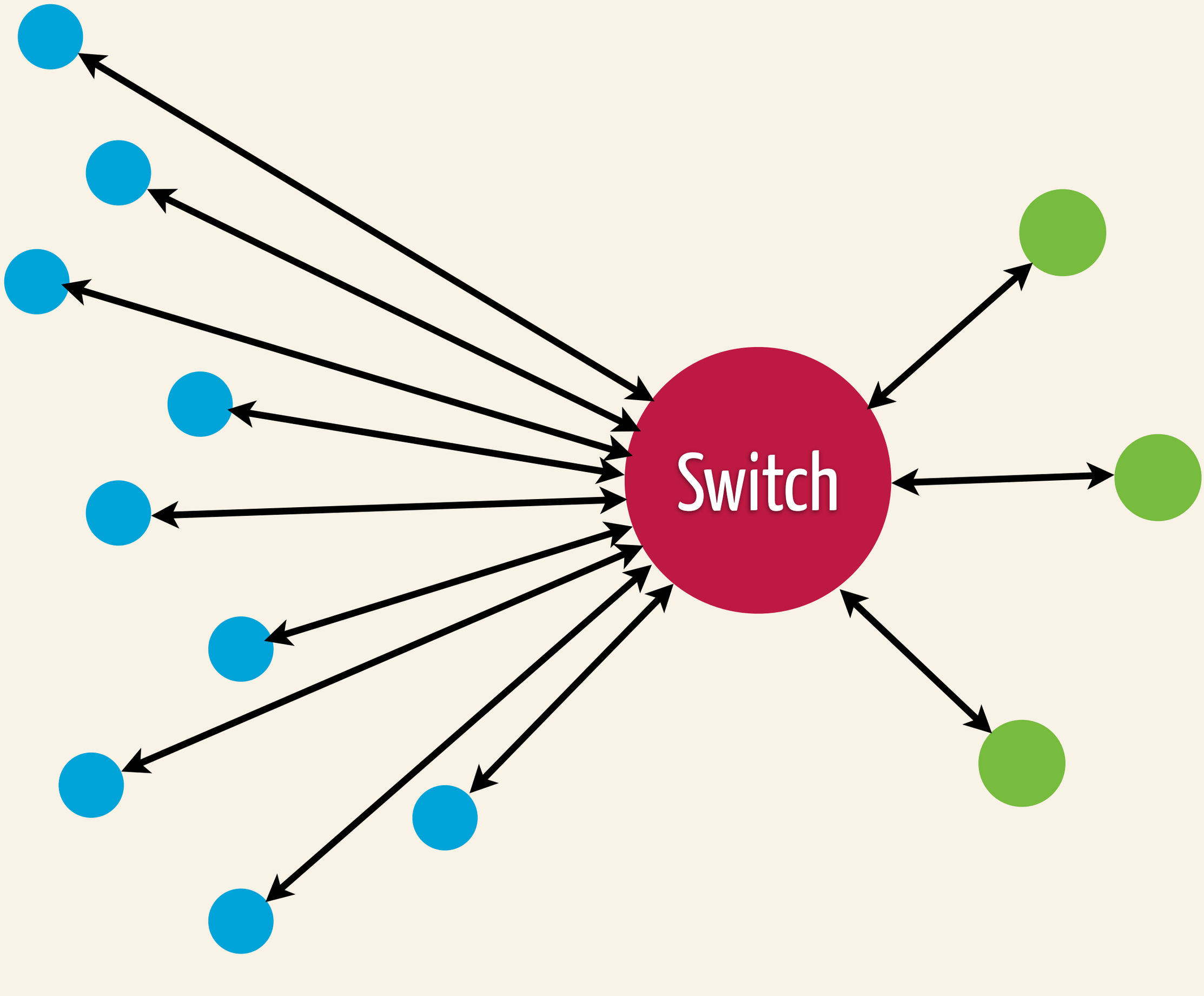
Why?

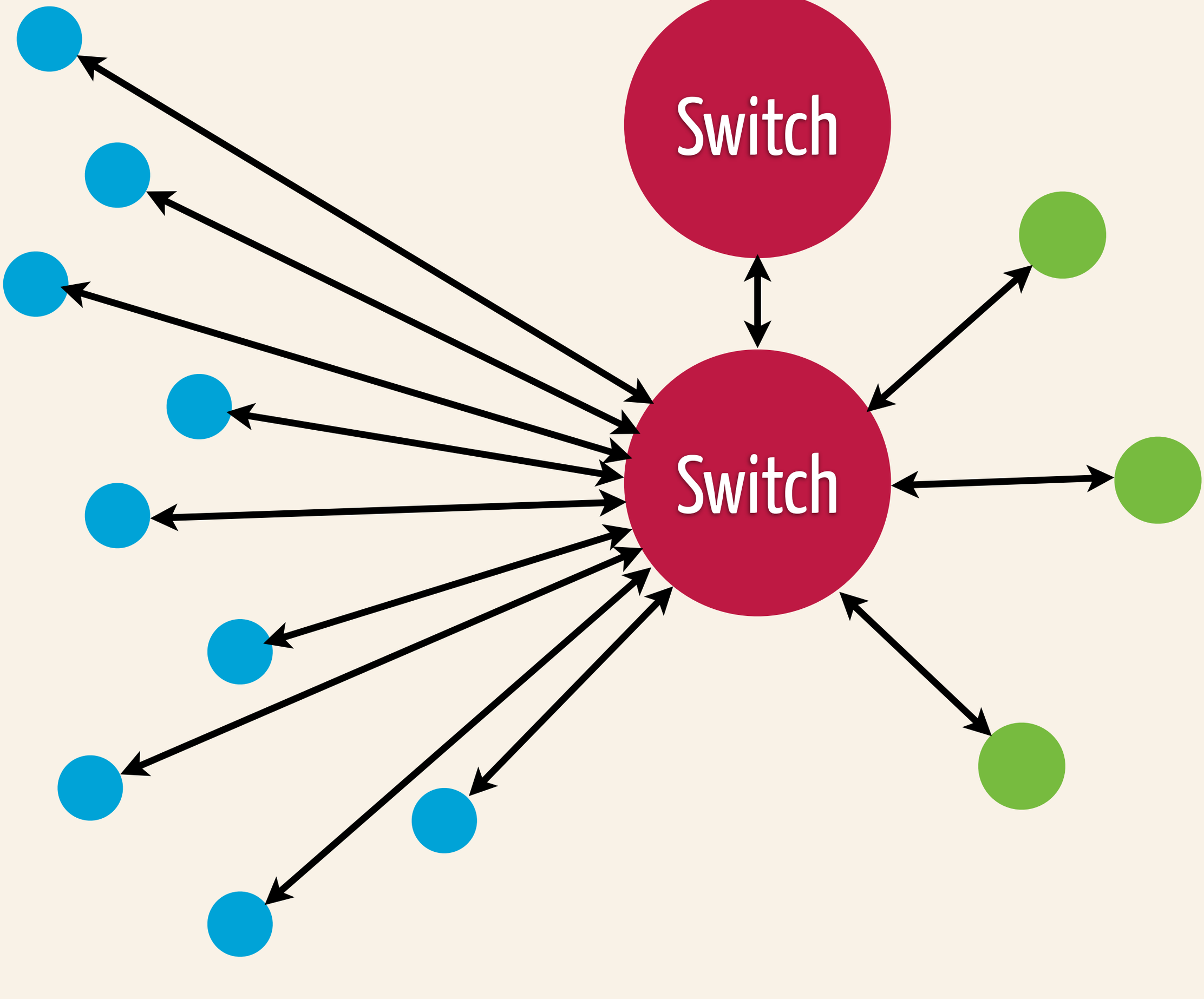
ERLANG

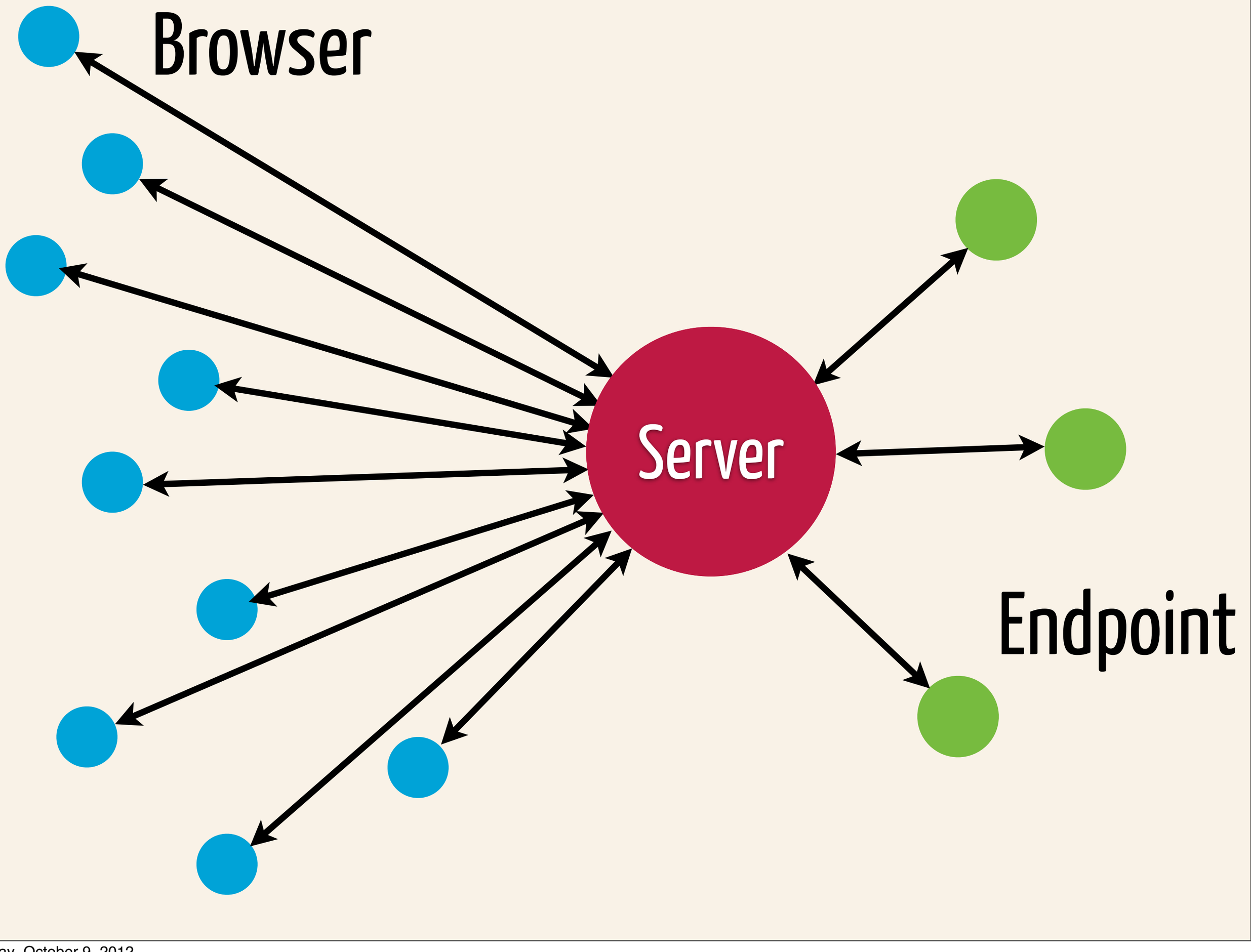

Switch


Switch


Switch

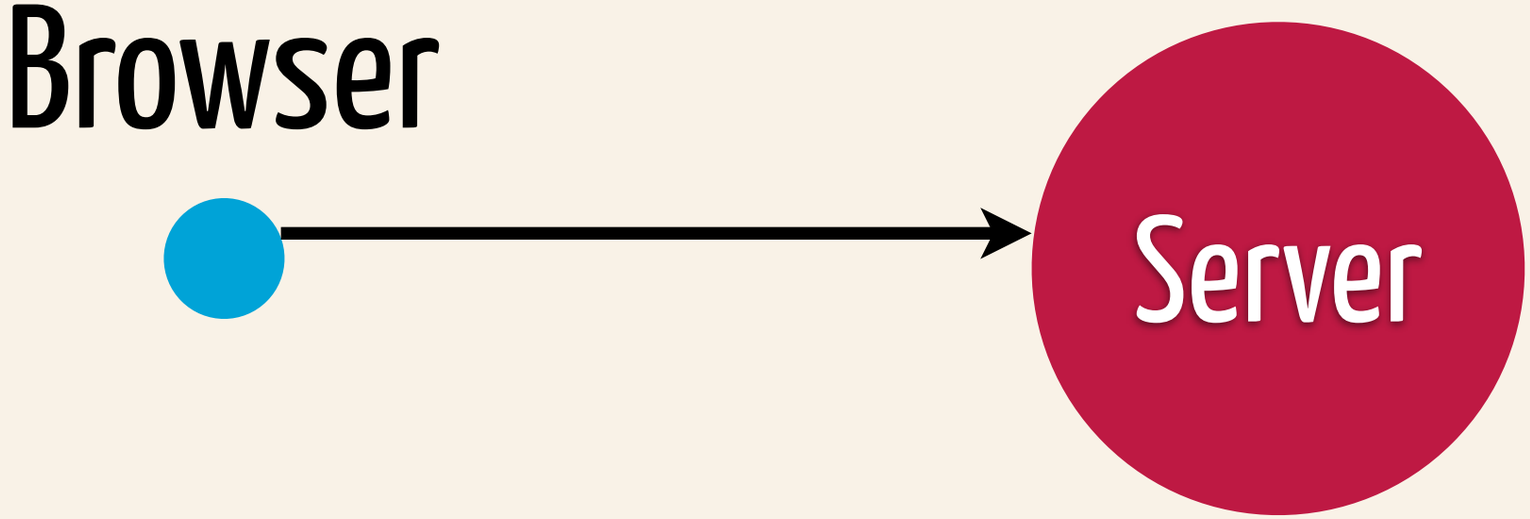

Switch

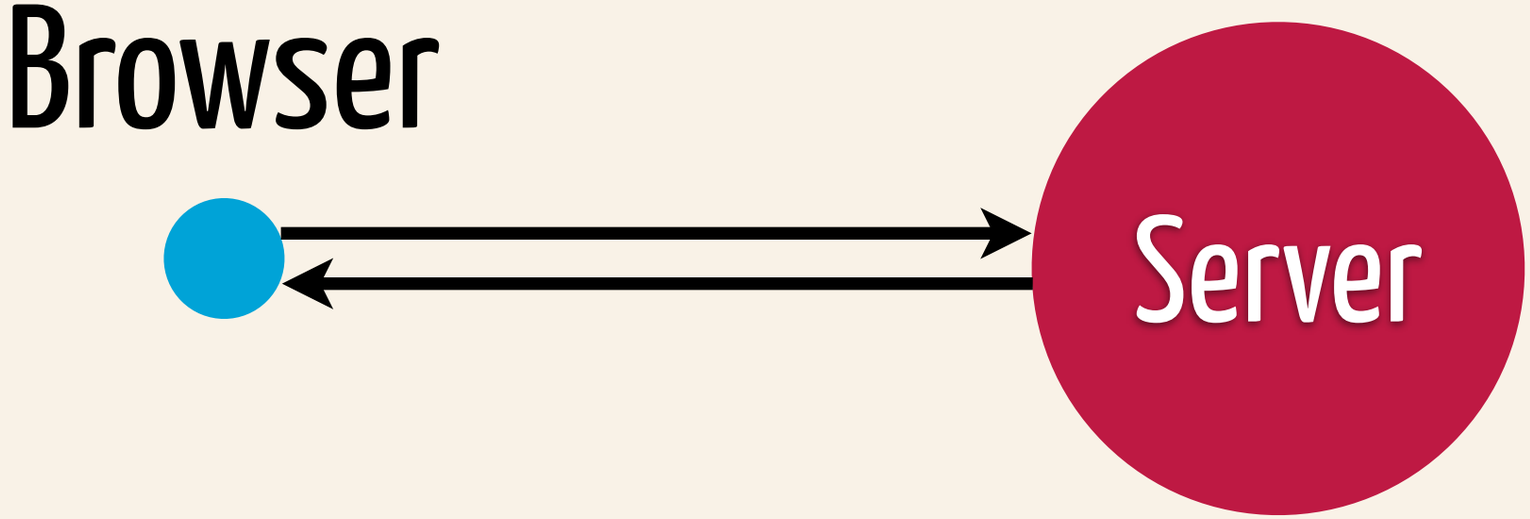

Switch
Switch

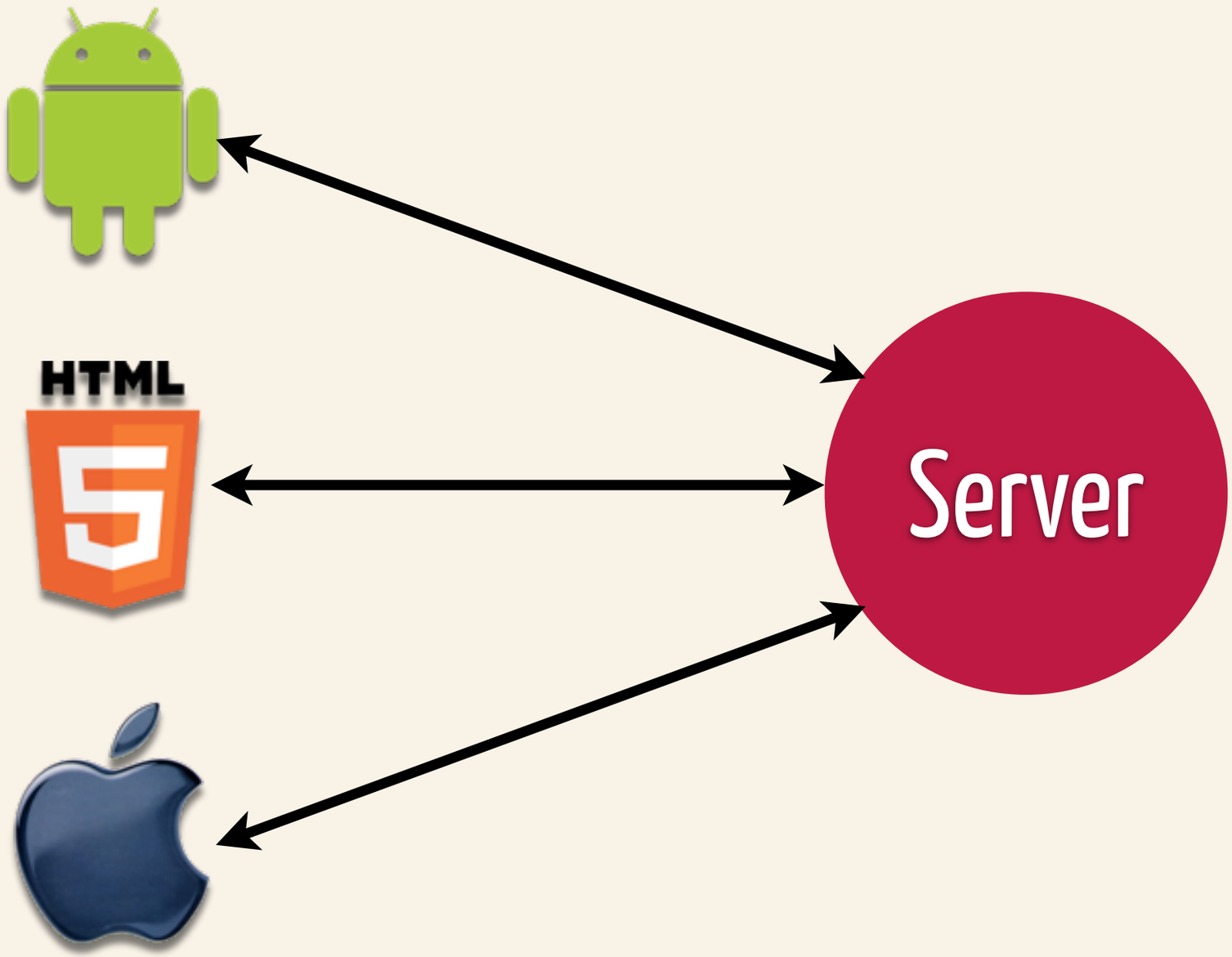

Browser
Server
Endpoint

Browser

Server


Browser
Server


Browser
Server

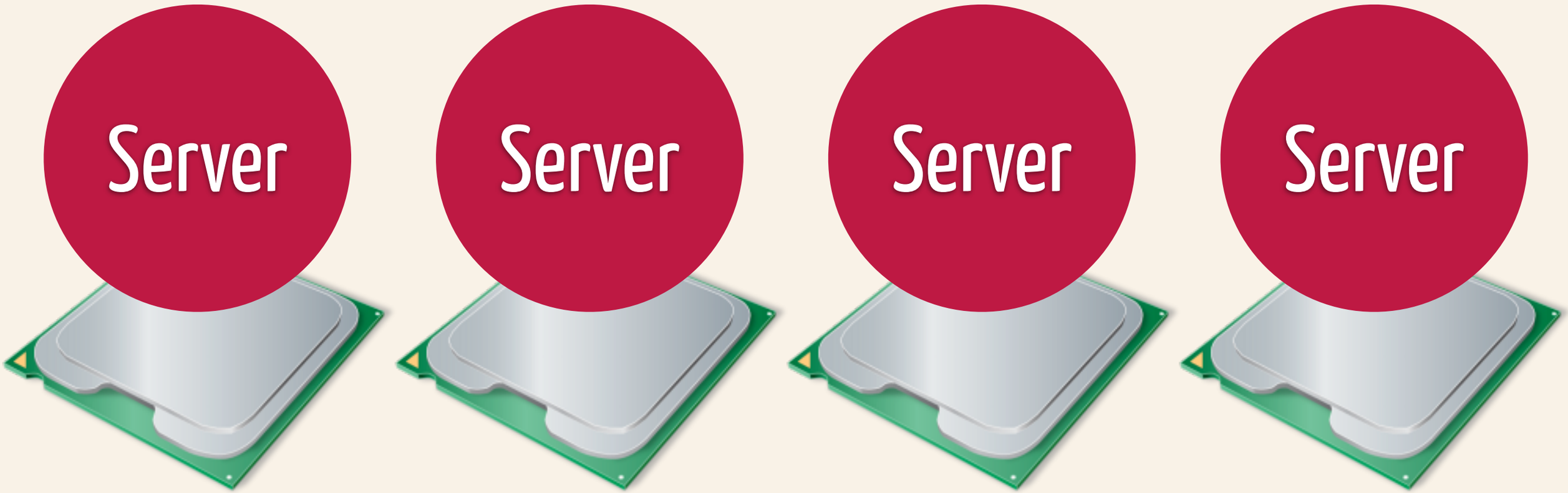

HTML
Server

Server

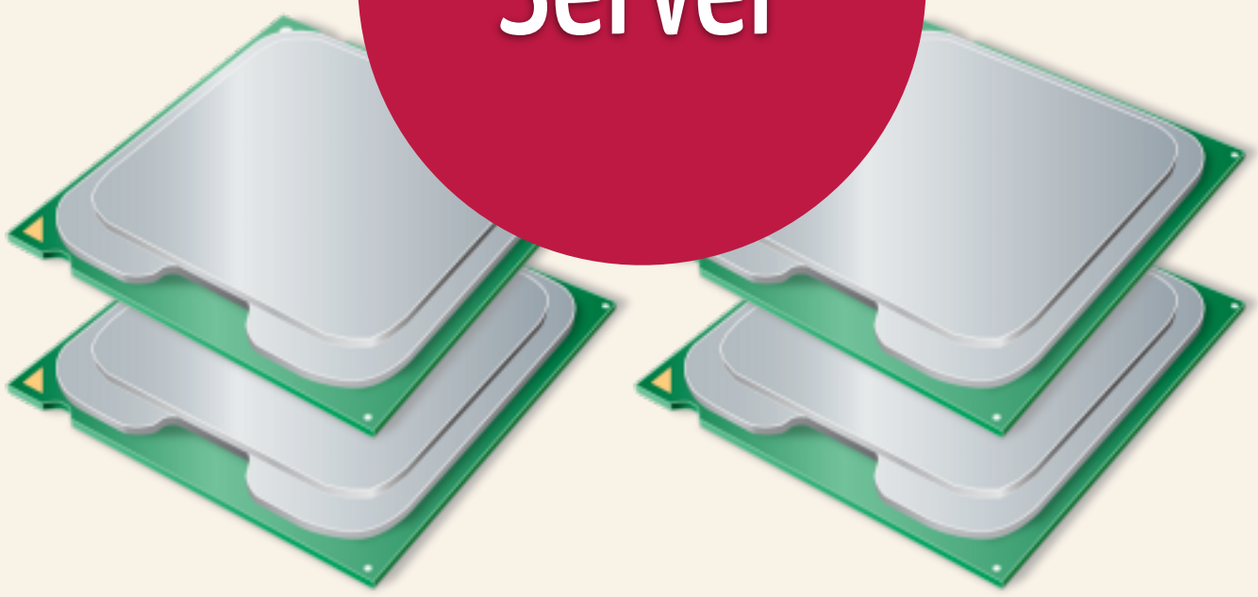
- Web sockets
- API streaming
- Server sent events

case #2

Multi-core


Server
Server
Server
Server

Server



case #3

Doing it live!

2 million connections on a single node

http://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011/

http://stackoverflow.com/questions/1636455/where-is-erlang-used-and-why

Goals

goal #1

Productivity

Everything is an expression

erlang

```erlang
-module(my_module).

some_function(Foo) ->
    % ...

other_function(Bar) ->
    % ...
```

erlang

```erlang
-module(my_module).

% won't compile
io:put_chars("hello").

some_function(Foo) ->
    % ...

other_function(Bar) ->
    % ...
```

elixir

```
defmodule MyModule do
  def some_function(foo) do
    # ...
  end

  IO.puts "hello"

  def other_function(bar) do
    # ...
  end
end
```

Macros

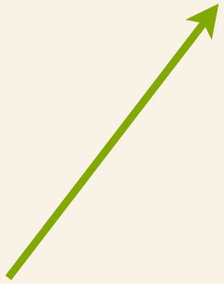
```
:foo                - atoms/symbols
```

```
:foo              - atoms/symbols
{ 1, 2, 3 } - tuples
```

```
:foo              - atoms/symbols
{ 1, 2, 3 } - tuples
[ 1, 2, 3 ] - lists
```

```
is_atom(:foo)
```

atom



```
is_atom(:foo)
```

atom

```
is_atom(:foo)
```

```
{ :is_atom, 1, [:foo] }
```

function

line

args

$$1 + 2$$

1 + 2

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

function

line

args

elixir

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

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

Domain Specific Languages

elixir

```
defmodule MathTest do
  use ExUnit.Case

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

elixir

```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
```

goal #2

Extensibility

Protocols

erlang

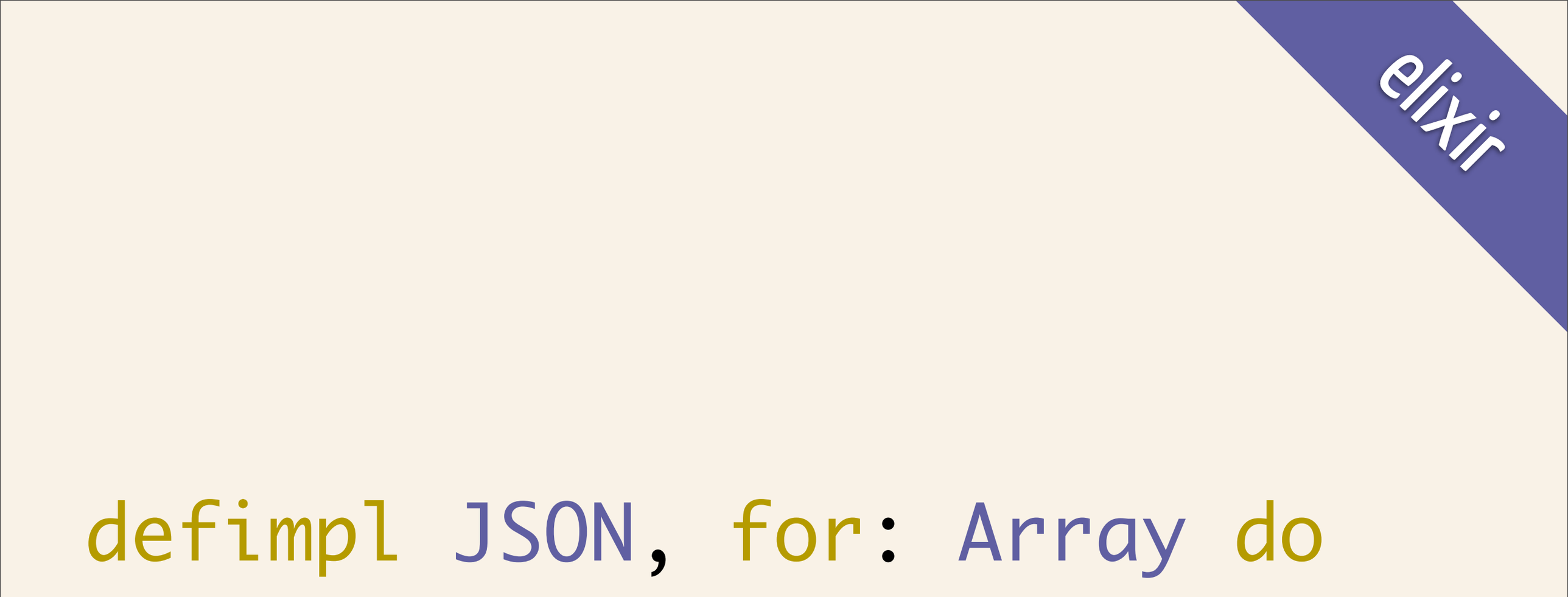
```erlang
-module(json).

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

elixir

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

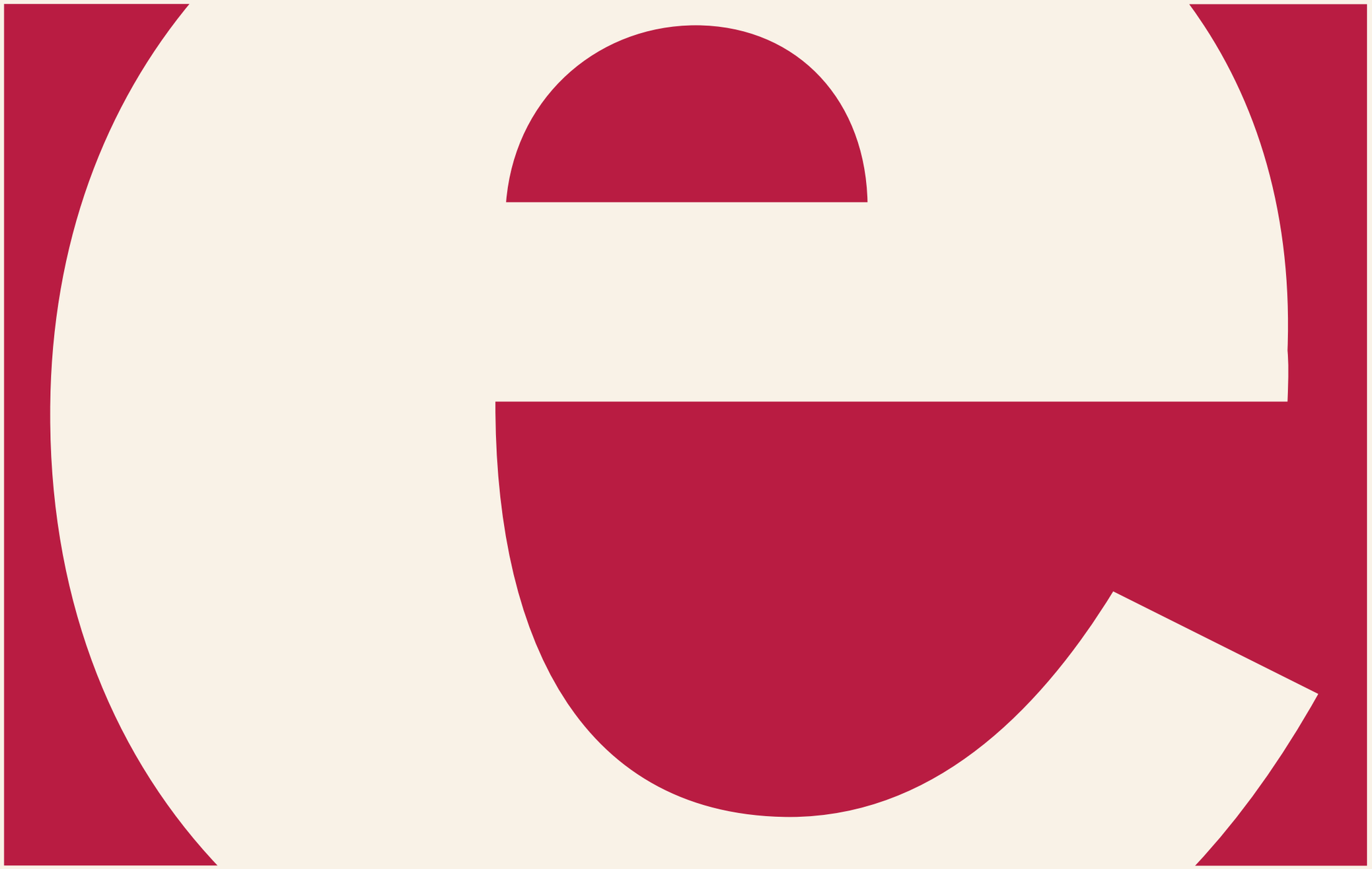
```
JSON.to_json(item)
```

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

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

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


elixir
defimpl JSON, for: Array do
...
end

goal #3

Compatibility



ERLANG

DISTRIBUTED FAULT-TOLERANT APPLICATIONS WITH HOT-CODE SWAPPING

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

Example

elixir

```
defmodule MathTest do
  use ExUnit.Case, async: true

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

1. MathTest
2. SystemTest
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...

ExUnit.
Runner

ExUnit.
Formatter

1 MathTest

2 SystemTest

3 ...

4 ...

5 ...

6 ...

7 ...

8 ...

run

ExUnit.
Runner

ExUnit.
Formatter

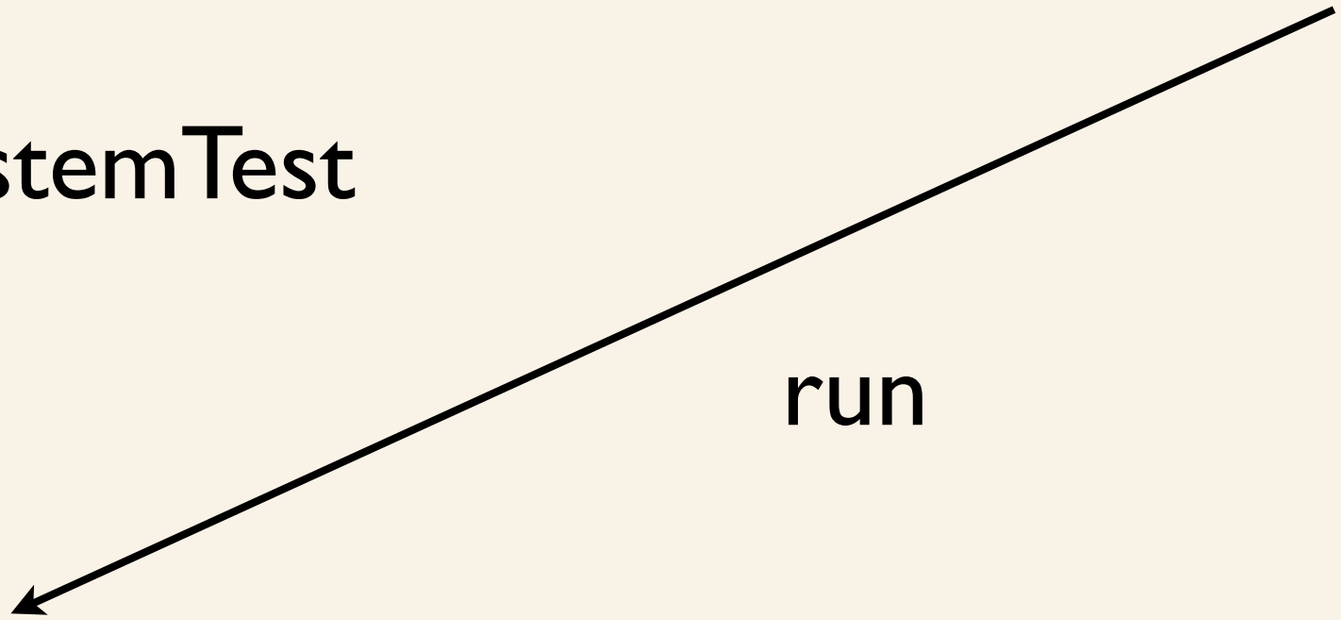
1. MathTest
2. SystemTest
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...

run

ExUnit.
Runner

ExUnit.
Formatter

1. MathTest
2. SystemTest
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...



ExUnit.
Runner

ExUnit.
Formatter

1. MathTest
2. SystemTest
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...

ExUnit.
Runner

ExUnit.
Formatter

1 MathTest

2 SystemTest

3 ...

4 ...

5 ...

6 ...

7 ...

8 ...

done

ExUnit.
Runner

ExUnit.
Formatter

1 MathTest

2 SystemTest

3 ...

4 ...

5 ...

6 ...

7 ...

8 ...

done

ExUnit.
Runner

info

ExUnit.
Formatter

1 MathTest

2 SystemTest

3 ...

4 ...

5 ...

6 ...

7 ...

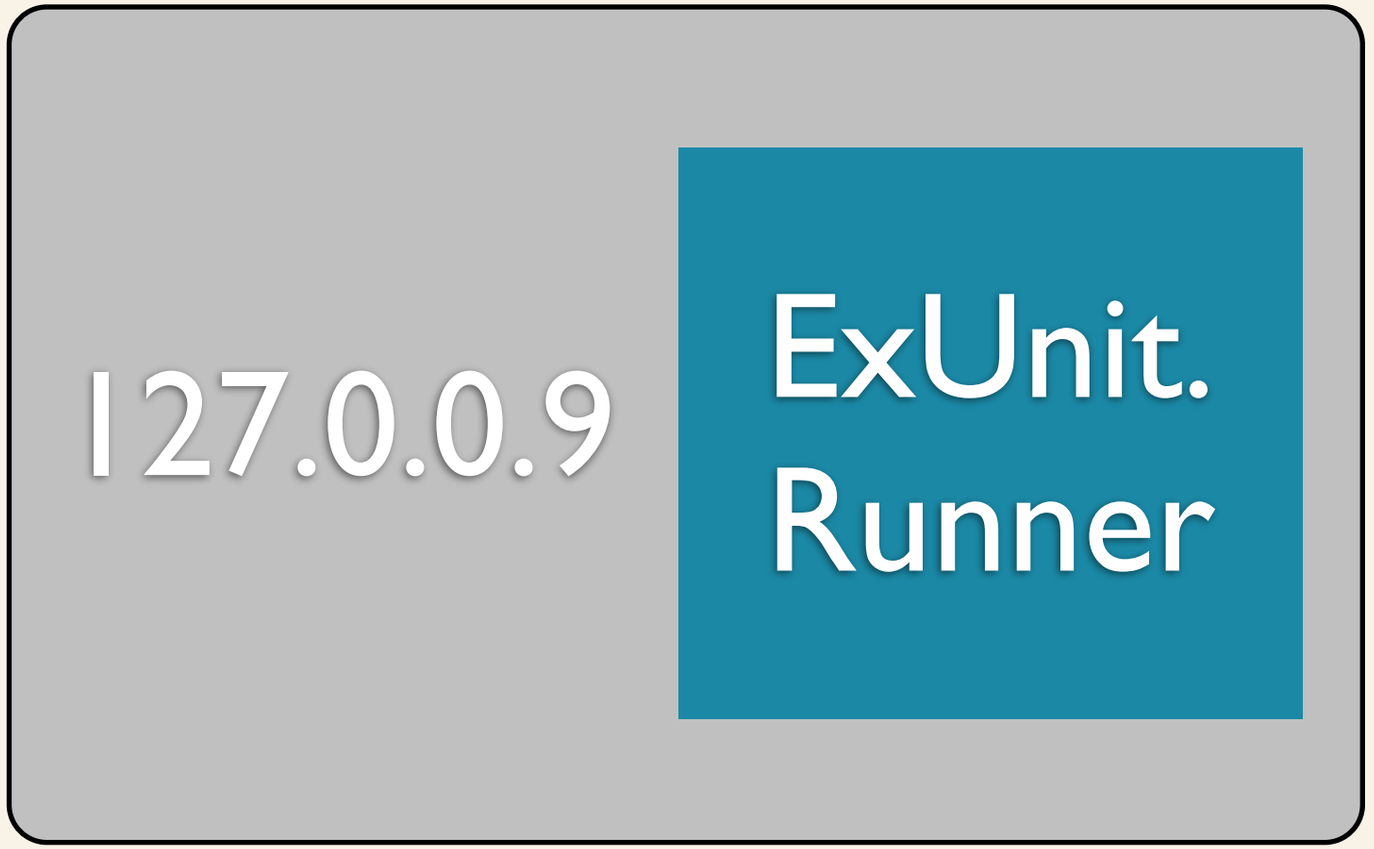
8 ...

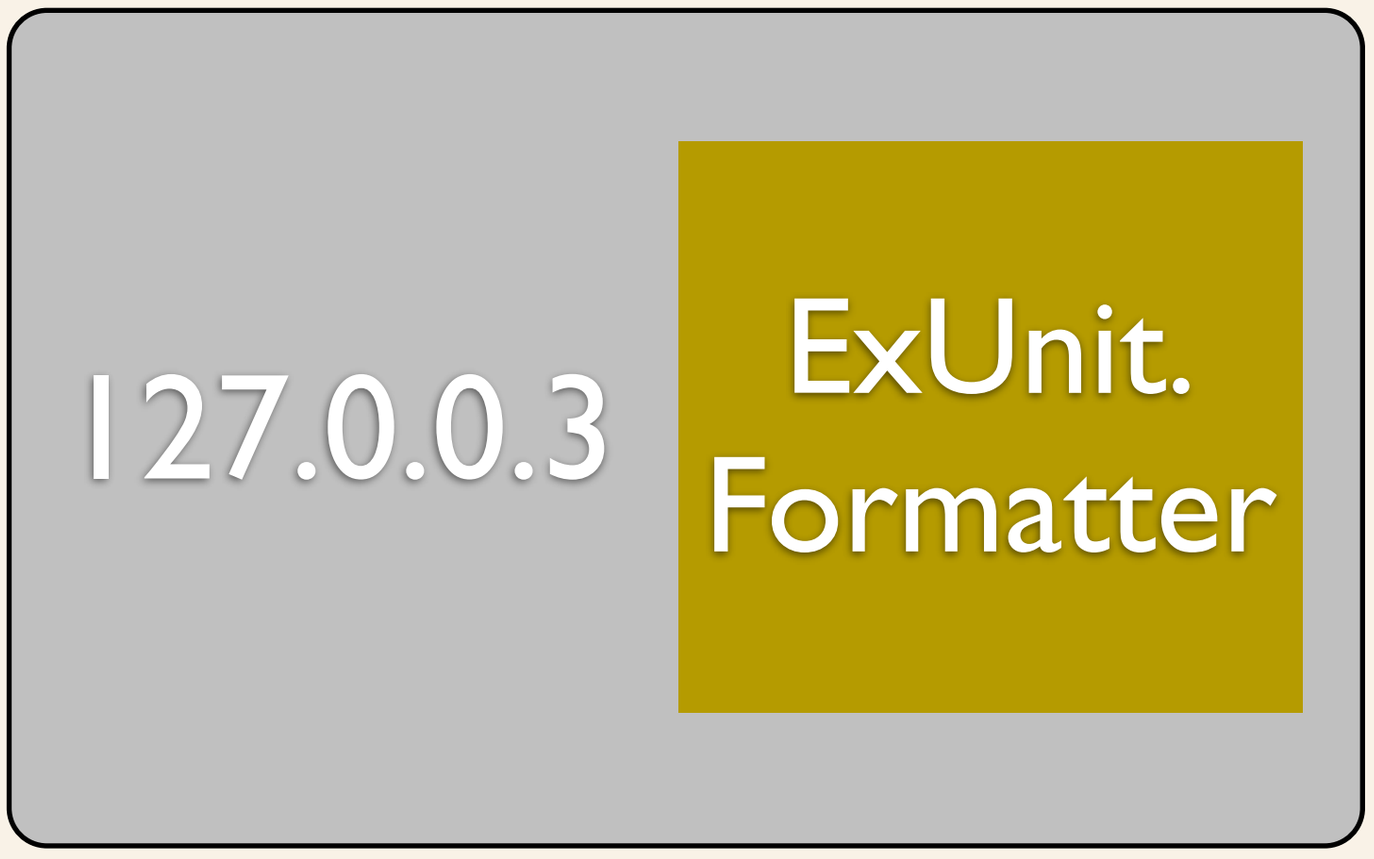
run

ExUnit.
Runner

ExUnit.
Formatter

1. MathTest
2. SystemTest
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...





1. MathTest
2. SystemTest
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...

ExUnit.
Supervisor

127.0.0.9

ExUnit.
Runner

127.0.0.3

ExUnit.
Formatter

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.

News: ixir v0.6.0 released

Search...

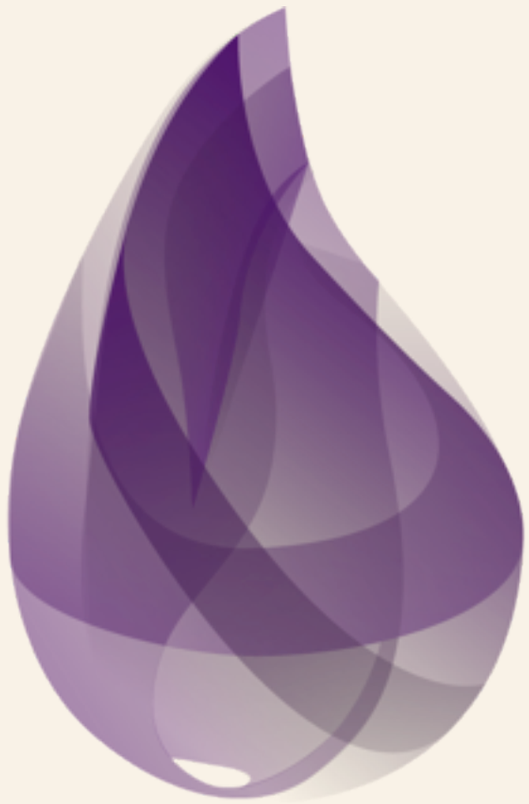
IMPORTANT LINK

- #elixir-lang on freenode IRC
- Twitter
- Mailing list
- Issues Tracker
- Textmate Bundle
- Vim Elixir
- 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"
```



elixir

@elixirlang / elixir-lang.org