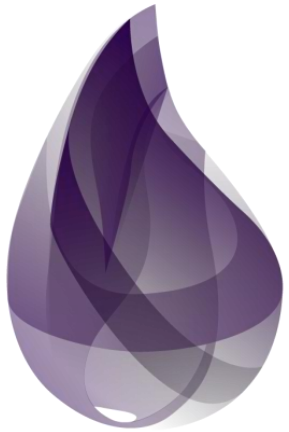
Hot-or-Not Elixir
with José Valim

SOURCE OF YOUR TECHNOLOGY

Author: Paul Zenden

|> 

|> elixir 

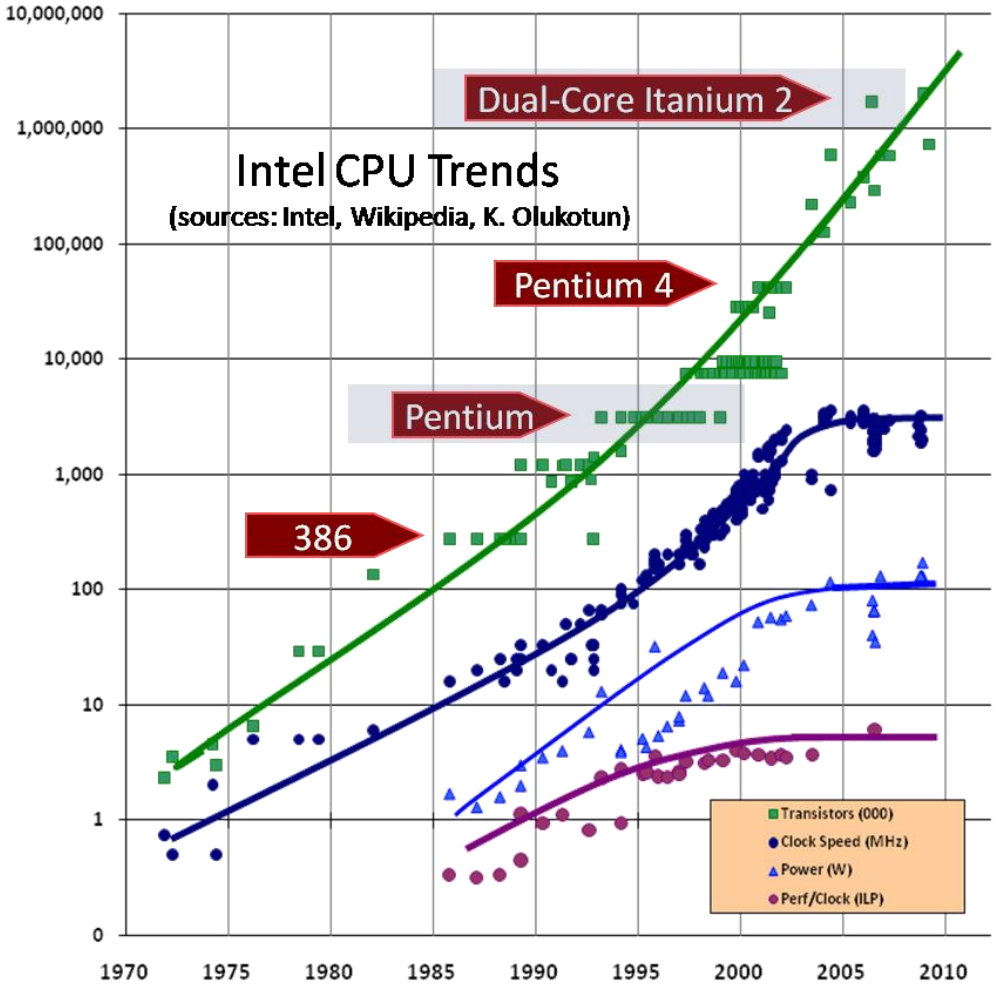
```
with {:ok, :"José Valim"} <-
                creator_of_elixir()
do
        hot_or_not()
end
```

Timetable

18:00h	Introduction
18:05h	Elixir (José Valim)
19:30h	Break
20:00h	Demonstrator & Elixir in real practice (Sioux team)
20:45h	Q & A
21:00h	Drinks

#End of Program

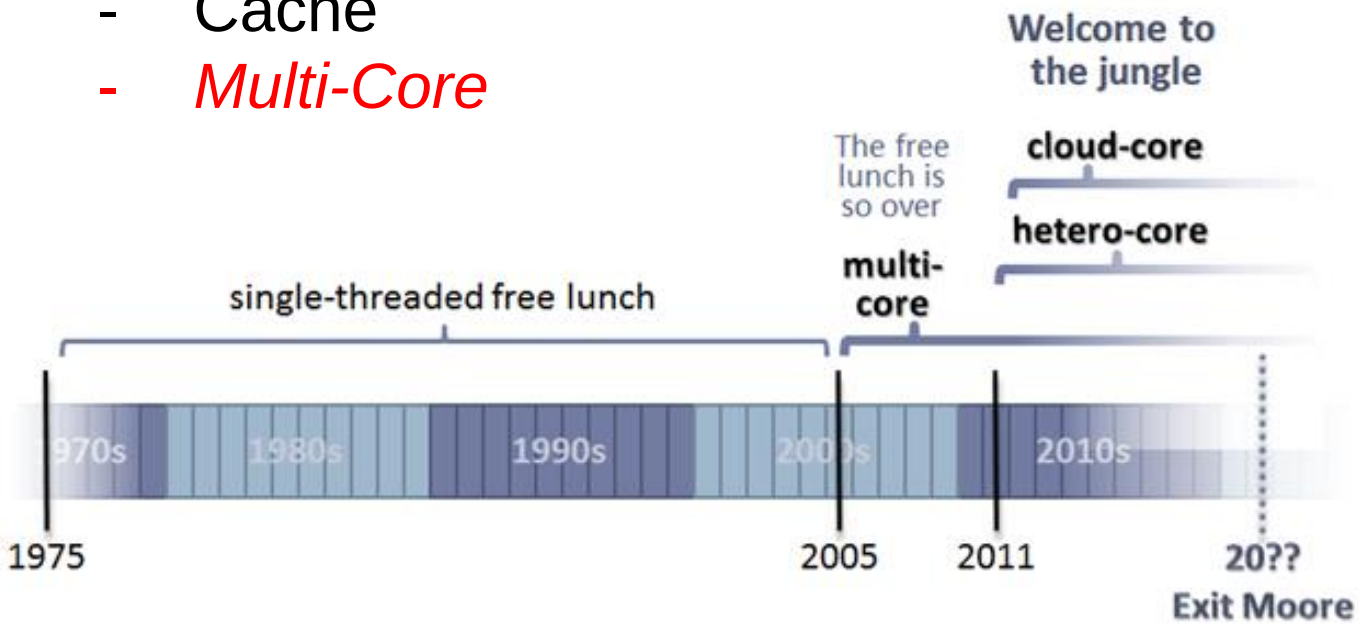
The Free Lunch Is Over
A fundamental turn towards concurrency



Next gen CPU → no longer faster!

New performance drivers:

- Hyperthreading
- Cache
- *Multi-Core*



https://herbsutter.com/welcome-to-the-jungle/
http://www.gotw.ca/publications/concurrency-ddj.htm

Software expectations double every year!

- Embrace concurrency (to exploit new hw)
- More, complex features
- Always available
- Scalable
- Responsive
- DevOps
- Zero downtime deployment
- …

José Valim will explain how can help us …

Founder and Lead Developer at Plataformatec.
Creator of Elixir

… to cope with these challenges.

The Floor is Yours

elixir

@elixirlang / elixir-lang.org

4 Thread Safety

The work done to make Rails thread-safe is rolling out in Rails 2.2. Dep
infrastructure, this means you can handle more requests with fewer cop
leading to better server performance and higher utilization of multiple c

To enable multithreaded dispatching in production mode of your applic
in your `config/environments/production.rb`:

```
config.threadsafe!
```

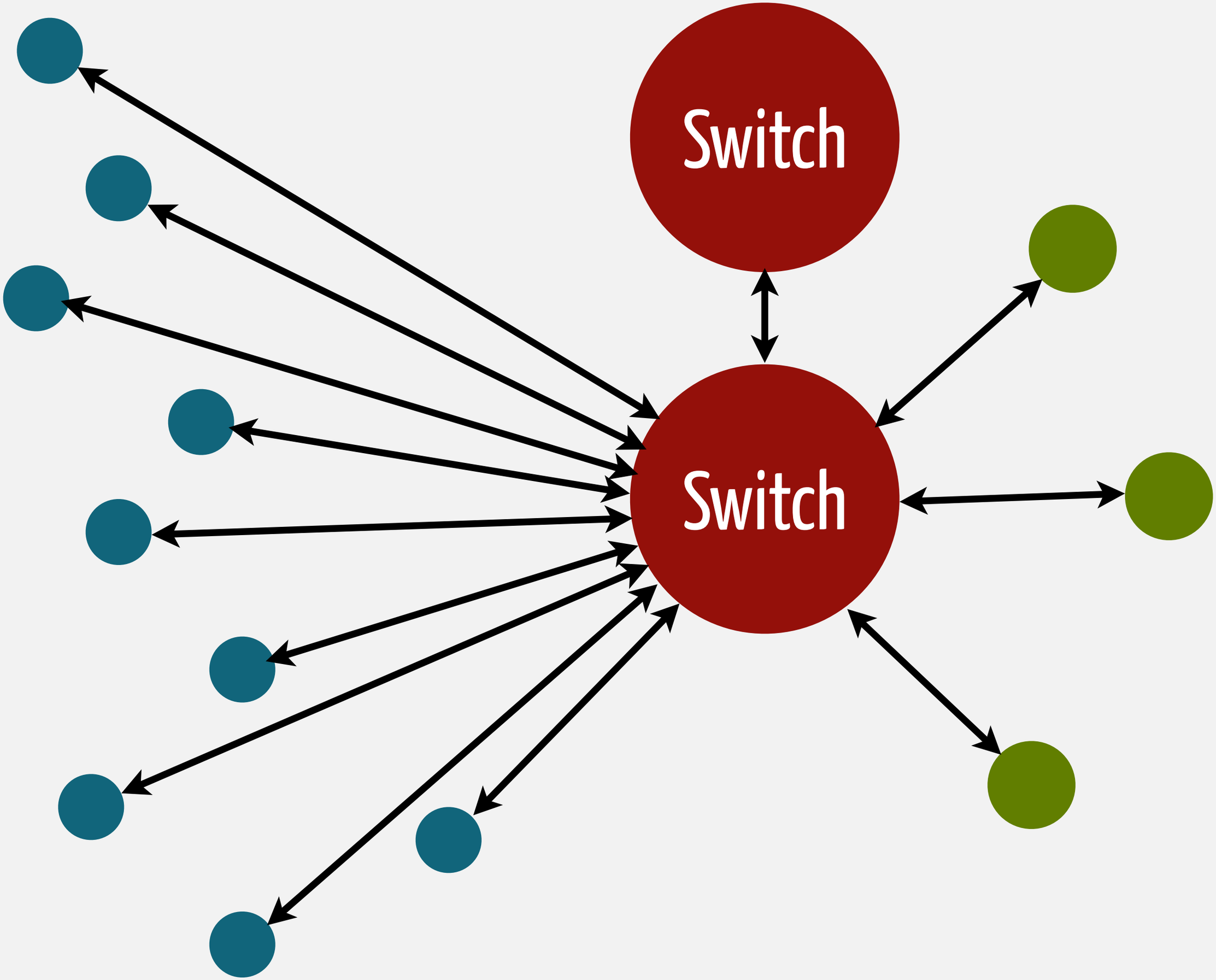
- More information :
  - Thread safety for your Rails
  - Thread safety project announcement
  - Q/A: What Thread-safe Rails Means

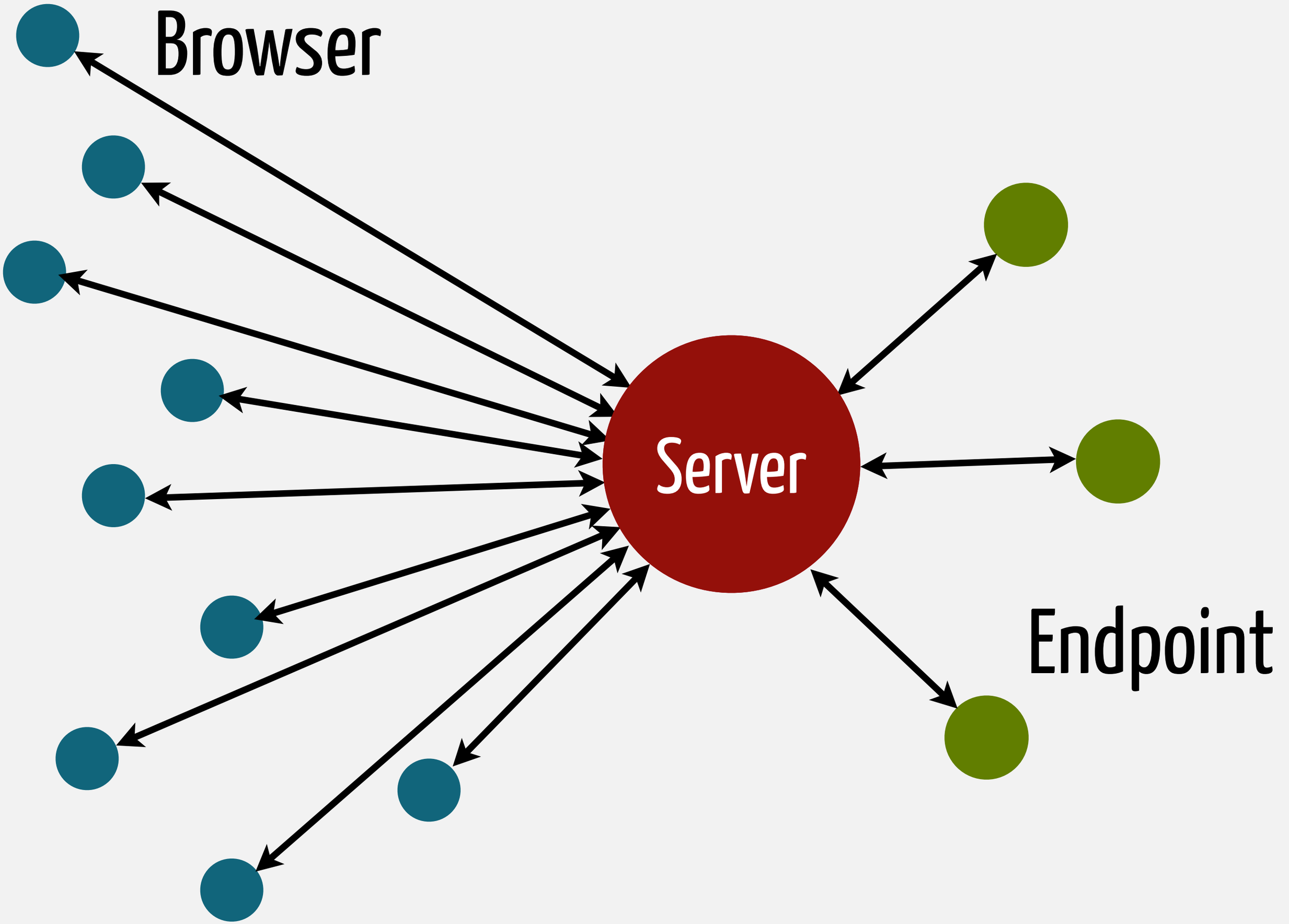
Rails 2.2
threadsafe

Functional programming

- Explicit instead of implicit state
- Transformation instead of mutation

ERLANG


Switch
Switch


Browser
Server
Endpoint

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

2 million connections on a single node

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

Intel Xeon CPU X5675 @ 3.07GHz
24 CPU - 96GB
Using 40% of CPU and Memory

elixir

- Functional
- Concurrent
- **Distributed**

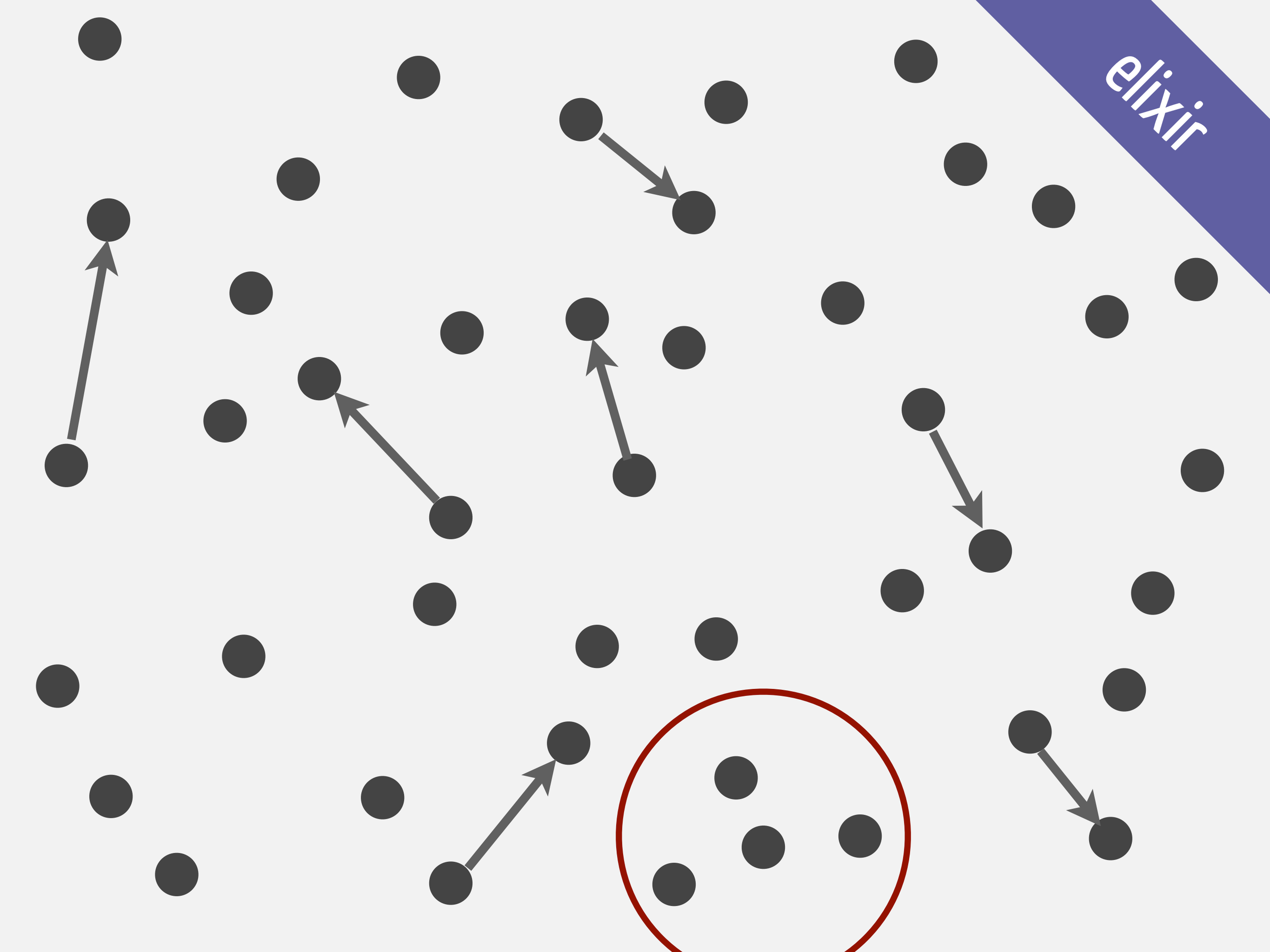
Idioms

Sequential code

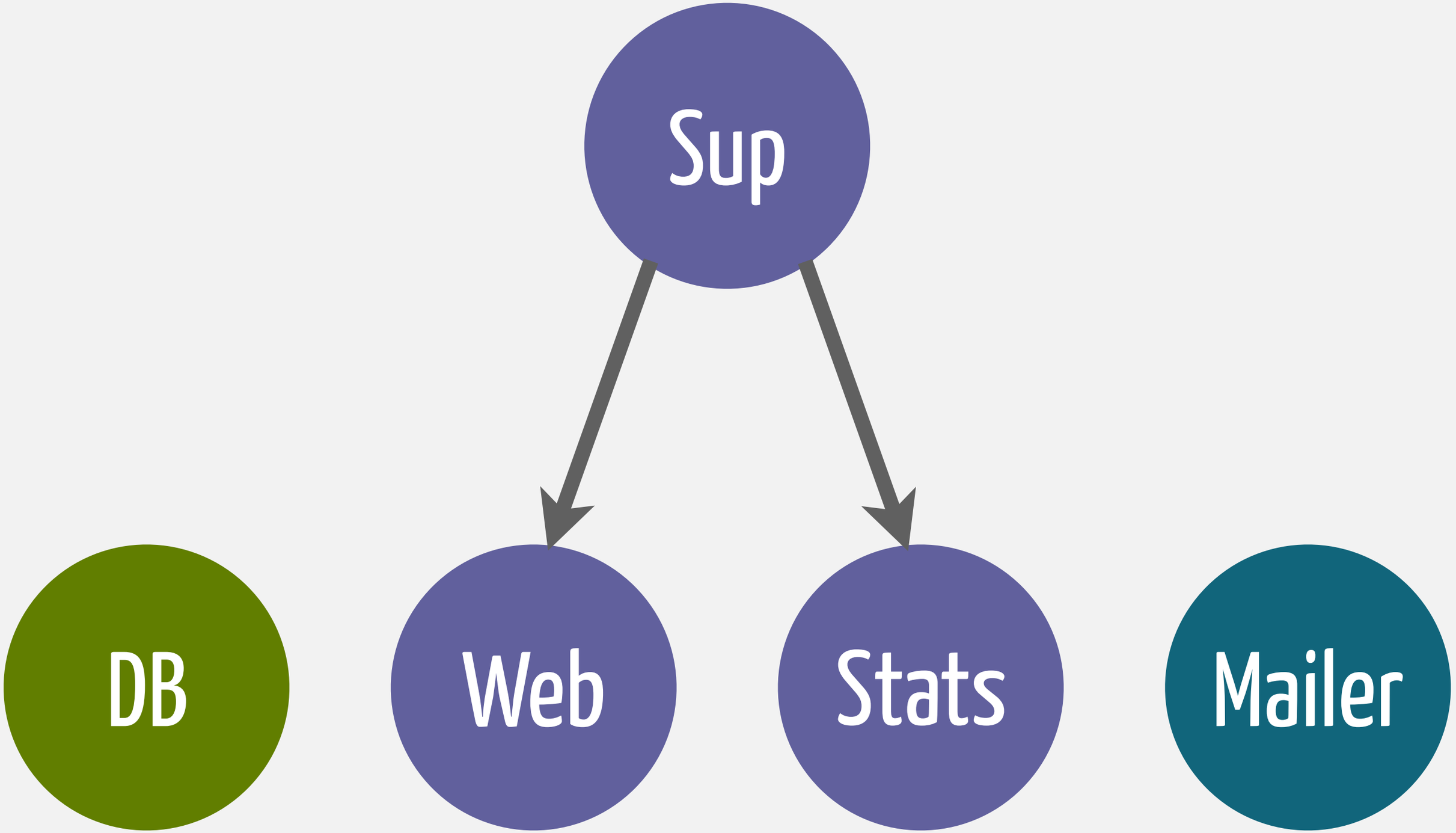
elixir

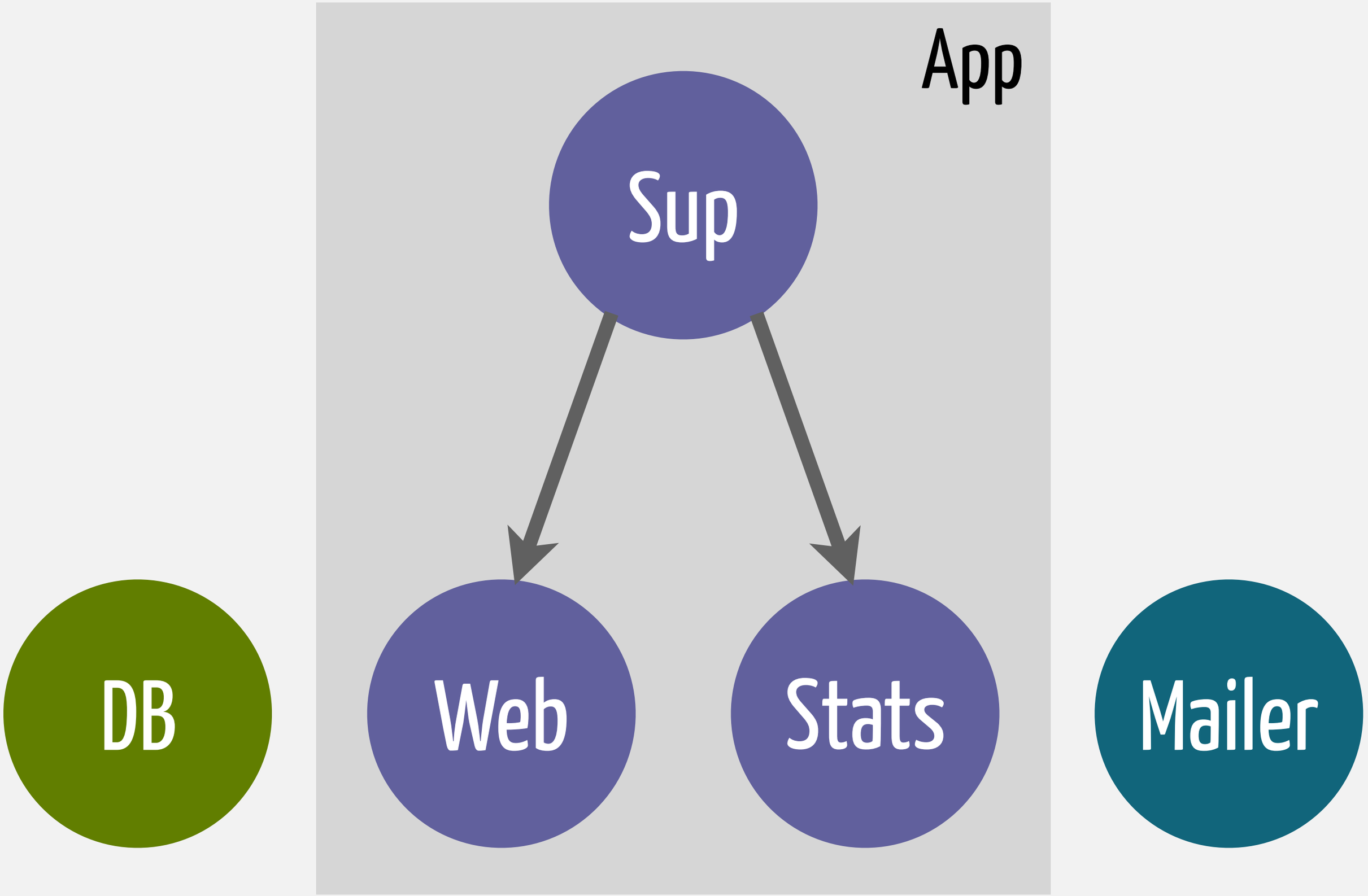
Sequential code

elixir


elixir


DB
Web
Stats
Mailer


Sup
DB
Web
Stats
Mailer


App
Sup
DB
Web
Stats
Mailer

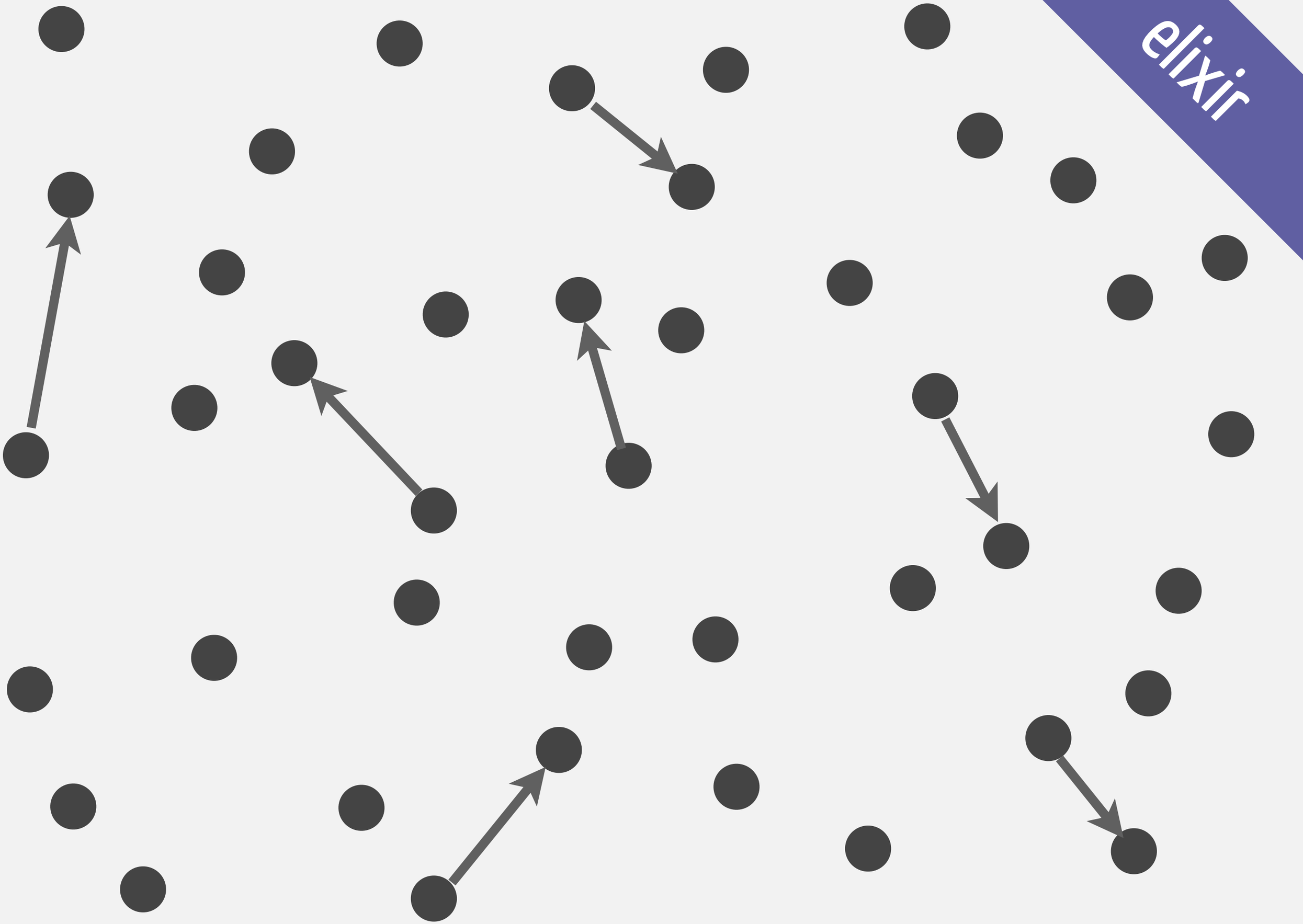
Observer Demo

Applications

- Introspection & Monitoring
- Visibility of the application state
- Easy to break into "components"
- Reasoning when things go wrong

- Processes
- Supervisors
- Applications
- Message passing

- Concurrent
- Fail fast
- Fault tolerant
- Distributed?


elixir

elixir

app1@local

app2@local

elixir

- Compatibility
- Extensibility
- Productivity

goal #1

Compatibility

goal #2

Extensibility

Now we need to go meta. We should now think of a language design as being a pattern for language designs. A tool for making more tools of the same kind.

Guy Steele – “Growing a language”
at ACM OOPSLA 1998

elixir

```elixir
defmodule MathTest do
  use ExUnit.Case

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

```
~/OSS/elixir[master *]$ elixir lib/ex_unit/examples/difference.exs


  1) test strings (Difference)
     lib/ex_unit/examples/difference.exs:10
     Assertion with == failed
     code:  string1 == string2
     left:  "hello world "
     right: "hello world!"
     stacktrace:
       lib/ex_unit/examples/difference.exs:13: (test)

  2) test keyword lists; reverse order (Difference)
     lib/ex_unit/examples/difference.exs:16
     Assertion with == failed
     code:  keyword1 == keyword2
     left:  [port: 4000, max_connections: 1000]
     right: [max_connections: 1000, port: 4000]
     stacktrace:
       lib/ex_unit/examples/difference.exs:19: (test)



Finished in 0.03 seconds (0.03s on load, 0.00s on tests)
2 tests, 2 failures
```

```
  from p in Post,
where: p.published_at < now and
       p.author == “José”,
order: p.created_at
```

goal #3

Productivity

- First-class documentation
- Tooling (ExUnit, IEx, Mix)
- Hex packages

Kernel

Provides the default macros and functions Elixir imports into your environment.

These macros and functions can be skipped or cherry-picked via the `import` macro. For instance, if you want to tell Elixir not to import the `if` macro, you can do:

```
import Kernel, except: [if: 2]
```

Elixir also has special forms that are always imported and cannot be skipped. These are described in `Kernel.SpecialForms`.

Some of the functions described in this module are inlined by the Elixir compiler into their Erlang counterparts in the `:erlang` module. Those functions are called BIFs (builtin internal functions) in Erlang-land and they exhibit interesting properties, as some of them are allowed in guards and others are used for compiler optimizations.

```
~/OSS/phoenix[master]$ iex
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]

Interactive Elixir (1.2.0-dev) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> h Kernel
```

Kernel

Provides the default macros and functions Elixir imports into your environment.

These macros and functions can be skipped or cherry- picked via the import
macro. For instance, if you want to tell Elixir not to import the if macro, you
can do:

```
| import Kernel, except: [if: 2]
```

Elixir also has special forms that are always imported and cannot be skipped.
These are described in Kernel.SpecialForms.

Some of the functions described in this module are inlined by the Elixir
compiler into their Erlang counterparts in the :erlang module. Those functions
are called BIFs (builtin internal functions) in Erlang-land and they exhibit
interesting properties, as some of them are allowed in guards and others are
used for compiler optimizations.

Most of the inlined functions can be seen in effect when capturing the
function:

```
| iex> &Kernel.is_atom/1
| &:erlang.is_atom/1
```

Those functions will be explicitly marked in their docs as "inlined by the
compiler".

```
iex(2)>
```

ALCHEMIST

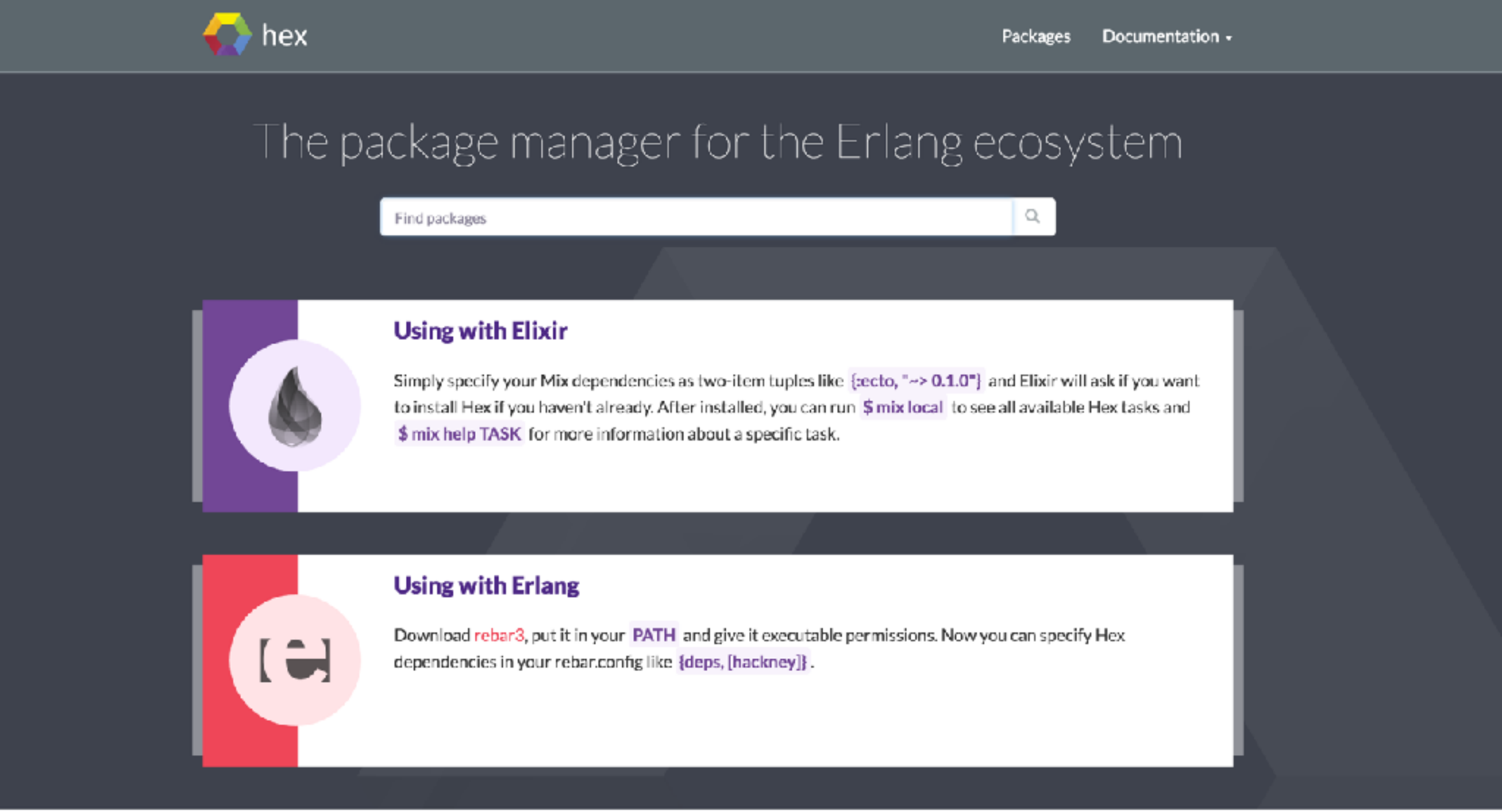
Elixir Tooling Integration Into Emacs

What Does Alchemist Do For You?

Alchemist brings you all the Elixir tooling and power inside your Emacs editor.

Alchemist comes with a bunch of features, which are:

- ☆ Compile & Execution
- ☆ Inline code evaluation
- ☆ Mix integration
- ☆ Documentation lookup
- ☆ Code definition lookup
- ☆ Smart code completion
- ☆ Powerful IEx integration



2 813 packages available

13 188 package versions

64 898 downloads yesterday

1 136 627 downloads last 7 days

51 728 251 downloads all time

hex.pm

Demo time!

Elixir

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 dynamic, functional language designed for building scalable and maintainable applications.

Elixir leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain.

To learn more about Elixir, check our getting started guide. Or keep reading to get an overview of the platform, language and tools.

Platform features

Scalability

All Elixir code runs inside lightweight threads of execution (called processes) that are isolated and exchange information via messages:

```
parent = self()

# Spawns an Elixir process (not an operating system one!)
```

News: Elixir v1.0 released

Search...

THE COMMUNITY

- #elixir-lang on freenode IRC
- elixir-talk mailing list (questions)
- elixir-core mailing list (development)
- Issue tracker
- @elixirlang on Twitter

IMPORTANT LINKS

- Source Code
- Wiki with events, resources and talks organized by the community
- Crash course for Erlang developers

elixir-lang.org

The Pragmatic Programmers

Pragmatic Bookshelf

Metaprogramming Elixir

Write Less Code, Get More Done (and Have Fun!)

The Pragmatic Programmers

Programming Elixir

Functional
|> Concurrent
|> Pragmatic
|> Fun

Dave Thomas

Foreword by José Valim, Creator of Elixir

edited by Lynn Beighley

THE LITTLE

Elixir & OTP GUIDEBOOK

Elixir IN ACTION

Sasa Jurić

MANNING

O'REILLY

Introducing Elixir

GETTING STARTED IN FUNCTIONAL PROGRAMMING

Simon St.Laurent & J.David Eisenberg

Built and designed at

plataformatec

consulting and software engineering

plataformatec

consulting and software engineering

Elixir coaching

Elixir design review

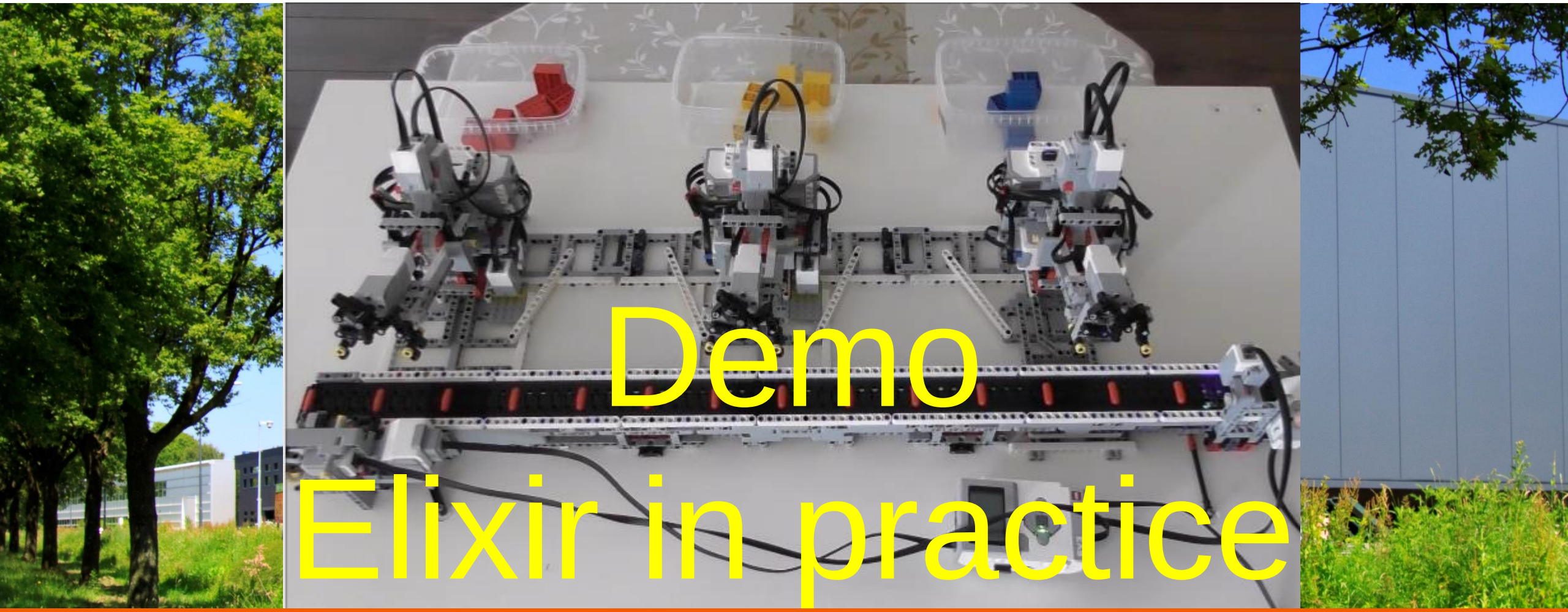
Custom development

elixir

@elixirlang / elixir-lang.org

Hot-or-Not Elixir
with José Valim

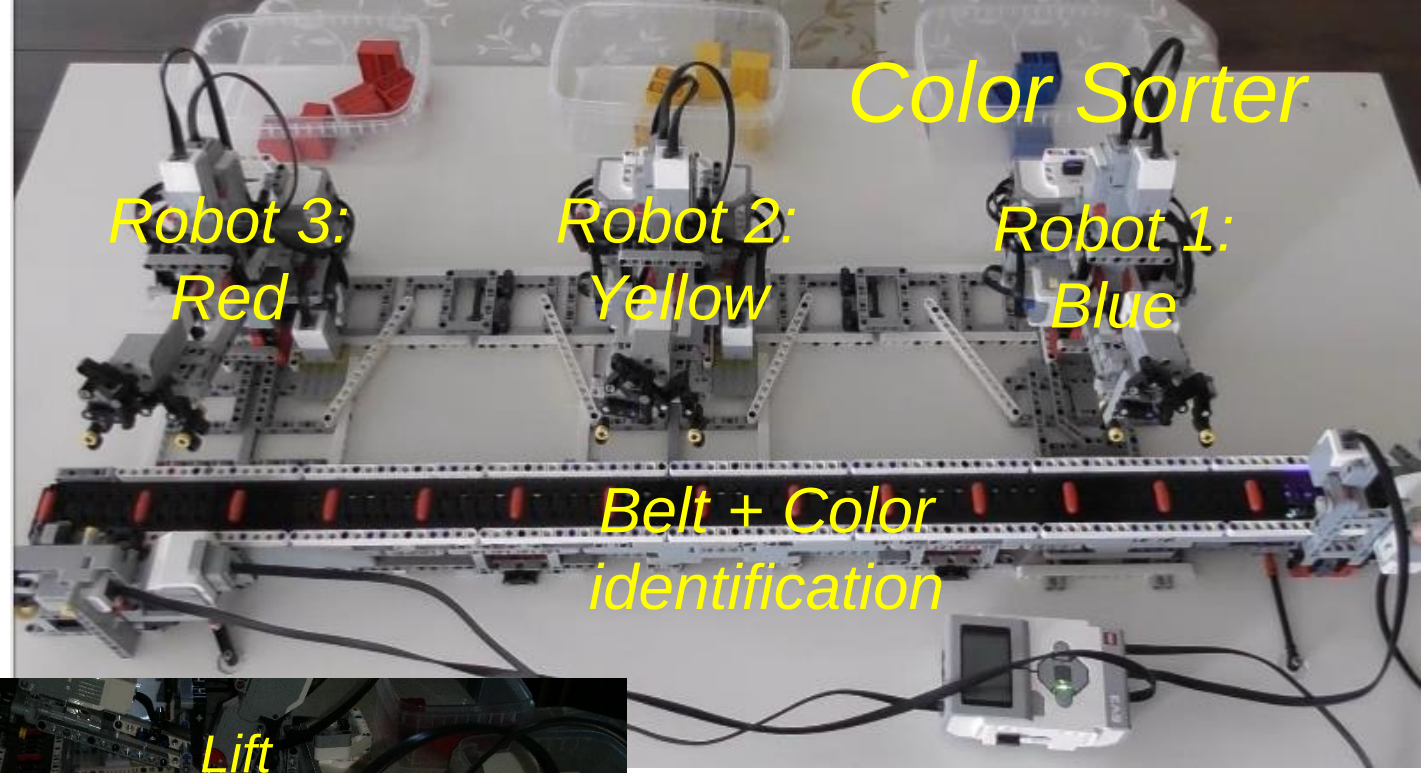
SOURCE OF YOUR TECHNOLOGY

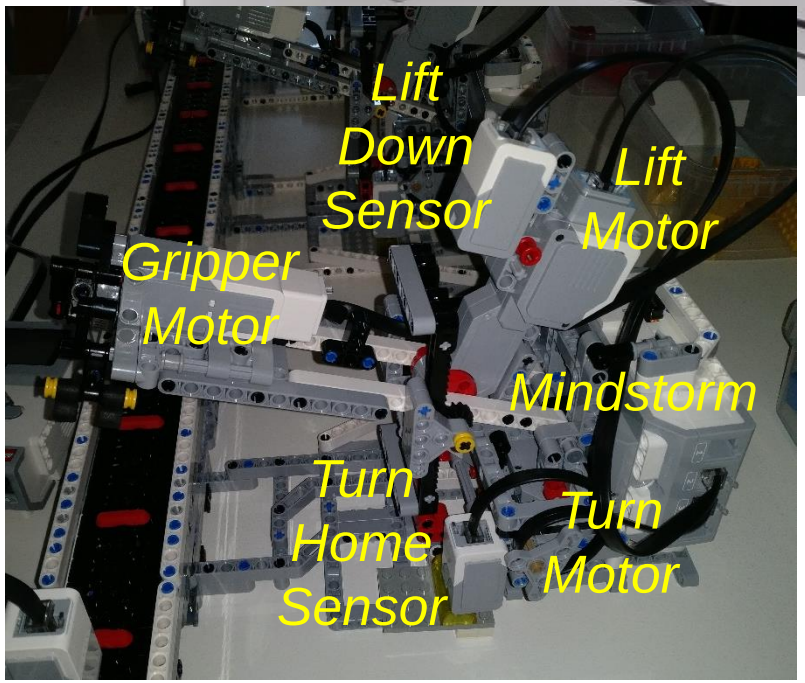


Elixir in practice

Goals:

- Learn Elixir
- Apply Elixir in embedded system
- Concurrent behavior
- Distributed application
- Complex control
- Test support
- Documentation
- (Failure recovery)

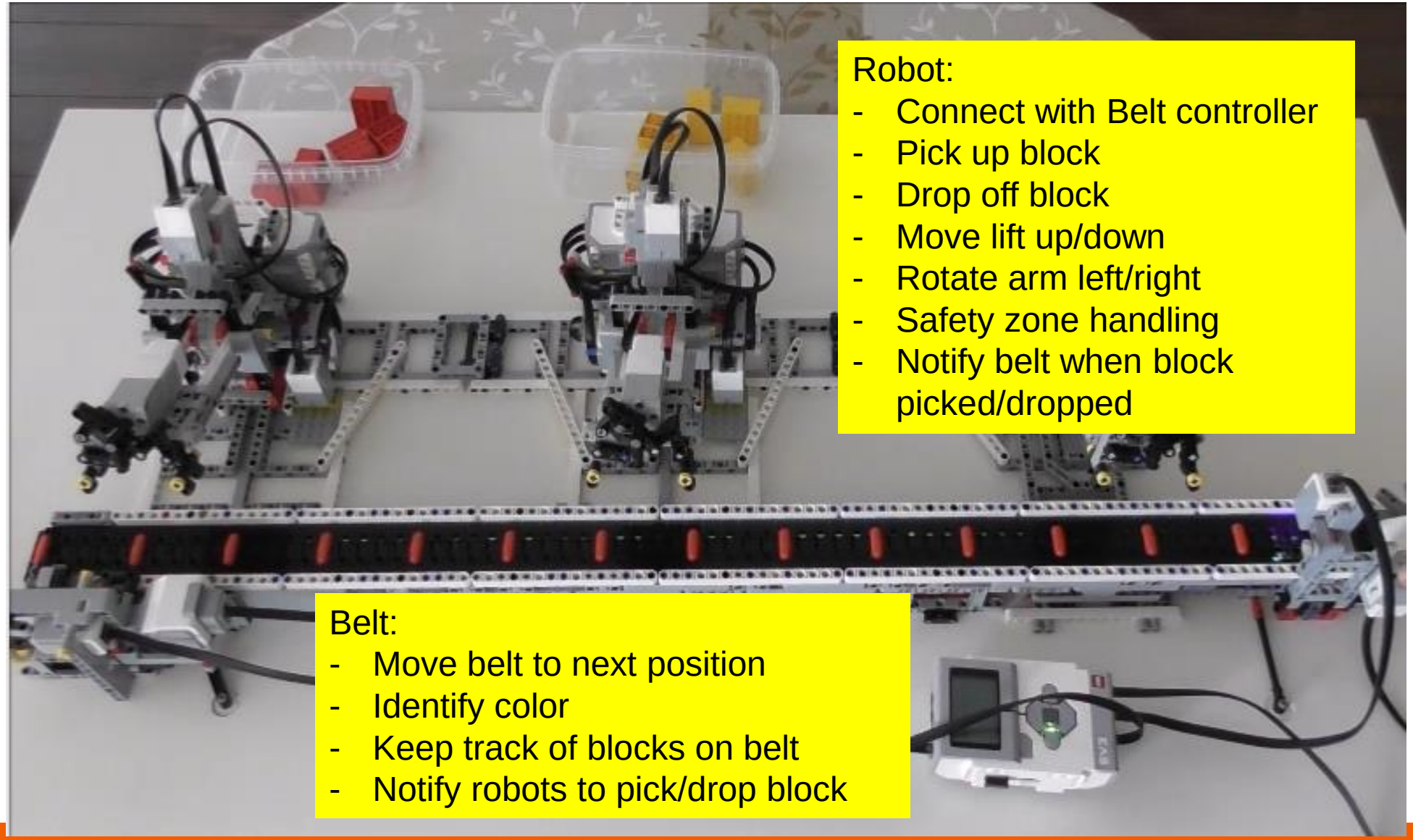




Team:

- Koen Rutten
- Jochem Berndsen
- Han van Venrooij
- Philippe Dirkse
- Paul Zenden

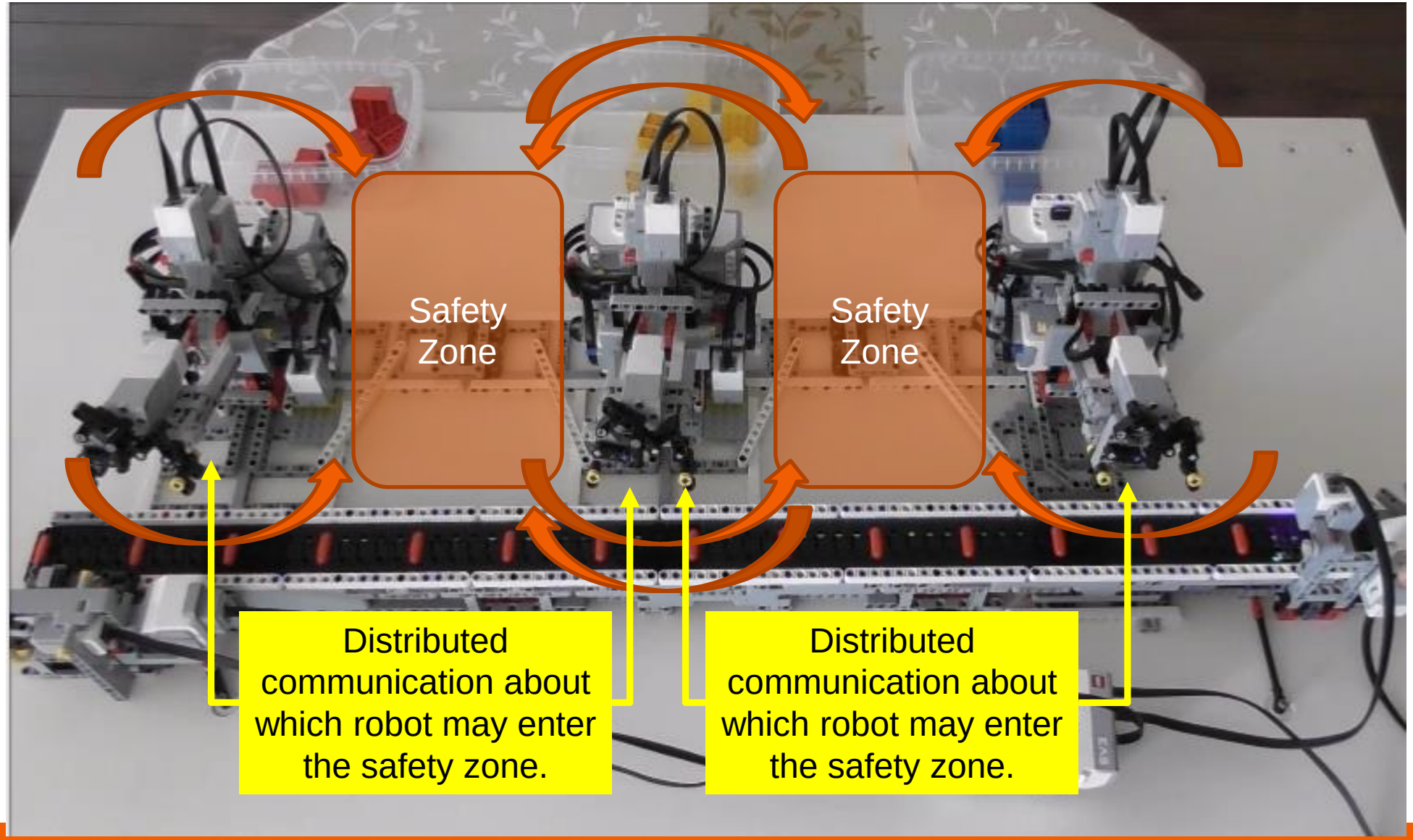
Functions



Robot:
- Connect with Belt controller
- Pick up block
- Drop off block
- Move lift up/down
- Rotate arm left/right
- Safety zone handling
- Notify belt when block picked/dropped

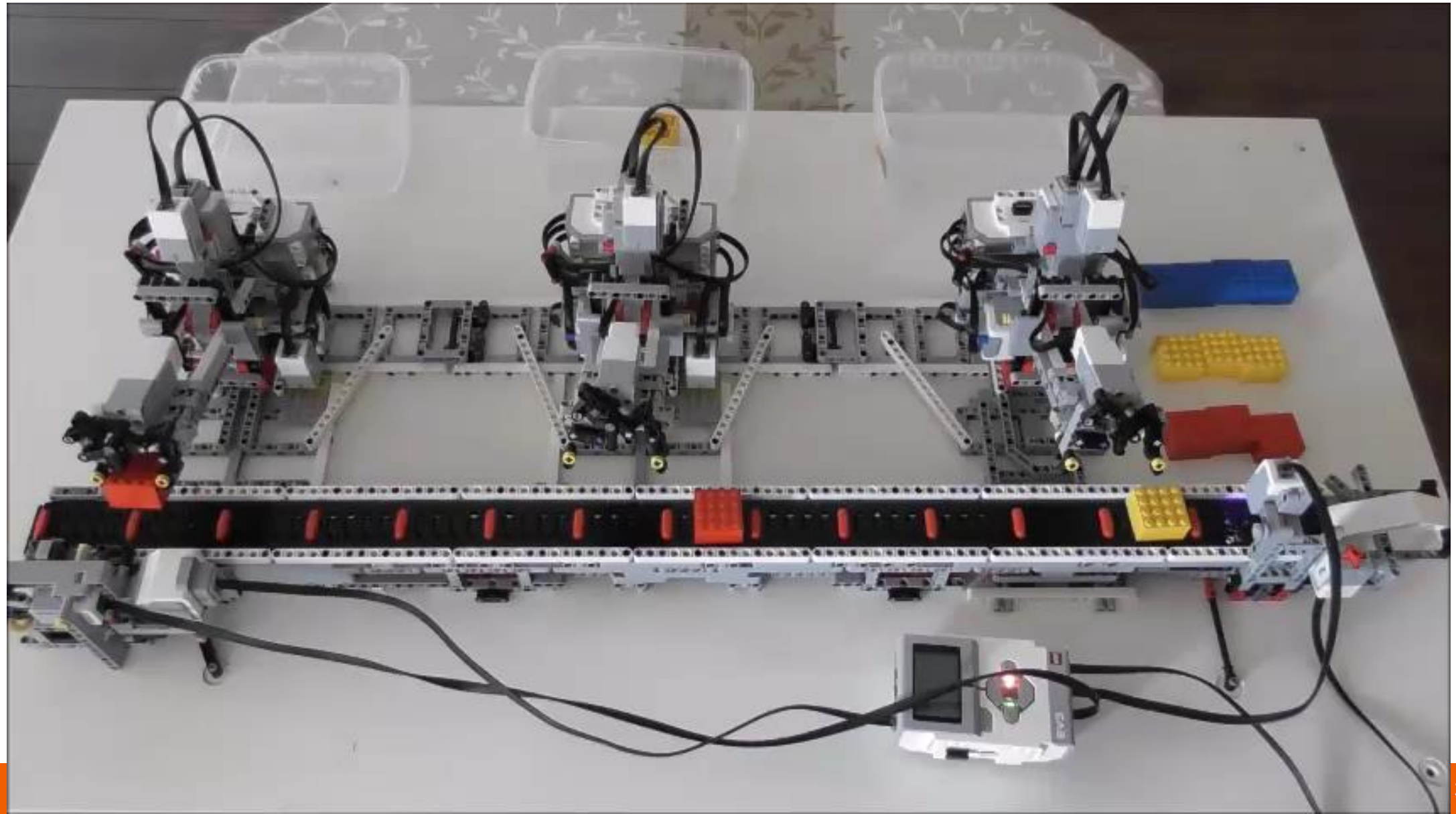
Belt:
- Move belt to next position
- Identify color
- Keep track of blocks on belt
- Notify robots to pick/drop block

Safety Layer – Limited Rotations



Allowed rotation

Demo first – Technicalities next



System levels

Elixir Application

Robot & Belt Application Logic
Robot & Belt Main Functions
Lego Device Handling: Derived from: https://github.com/jfcloutier/ev3/

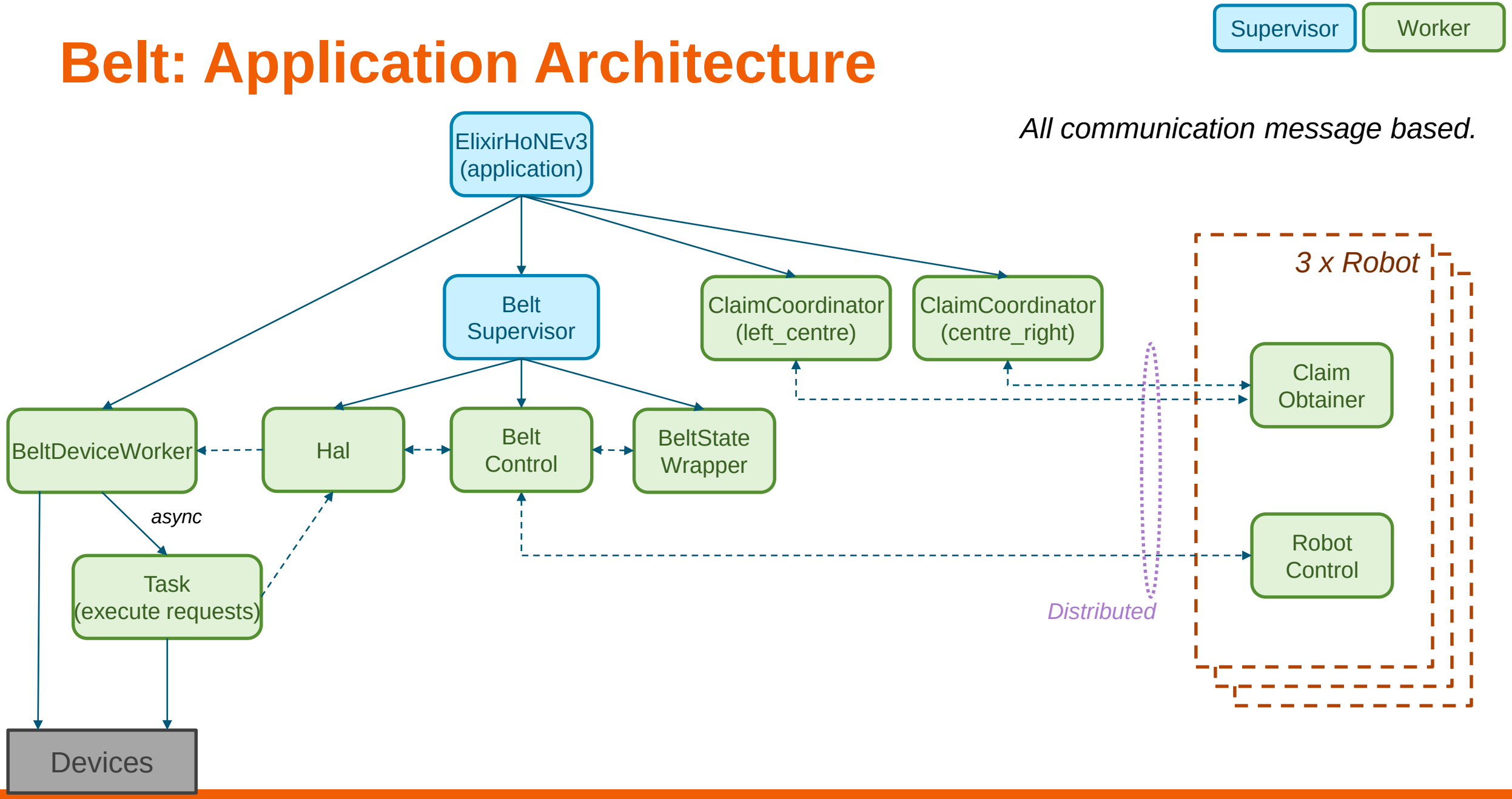
Nerves Framework

Frameworks: Networks, I/O, ev3dev, …
Tooling: Cross compilation tools for specific target
Platforms: cross compiled linux, boot directly to Erlang VM

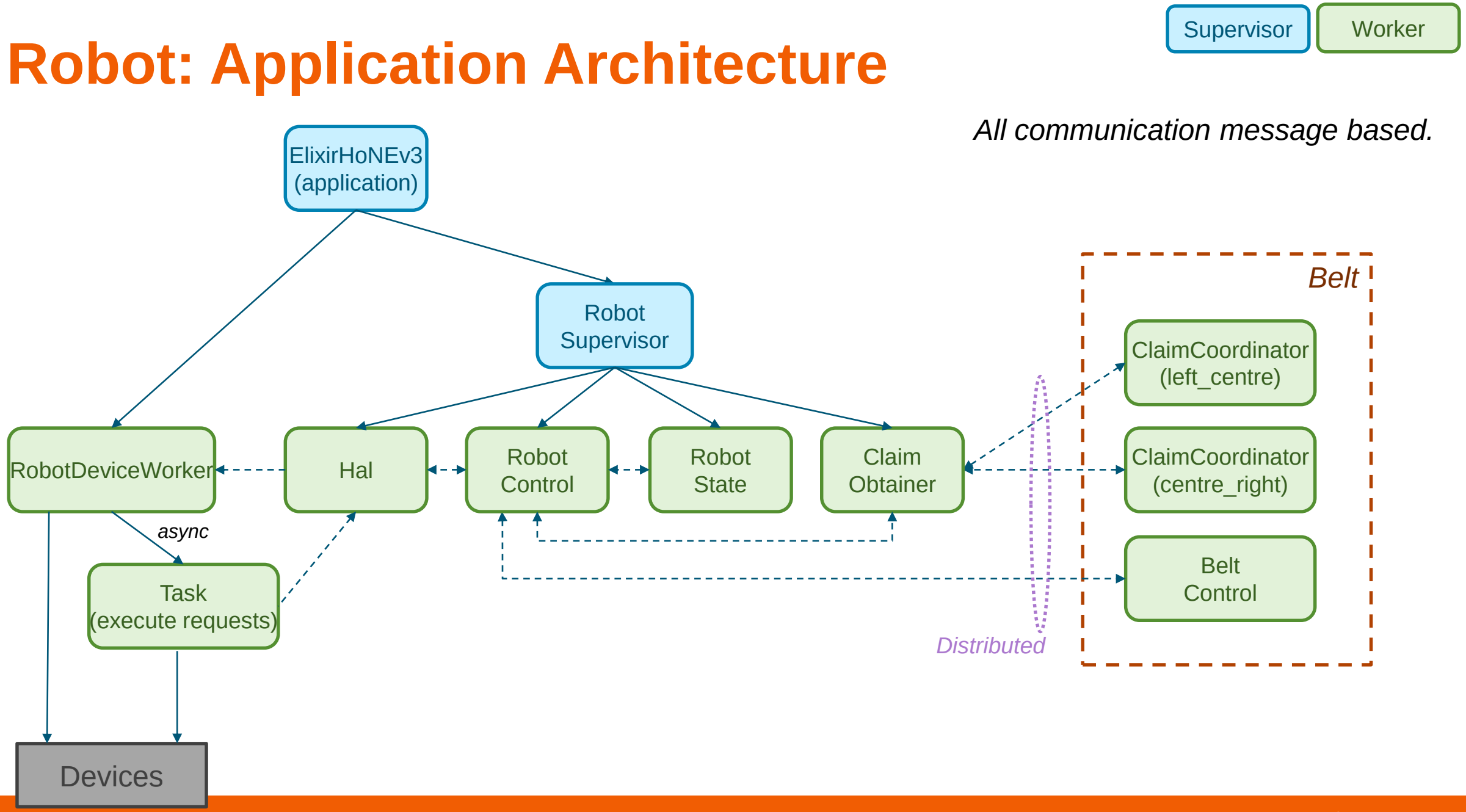
Lego Mindstorms Brick

300 MHz ARM926EJ-S
64 MB DRAM /16 MB Flash
MicroSD
Linux Kernel 4.4 – ev3dev patches
GPIO, I2C, SPI – ev3dev drivers
Ethernet/Wifi: USB dongle

Belt: Application Architecture

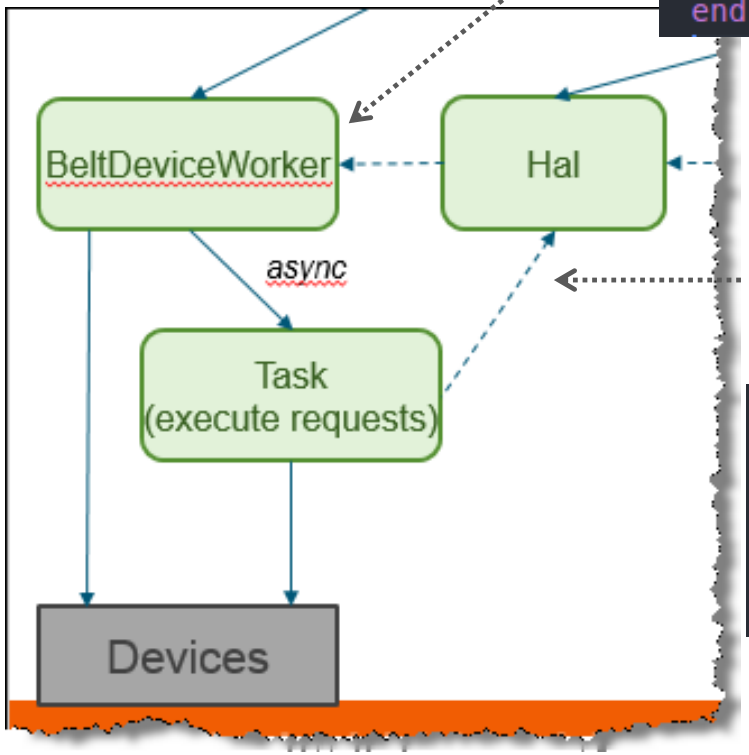


Robot: Application Architecture



Protocols

```elixir
defmodule HoneDevices.BeltDeviceBehaviour do
  @callback initialize(server_pid :: pid, client_pid :: pid) :: any
  @callback move_right(server_pid :: pid, client_pid :: pid, positions :: Integer) :: any
  @callback move_left(server_pid :: pid, client_pid :: pid, positions :: Integer) :: any
  @callback get_color(server_pid :: pid) :: :yellow | :red | :blue | :none | :other
  @callback get_status(server_pid :: pid) :: any
end
```



Defines client API
Compiler checks whether all functions are provided by BeltDeviceWorker

Hal will accept a notification message, but not explicitly defined (as above).

Task sends notification message directly.

Could be improved ☺

```elixir
defp notify_client(belt_state, client_pid, notification) do
  GenServer.cast(client_pid, {:ok, belt_state.worker_pid, notification})
  belt_state
end
```

Some Code

Starting application, belt and robot specific children

```
device_role = DeviceConfiguration.determine_device_role()

# Define workers and child supervisors to be supervised
children =
  device_workers(device_role)

# See http://elixir-lang.org/docs/stable/elixir/Supervisor.html
# for other strategies and supported options
opts = [strategy: :one_for_one, name: ElixirHoNEv3.Supervisor]
Supervisor.start_link(children, opts)
```

```
def determine_device_role() do
  roles = Application.get_env(:elixirHoN_ev3, :roles)
  {:ok, my_hostname} = :inet.gethostname
  my_hostname = to_string(my_hostname)
  my_role = roles[String.to_atom(my_hostname)]

  my_role
end
```

```
# Hot or not demo options
config :elixirHoN_ev3, :roles,
  "nerves-4b16": {:belt},
  "nerves-5370": {:robot, 1},
  "nerves-5783": {:robot, 2},
  "nerves-66a6": {:robot, 3}
```

```
defp device_workers({:belt}) do
  [worker(HoneDevices.BeltDeviceWorker, []),
   supervisor(Hone.Belt.Supervisor, [:"Belt Control"]),
   worker(Hone.Robot.ClaimCoordinator, [:left_centre, {:global, :left_centre_coordinator}], [id: :left_centre_coordinator]),
   worker(Hone.Robot.ClaimCoordinator, [:centre_right, {:global, :centre_right_coordinator}], [id: :centre_right_coordinator])
  ]
end
defp device_workers({:robot, id}) do
  [worker(HoneDevices.RobotDeviceWorker, [id]),
   supervisor(Hone.Robot.Supervisor, [translate_position(id), {:global, :belt}, :"Robot_#{inspect(translate_position(id))} Control"])
]
end
```

Some Code

Messages: Synchronous & Asynchronous

Note: still message exchange between processes.

Synchronous Messages: GenServer.call()
- to be handled by handle_call()

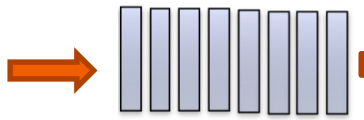
```
GenServer.call(server_pid, {:get_color})
```

"Direct call"

```
def handle_call({:get_color} = command, from, belt_state) do
  {color, belt_state} = BeltDeviceTasks.get_color(belt_state)
  {:reply, color, belt_state}
end
```

Asynchronous Messages: GenServer.cast()
- to be handled by handle_cast()

```
GenServer.cast(server, :initialise)
```



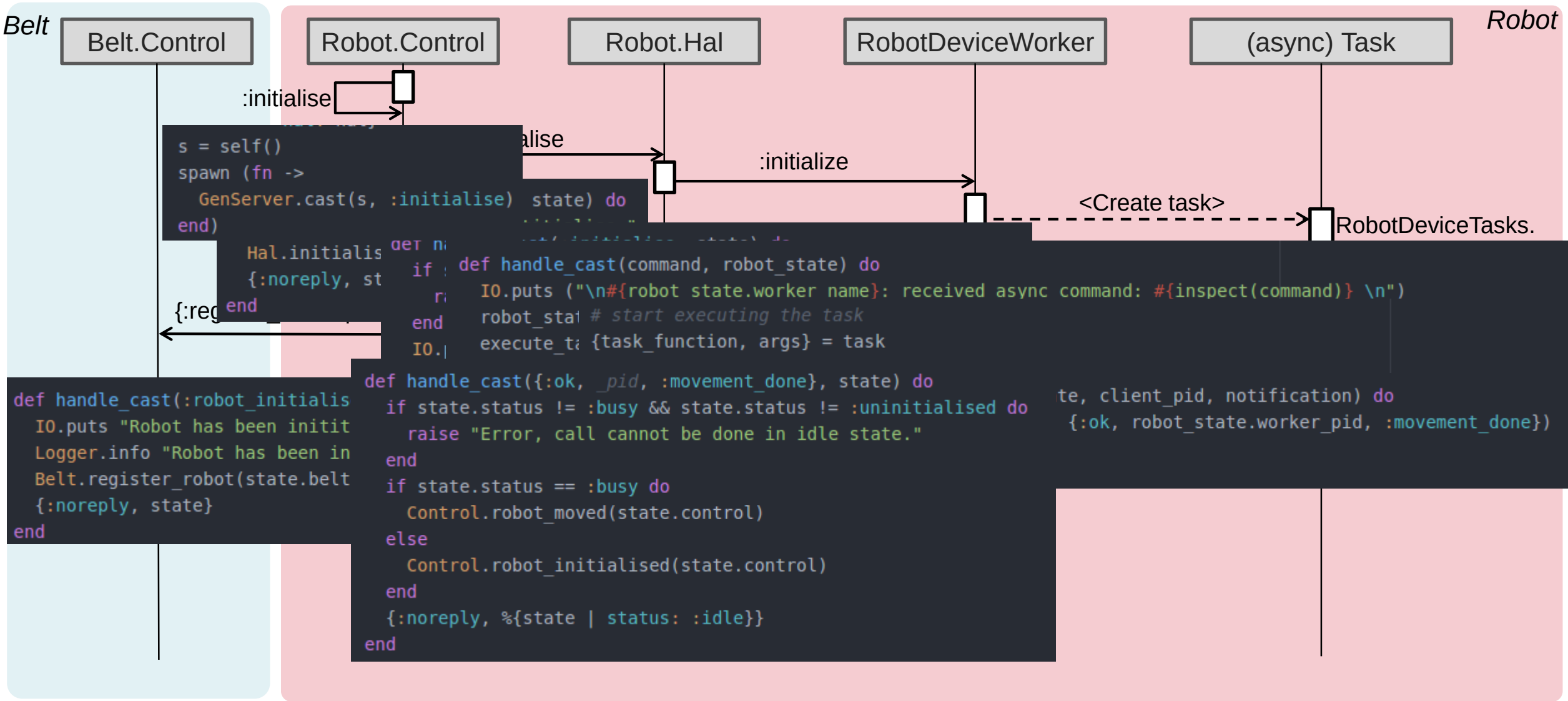
Async call

Msg Loop

```
def handle_cast(:initialise, state) do
  if state.status != :uninitialised do
    raise "Error, call cannot be done in initialised state."
  end
  IO.puts "Asking delegate to initialize."
  state.delegate.initialize(state.delegate, self())
  {:noreply, state}
end
```

Some Code

Robot initialization sequence



Some code - Pattern matching

```
def check_stop_button(belt_state) do
  {stop_sensor_state, _} = Ev3Lego.TouchSensor.read(belt_state.stop_sensor, :touch)
  belt_state = %BeltDeviceState{ belt_state | stop_sensor_state_previous: belt_state.stop_sensor_state_current }
  belt_state = %BeltDeviceState{ belt_state | stop_sensor_state_current:  stop_sensor_state }

  belt_state = interpret_stop_button_state( belt_state)
  belt_state
end

defp interpret_stop_button_state(
     %BeltDeviceState{
       stop_sensor_state_current: :pressed,
       stop_sensor_state_previous: :released} = belt_state), do: notify_client(belt_state, belt_state.client_pid, :stop_pressed)
defp interpret_stop_button_state(belt_state),  do: belt_state #Otherwise ignore
```

Some code

Belt state

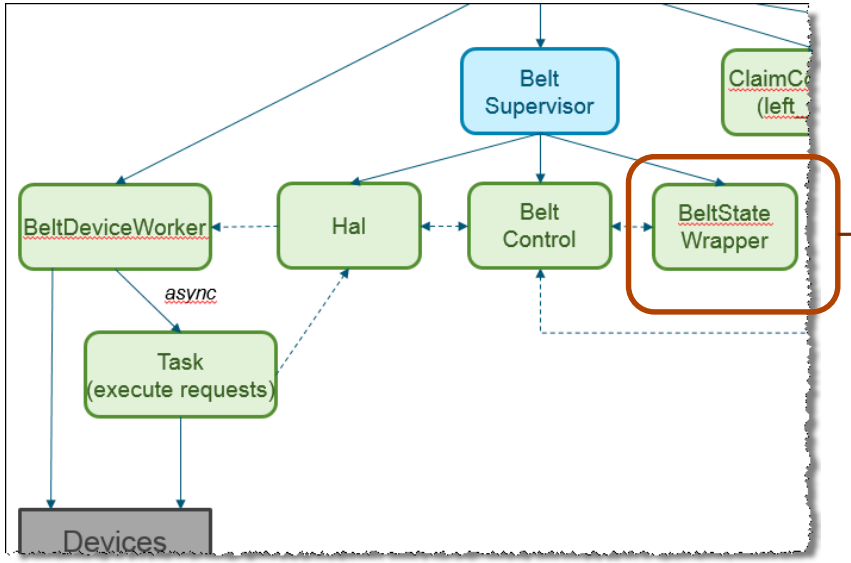
```
def move_safe?(state) do
  safe?(move_left!(state))
end
```

```
defp move_left!(state) do
  new_blocks = map(state.blocks, fn {pos, block} -> {pos - 1, block} end)
  %{state | blocks: new_blocks}
end
```

```
defp safe?(state) do
  all?(state.blocks, fn {pos, block} -> safe_pos?(state, block, pos) end)
end
```

```
defp safe_pos?(state, block, pos) do
  config = state.config
  safe_pos = config.safe_min <= pos && pos <= config.safe_max
  assigned_to = Map.get(state.block_to_robot, block)
  beyond_assignment = case assigned_to do
    nil -> pos < config.picks.left || pos < config.picks.centre || pos < config.picks.right
    robot -> pos < config.picks[robot]
  end
  safe_pos && !beyond_assignment
end
```

Testing



Tested by

```
test "moving to the right moves the blocks", %{state: state} do
  {_, state} = BeltState.with_block!(state, :red)
  state = BeltState.move!(state)
  assert nil == BeltState.colour_at(state, @config.sensor)
  assert :red == BeltState.colour_at(state, @config.sensor-1)

  state = BeltState.move!(state)
  assert nil == BeltState.colour_at(state, @config.sensor)
  assert nil == BeltState.colour_at(state, @config.sensor-1)
  assert :red == BeltState.colour_at(state, @config.sensor-2)
end

test "empty belt can safely be moved", %{state: state} do
  assert BeltState.move_safe?(state)
  state = BeltState.move!(state)
  assert BeltState.move_safe?(state)
end

test "unassigned blocks cannot move beyond first pickup position", %{state: state} do
  {_, state} = BeltState.with_block!(state, :red)

  assert BeltState.move_safe?(state)
  state = BeltState.move!(state)

  assert BeltState.move_safe?(state)
  state = BeltState.move!(state)

  assert !BeltState.move_safe?(state)
end
```

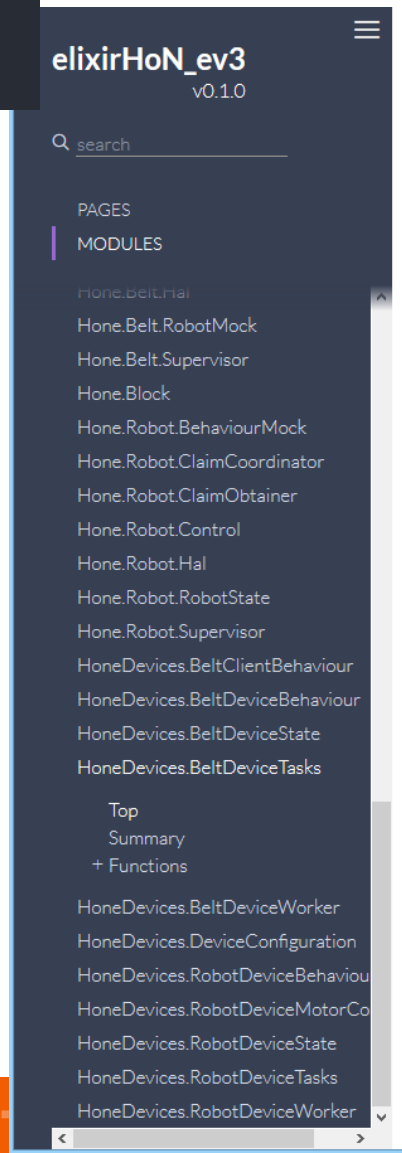
Documentation

```elixir
defmodule  HoneDevices.BeltDeviceTasks do
  @moduledoc "The individual tasks the belt device can perform"
```

```elixir
  @doc """
  Initialize all devices; moves the belt to home position.
  """
  def initialize(belt_state, client_pid) do
```

```
zendenp@ubuntu:~/work/elixir_hon/HoNElixer/Src/elixirHoN_ev3$ mix docs
Env
  MIX_TARGET:   sioux_system_ev3
  MIX_ENV:      dev

Docs successfully generated.
View them at "doc/index.html".
```



HoneDevices.BeltDeviceTasks

The individual tasks the belt device can perform

Summary

Functions

`check_stop_button(belt_state)`
Check whether the stop button is pressed. If so, send a message to the client

`get_color(belt_state)`
Read the current value of the color sensor

`init(belt_state)`
Initialise at start of the worker

`initialize(belt_state, client_pid)`
Initialize all devices; moves the belt to home position

`move_left(belt_state, client_pid, positions)`
Move the belt positions to the left (from the color sensor away)

`move_right(belt_state, client_pid, positions)`
Move the belt positions to the right (towards the color sensor)

Functions

> **`check_stop_button(belt_state)`**

Check whether the stop button is pressed. If so, send a message to the client.

Development, build & deployment

- Linux only (due to cross-compilation) if using nerves
- Nerves toolchain, commands: powerful
  - Mix compile, mix firmware, mix firmware.burn
  - Buildroot linux kernel configuration
  - Busybox user space commands
- Documentation somewhat scattered
  - Searching, trial & error, active slack channel, github example code
- Possible to upload firmware via network (but did not try it)
  - https://github.com/nerves-project/nerves_firmware_http
  - https://github.com/nerves-project/nerves_firmware

Our Experiences - Benefits

- Easy to pick up
- REPL (read-evaluate-print loop), Elixir toolset (mix)
- Syntax
- High fun factor (powerful)
  - Focus on essence, less on technical details
- Provides a "fighting chance" to develop a proper distributed, fault-tolerant system
  - Need to develop/learn best practices/patterns
- Vibrant community: tooling, libraries and frameworks abound
  - nerves, mix, Phoenix, Ecto, …
  - Erlang libraries
- Erlang VM/OTP: Great!

Our Experiences - Concerns

- No mandatory type system; critical for any non-trivial project
  - Defining types and type specifications is optional (Need guidelines, best practices)
  - Dialyzer might provide sufficient support
    - Not clear whether more complex type specifications are supported
- Decomposition design of components (supervisors, workers), including protocol (messages) is important
- IDE support
  - Partly inherent for dynamic languages
  - Different level of support for different kind of general editors (emacs, atom, IntelliJ, Sublime, …)
- Performance not sure; did no measurements
  - Calculation intensive might be slow.
  - IoT, distributed systems probably OK.
  - Can call C/C++ functions if needed

Our Experiences - Conclusion

- Great learning experience & fun

- Elixir/Erlang looks interesting, especially distributed systems, but you still have to think about:
  - Difficult aspects like fault-tolerance, network stability, security, etc.
  - Decomposition and interface protocol design:
    - messages should result in atomic actions

- Would we apply it in a new project?
  - Depends highly on the nature of project.

Active development – vibrant community

- Many interesting new applications and developments
  - Auto-connecting devices, machine learning, game back-end, web services, drones, …
  - Search for ElixirConf EU 2017, ElixirDaze 2017, Lambda Days 2017, more …
  - Look at:
    - https://elixir-lang.org/
    - https://elixirforum.com/
    - http://elixir.community/
    - https://www.erlang-solutions.com/
    - http://nerves-project.org/
    - http://www.phoenixframework.org/
    - https://hex.pm/

Elixir - Hot-or-Not?

Thank you, José for Elixir

More Hot-or-Not, more Sioux

Q3 2017 > Hot-or-Not “Accelerating Intelligence”

Q4 2017 > Hot-or-Not The Next Generation (workshop)

Go to www.sioux.eu for more information.


thanks


DRINKS

Source of your technology