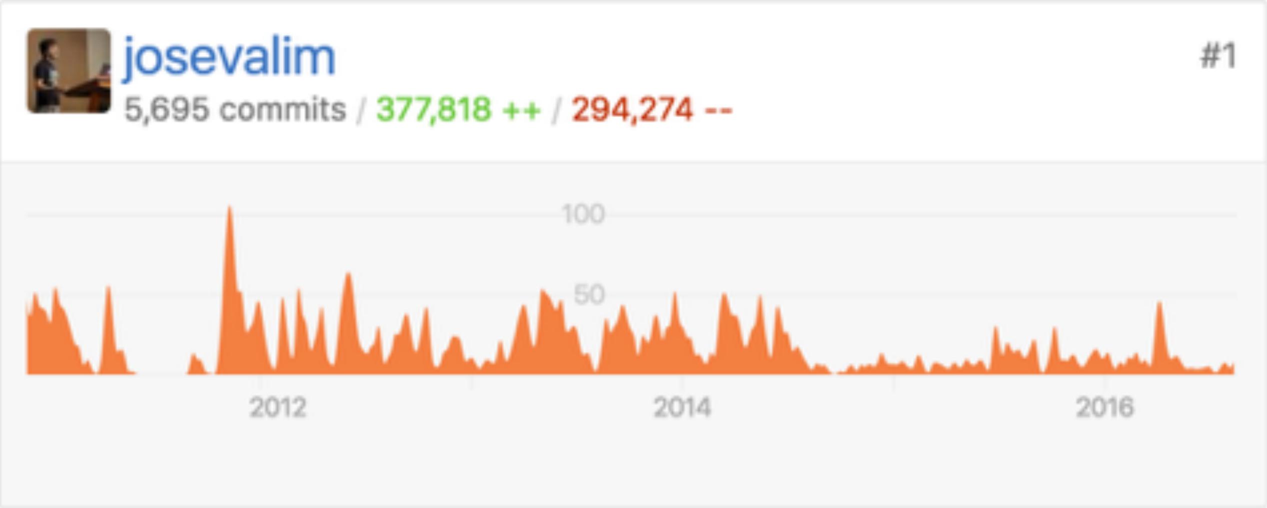
Building Scalable Applications with Elixir

Alexei Sholik

Kyiv Elixir Meetup, 1 Oct 2016

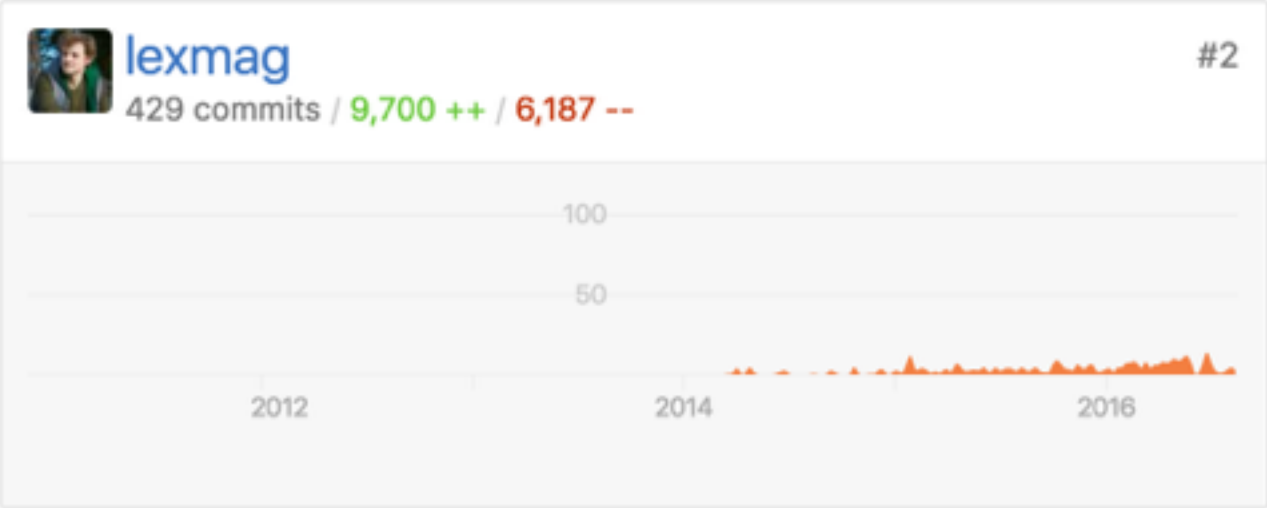
About me

Familiar with Elixir since 2012



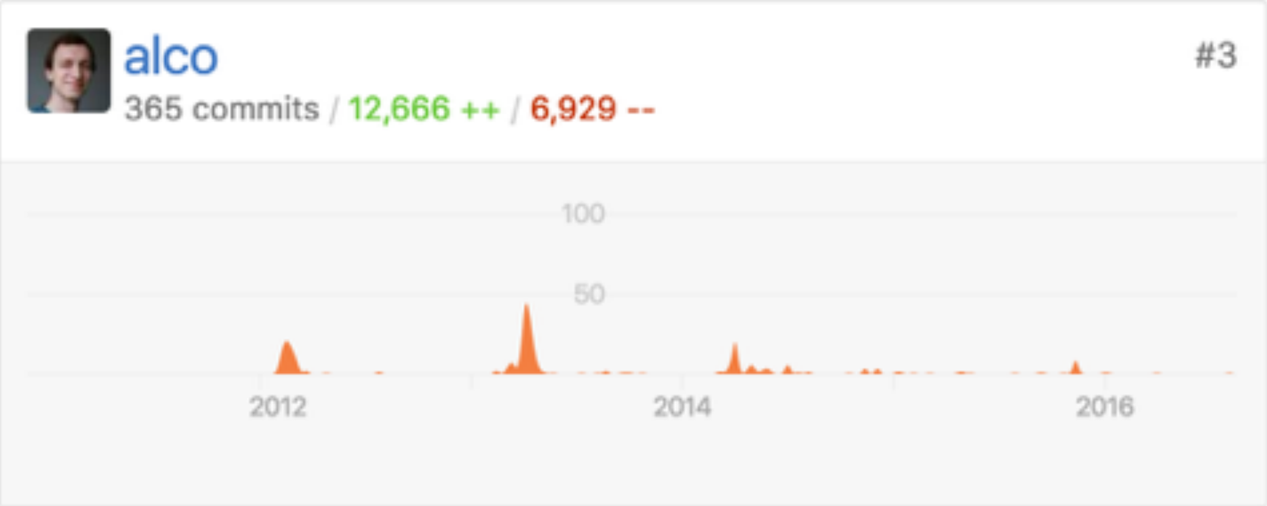
josevalim #1

5,695 commits / 377,818 ++ / 294,274 --



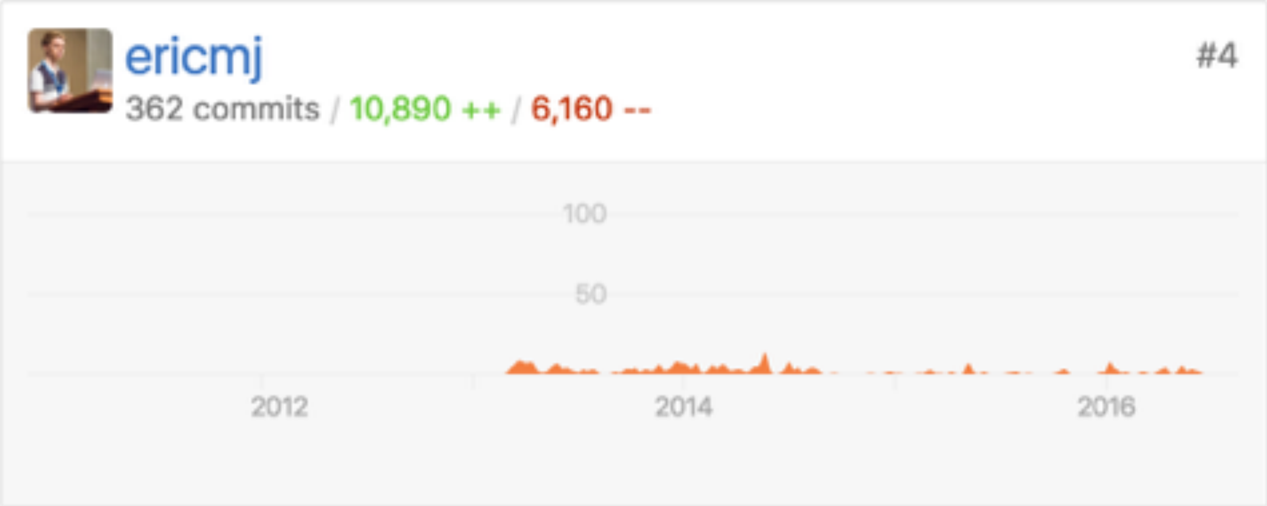
lexmag #2

429 commits / 9,700 ++ / 6,187 --



alco #3

365 commits / 12,666 ++ / 6,929 --

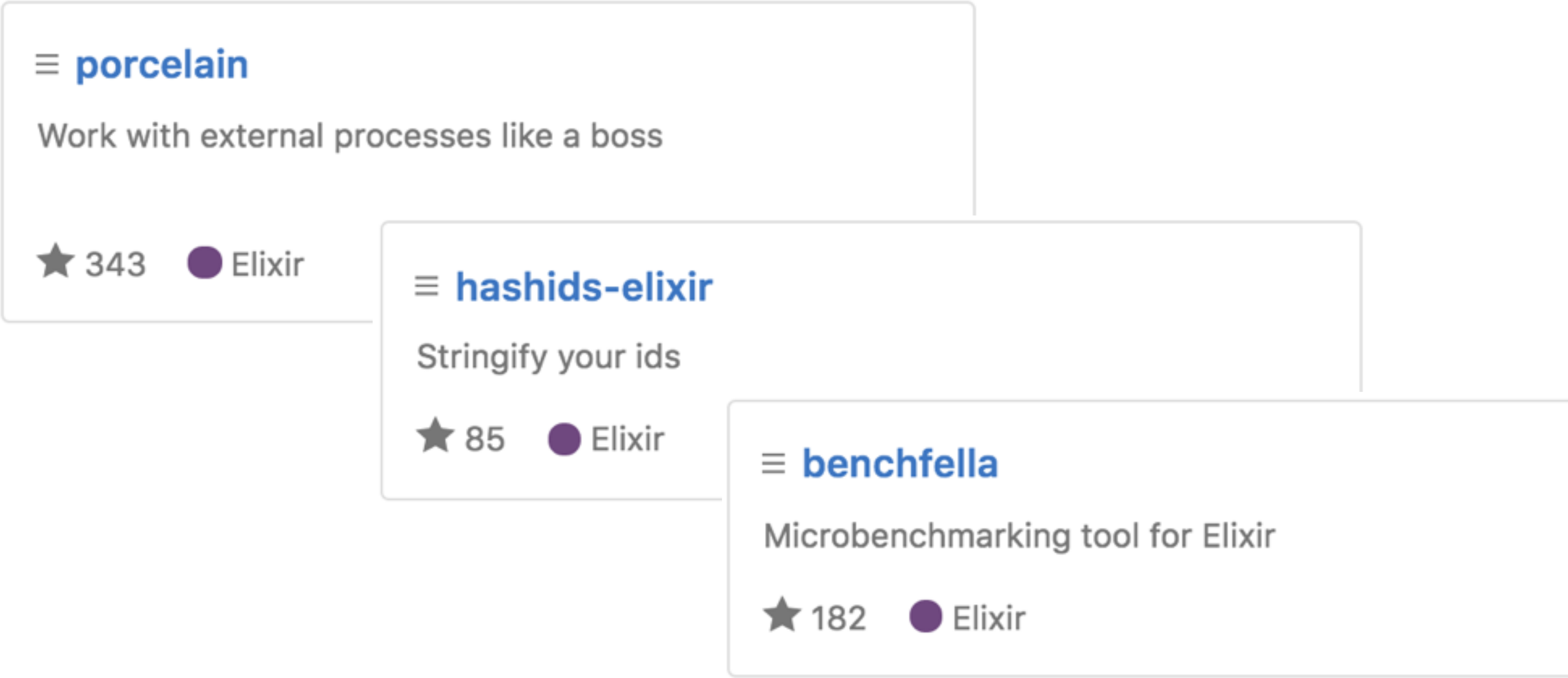


ericmj #4

362 commits / 10,890 ++ / 6,160 --

About me

I love contributing to open source



≡ porcelain

Work with external processes like a boss

★ 343 ● Elixir

≡ hashids-elixir

Stringify your ids

★ 85 ● Elixir

≡ benchfella

Microbenchmarking tool for Elixir

★ 182 ● Elixir

About me

2014–present

PSPDFKit

The Leading PDF Framework for iOS, Android and Web

About me

2014–present

Annotations, everywhere, in real time.

https://pspdfkit.com/instant

Language: elixir

Team:

Me

🙏

Martin Schürrer

🙏

Built to scale (hopefully)

What does being scalable mean?

A scalable system

efficiently uses all available hardware resources

A scalable system

is able to handle increased loads when new hardware resources are added (horizontal/vertical scaling)

A scalable system

can withstand prolonged overload, gracefully degrading the quality of service

Motivation

We want our application to keep running and make our customers happy

If it becomes slow or unusable when faced with increased load, we have a problem

Designing the application in a way that makes it scalable will help alleviate the problem

Why Elixir?

Why Elixir?

This is an Elixir meetup

Why Elixir?

Built on top of Erlang/OTP

Why Elixir?

Known for developer productivity, great performance and maintainability

The scope

- A single-node system
- Running a bunch of OTP applications
- Using external services like DB, 3rd-party APIs, etc.

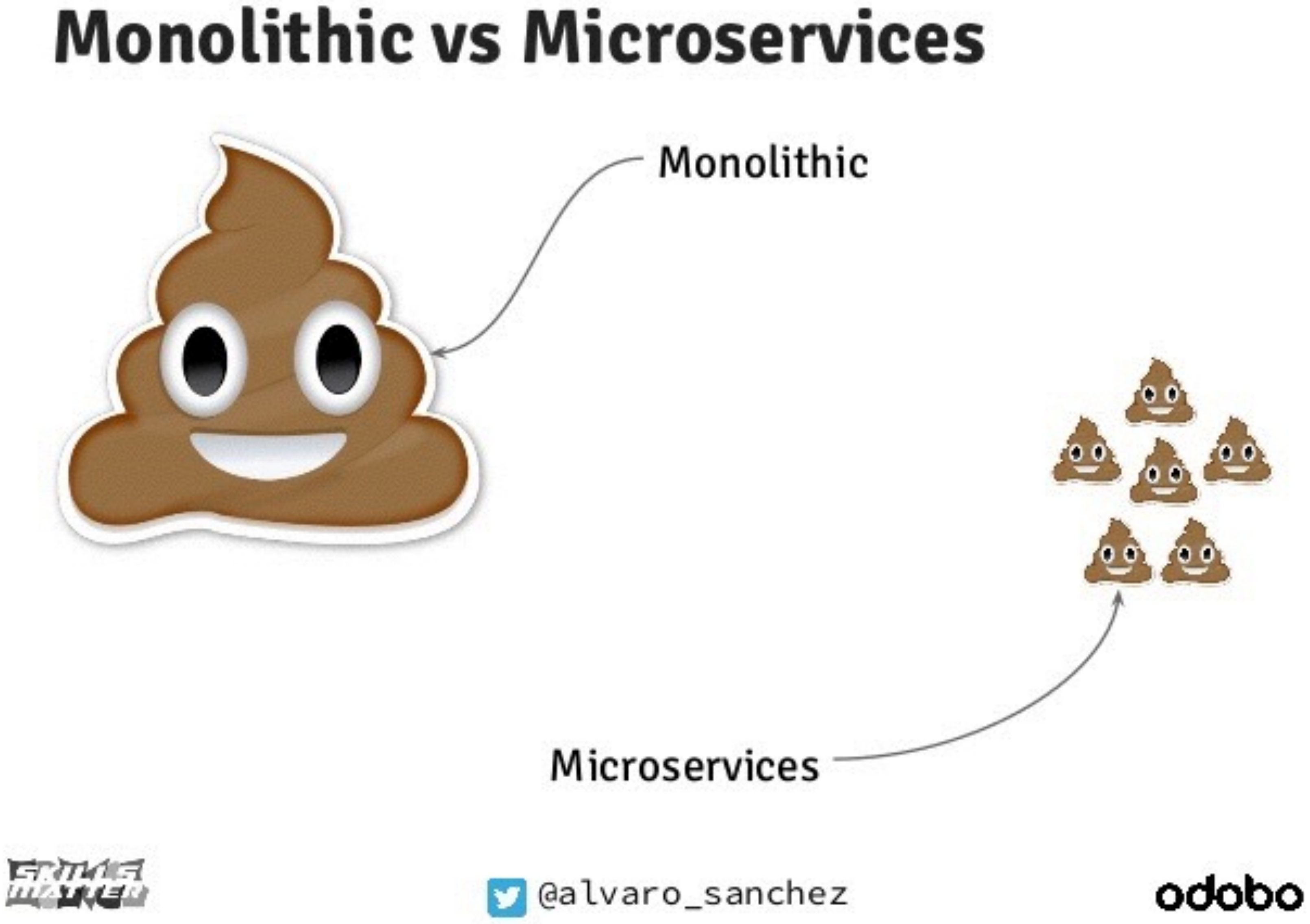
We will not talk about

HORIZONTAL

VERTICAL

scaling

Nor about



Monolithic vs Microservices

Monolithic

Microservices

SKILLS MATTER

@alvaro_sanchez

odobo

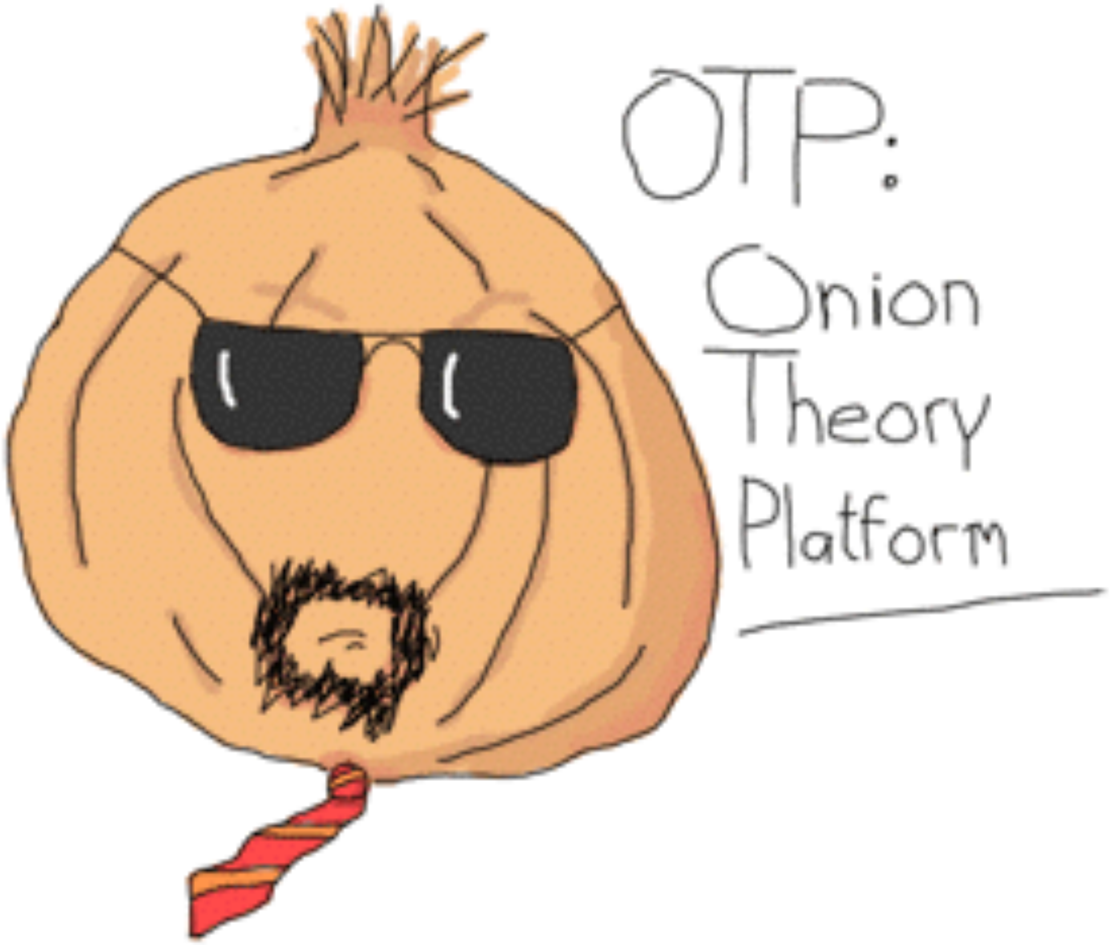
Definitely not about

(maybe next time)

<interlude>

Let's talk about Erlang/OTP

(or the starting point for every Elixir and Erlang system ever)



OTP: the good parts

Applications

OTP: the good parts

Behaviours
(supervisor, gen_server)

OTP: the good parts

Error handling
(error_logger, sasl)

OTP: the good parts

Tools for debugging and tracing (dbg, fprof, observer)

OTP: the good parts

ETS
(Erlang Term Storage)

But there's more...

Beyond OTP

- Logger
- Task
- Whatsapp's gen_factory and gen_industry
- Process pools, circuit breakers, job schedulers, etc.

Take-away:

start with OTP but don't let it limit your options

</interlude>

Principles of scalable design

1
Measure and observe

Metrics

- Identify points of interest: DB, caches, different kinds of workers, tasks, etc.
- What to measure: processing time, latency, error rates, success rates, number of requests, etc.

Metrics: tools

- Exometer + DataDog
- Telegraf + InfluxDB + Grafana[1]

[1] http://tech.footballaddicts.com/blog/gathering-metrics-in-elixir-applications

Logging: what to log

- unexpected or rare results
- info about requests, SQL queries, connected clients
- interesting events

Logging protip

Properly configure Logger metadata

```
config :logger, :console,
  format: "\n$time $metadata[$level] $levelpad$message\n"
  metadata: [:user_id]
```

Logging protip

Properly configure Logger metadata

```
config :logger, :console,
  format: "\n$time $metadata[$level] $levelpad$message\n"
  metadata: [
    :user_id, :application, :module, :function, :file, :line
  ]
```

Logging protip

Properly configure Logger metadata

```
config :logger, :console,
  format: "\n$time $metadata[$level] $levelpad$message\n"
  metadata: [
    :user_id, :application, :module, :function, :file, :line
  ]

Logger.info "Something interesting", user_id: 1
```

:observer

- Helpful at the development stage but can also attach to a system running in production environment

- See http://www.phoenixframework.org/blog/the-road-to-2-million-websocket-connections for an example

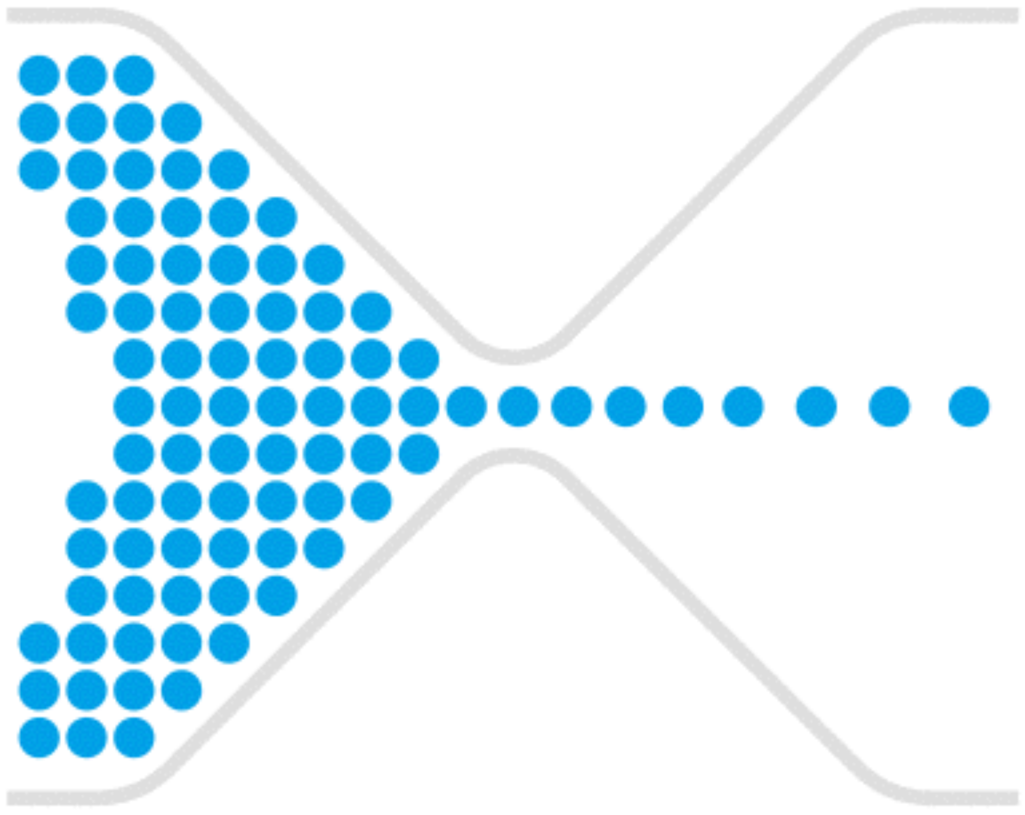
2
Identify and prevent bottlenecks

Typical bottlenecks

- A single process receiving more requests than it can actually process in a given time interval
- A supervisor process receiving too many requests to start new children
- An event handler broadcasting messages to many processes sequentially
- File I/O, 3rd-party APIs, logging overflow, etc.

Identifying bottlenecks

Basically, look for things that look like this



Identifying bottlenecks

On a serious note, measure and observe

Example 1

Problem: supervisor can't keep up with requests to start more children

Example 1

Solution: add more supervisors!

https://github.com/basho/sidejob

Example 2

Problem: broadcasting messages to many processes is too slow

Example 2

Solution: send messages in chunks, in parallel

Dealing with bottlenecks

- caching
- splitting work between multiple processes
- in general, it depends...

Caching protip

Use a cache with locking,
e.g. ConCache

```
ConCache.get_or_store(:my_cache, key, fn ->
  heavy_computation()
end)
```

Process pools

When a single process is not enough

Process pools

- reduce response times under normal load (by removing the startup delay)
- define an upper bound on the number of simultaneously allowed consumers
- may keep intermediate state between invocations

Examples of pools

- pool of DB connections
- socket acceptor pool
- pool of processes communicating with an external service

Process pools: tools

- poolboy
- sbroker

3
Introduce limits to the components that are likely to cause contention

Ecto has limits

```
config :app, Ecto.Repo,
  adapter: Ecto.Adapters.Postgres,
  pool_size: 100,
  timeout: :infinity,
```

Logger has limits

- `:sync_threshold`
- `:discard_threshold_for_error_logger`

3rd-party APIs have limits

HTTP/1.1 503 Service Unavailable

Your machine has limits

Choose realistic limits

based on load guesstimates

or

results of load testing

(better)

4
Choose a strategy for managing overload

Back pressure

limit the rate of incoming requests
over a single channel

Examples of back pressure

- `:sync_threshold` in Logger configuration
- TCP and the buffer bloat fiasco
- rate limiting

Load shedding

drop pending or incoming requests before they are processed

Breaking the circuit

protect the application from being overflowed with failures

A basic circuit breaker

https://github.com/jlouis/fuse

```
strategy = {:standard, max_restarts, max_time}
refresh = {:reset, 60000}
opts = {strategy, refresh}
:fuse.install(api_endpoint_fuse, opts)
```

```
case :fuse.ask(api_endpoint_fuse, :sync) do
  :ok -> HTTP.get(...)
  :blown -> {:error, :no_connection}
end
```

5
Test and refine

Use load testing to identify bottlenecks and weak points early on

Adjust your limits to achieve desired performance,
i.e. throughput and latency

Load testing: tools

- basho_bench
- ponos
- Tsung

Let's recap...

What we've learned

1. Measure and observe
2. Identify and prevent bottlenecks
3. Introduce limits around weak points
4. Choose a strategy for managing overload
5. Test and refine

Final step...

Watch it scale

Thank you!

Alexei Sholik

github.com/alco

twitter.com/true_droid

Image references

- https://electric-cloud.com/solutions/unlock-agile-bottleneck
- http://learnyousomeerlang.com/building-applications-with-otp
- http://www.gettyimages.com/detail/photo/burning-rack-of-network-servers-high-res-stock-photography/85925316
- https://twitter.com/kellan/status/378167190505017344