# Discover the potential of Elixir

Real world use case

# Hello

Damián Le Nouaille-Diez

- @damln (twitter / github)
- damln.com
- Freelance Senior Developer (8+ years Ruby, Oouch...)

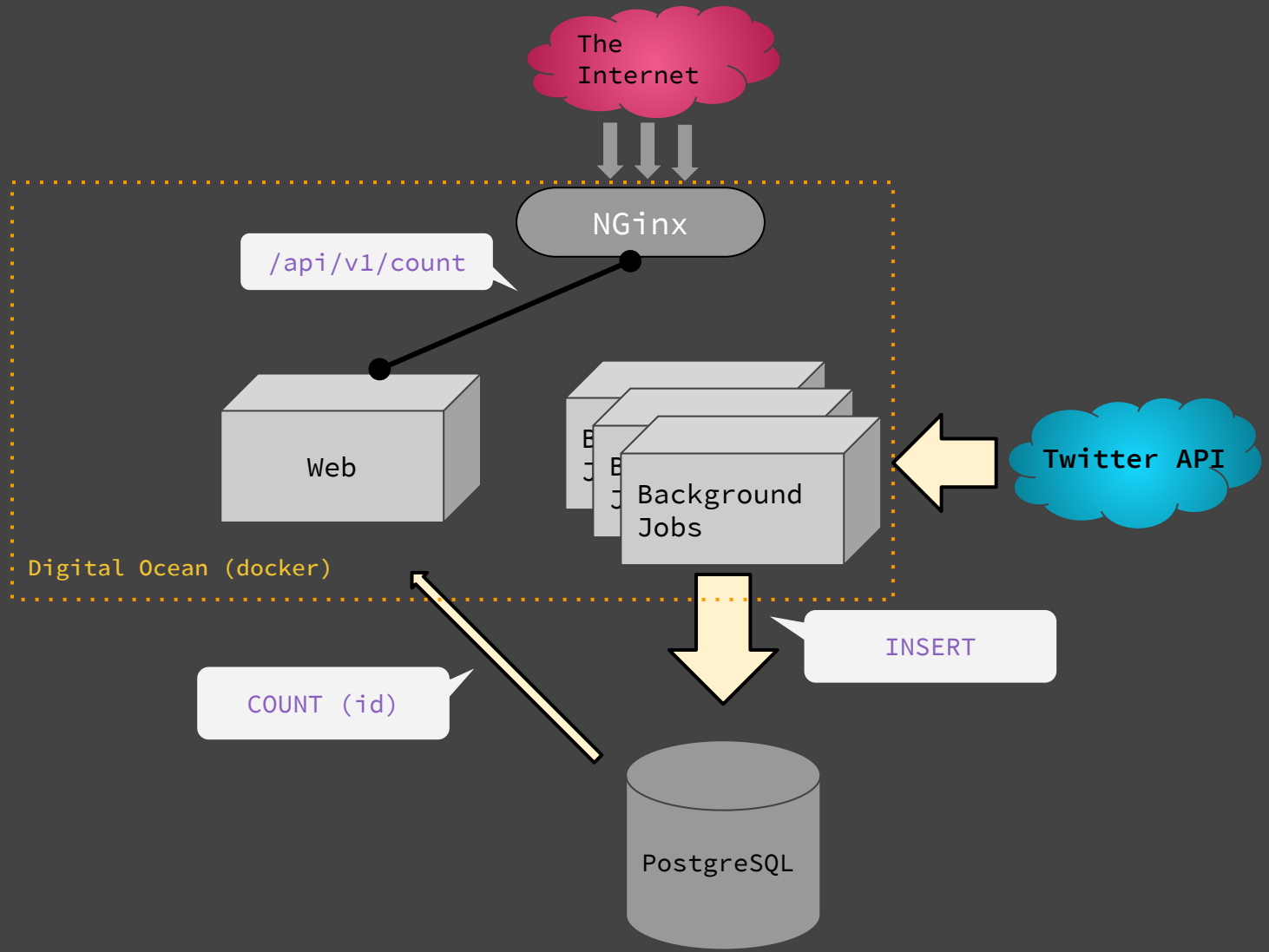
“Recently moved in BCN”

---

# Getting real

- What can I do in Elixir **today** (not in a future fantasy projects)
- HashtagBattle.com: Counting hashtags in real time.
- Realtime? Let's play with Phoenix Channels!

# Real world.


The Internet
NGinx
/api/v1/count
Web
Background Jobs
Twitter API
Digital Ocean (docker)
INSERT
COUNT (id)
PostgreSQL

# Goals

- Offer real time updates in the browser on every new tweet (current status: doing AJAX polling every 5 secs, not really real time).
- Resilient to network failures.

# First steps

- Creating an empty Phoenix application and make it "run" (mix phx.server).
- Create a Dockerfile to be able to deploy in my current server with existing tooling.
- Connect the Phoenix app to the PostgreSQL database (needed to read data, no write).

---

# Hypothesis

---

**Ecto is gonna be hard to setup.**

**Queries will be hard to write.**

“Just ship it.”

---

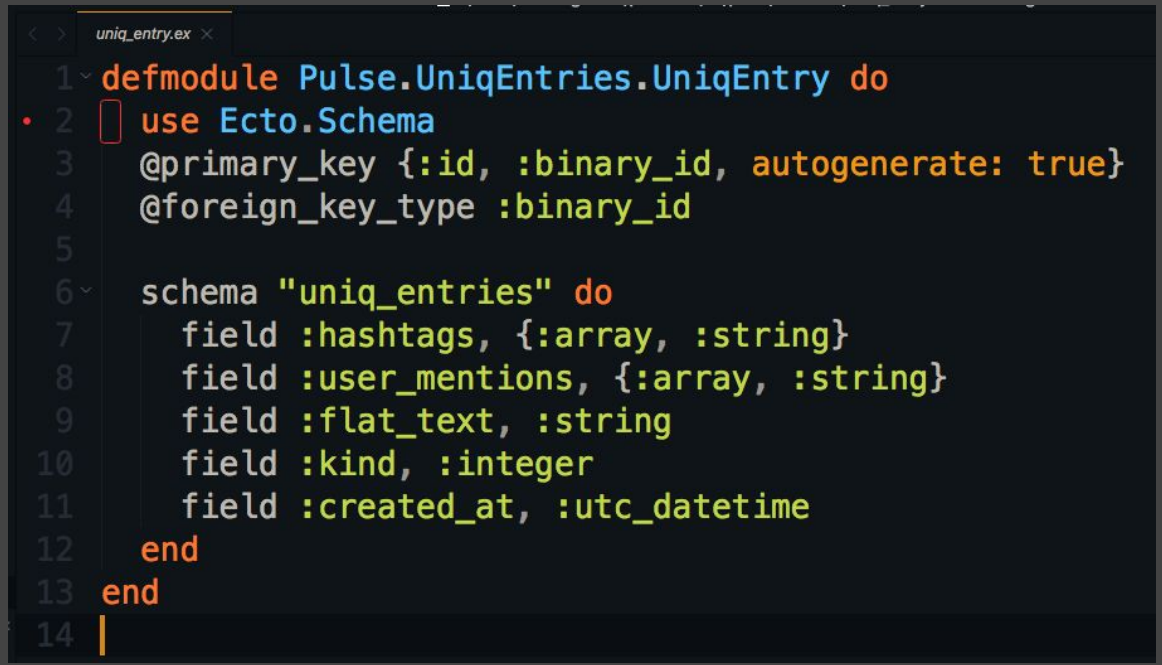
# Yeah

- Idea: "When a new tweet is saved into PostgreSQL, send a message to the UI to update the React Component"
- Implementation:
  - BG jobs send HTTP POST requests to Phoenix to notify of new tweets
  - Phoenix broadcast the message to the channel for all browsers (with the full payload)

# Config


prod.exs
use Mix.Config
# Configure your database
config :pulse, Pulse.Repo,
adapter: Ecto.Adapters.Postgres,
url: System.get_env("DATABASE_URL"),
pool_size: String.to_integer(System.get_env("POOL_SIZE") || "10"),
ssl: true

# Mapping


uniq_entry.ex
defmodule Pulse.UniqEntries.UniqEntry do
use Ecto.Schema
@primary_key {:id, :binary_id, autogenerate: true}
@foreign_key_type :binary_id
schema "uniq_entries" do
field :hashtags, {:array, :string}
field :user_mentions, {:array, :string}
field :flat_text, :string
field :kind, :integer
field :created_at, :utc_datetime
end
end

# Query

uniq_entry_keywords.ex

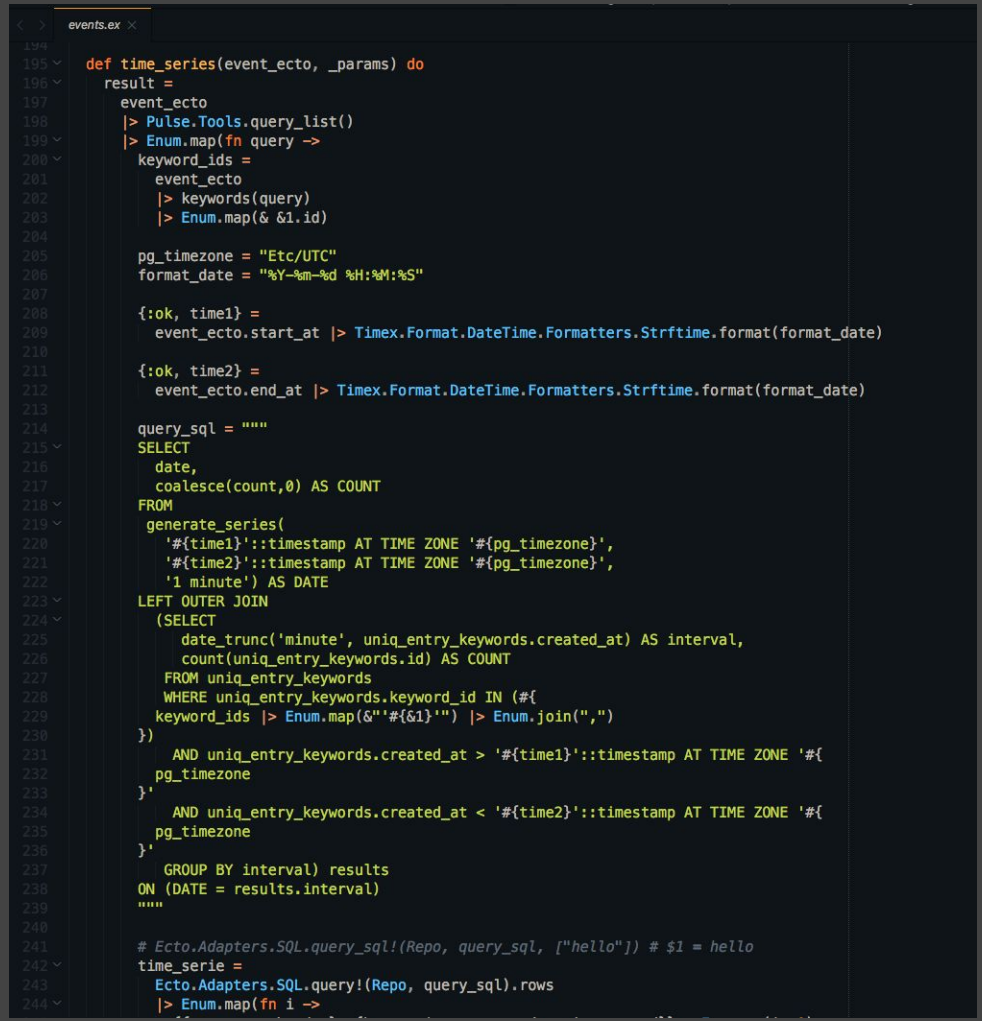
```elixir
defmodule Pulse.UniqEntryKeywords do
  import Ecto.Query, warn: false
  alias Pulse.Repo

  def count(event_ecto, keyword_ecto) do
    query =
      from(u in Pulse.UniqEntryKeywords.UniqEntryKeyword,
        where: u.keyword_id == ^keyword_ecto.id,
        where: u.created_at >= ^event_ecto.start_at,
        where: u.created_at <= ^event_ecto.end_at,
        select: count(u.id)
      )

    Repo.all(query) |> List.first()
  end

end
```

# More Query


events.ex
def time_series(event_ecto, _params) do
result =
event_ecto
|> Pulse.Tools.query_list()
|> Enum.map(fn query ->
keyword_ids =
event_ecto
|> keywords(query)
|> Enum.map(& &1.id)
pg_timezone = "Etc/UTC"
format_date = "%Y-%m-%d %H:%M:%S"
{:ok, time1} =
event_ecto.start_at |> Timex.Format.DateTime.Formatters.Strftime.format(format_date)
{:ok, time2} =
event_ecto.end_at |> Timex.Format.DateTime.Formatters.Strftime.format(format_date)
query_sql = """
SELECT
date,
coalesce(count,0) AS COUNT
FROM
generate_series(
'#{time1}'::timestamp AT TIME ZONE '#{pg_timezone}',
'#{time2}'::timestamp AT TIME ZONE '#{pg_timezone}',
'1 minute') AS DATE
LEFT OUTER JOIN
(SELECT
date_trunc('minute', uniq_entry_keywords.created_at) AS interval,
count(uniq_entry_keywords.id) AS COUNT
FROM uniq_entry_keywords
WHERE uniq_entry_keywords.keyword_id IN (#{
keyword_ids |> Enum.map(&"'#{&1}'") |> Enum.join(",")
})
AND uniq_entry_keywords.created_at > '#{time1}'::timestamp AT TIME ZONE '#{
pg_timezone
}'
AND uniq_entry_keywords.created_at < '#{time2}'::timestamp AT TIME ZONE '#{
pg_timezone
}'
GROUP BY interval) results
ON (DATE = results.interval)
"""
# Ecto.Adapters.SQL.query_sql!(Repo, query_sql, ["hello"]) # $1 = hello
time_serie =
Ecto.Adapters.SQL.query!(Repo, query_sql).rows
|> Enum.map(fn i ->

# Hypothesis

---

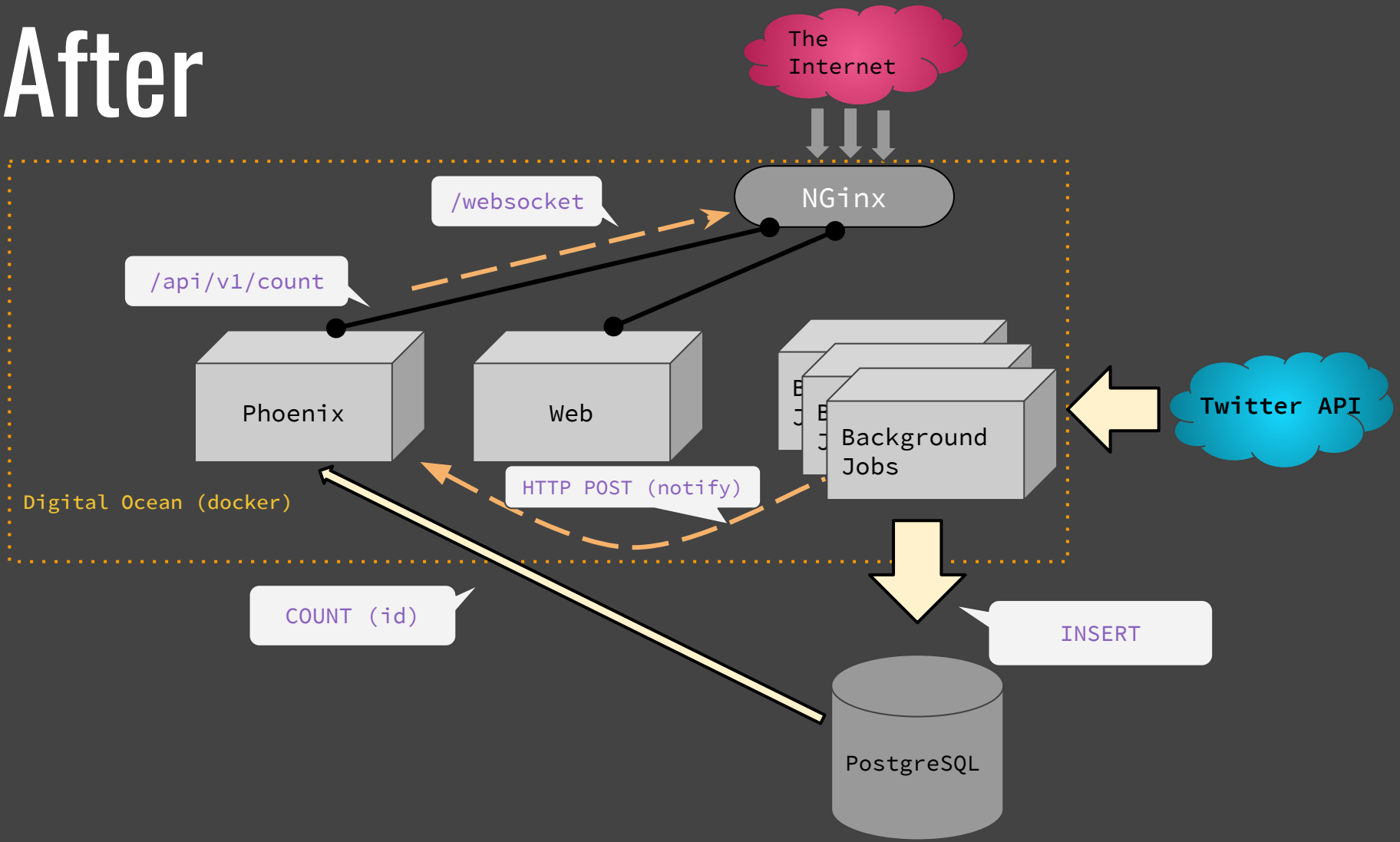
**Ecto is gonna be hard to setup.**

- Bullshit 1
- Bullshit 2
- Refer to Bullshit 1

**Queries will be hard to write**

- Bullshit 1
- Bullshit 2

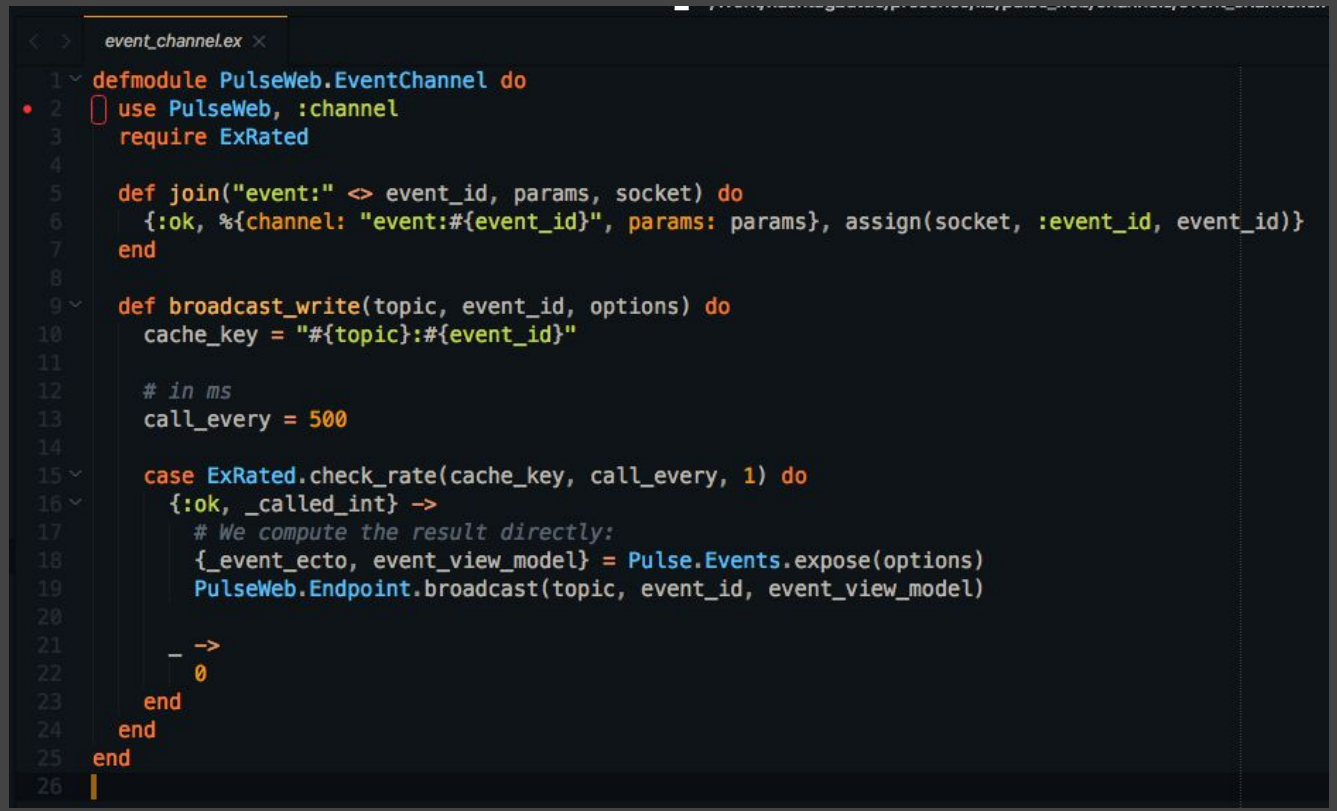
# After


The Internet
NGinx
/websocket
/api/v1/count
Phoenix
Web
Background Jobs
Twitter API
Digital Ocean (docker)
HTTP POST (notify)
COUNT (id)
INSERT
PostgreSQL

# Oups

- I wanted to connect the BG through WebSocket directly to Phoenix Channel, but fallback to simple HTTP POST requests.
- Too much processing.
  Solution: Throttle request to compute/broadcast message (with ExRated)

# ExRated


event_channel.ex
defmodule PulseWeb.EventChannel do
use PulseWeb, :channel
require ExRated
def join("event:" <> event_id, params, socket) do
{:ok, %{channel: "event:#{event_id}", params: params}, assign(socket, :event_id, event_id)}
end
def broadcast_write(topic, event_id, options) do
cache_key = "#{topic}:#{event_id}"
# in ms
call_every = 500
case ExRated.check_rate(cache_key, call_every, 1) do
{:ok, _called_int} ->
# We compute the result directly:
{_event_ecto, event_view_model} = Pulse.Events.expose(options)
PulseWeb.Endpoint.broadcast(topic, event_id, event_view_model)
_ ->
0
end
end
end

# Controller

event_controller.ex

```elixir
defmodule PulseWeb.EventController do
  use PulseWeb, :controller
  plug(:authenticate_user_by_api_key!, "" when action in [:show, :index])

  # resources "/events", EventController, only: [:show, :create, :index]
  # POST /v1/events
  def create(conn, %{"topic" => topic, "event" => event_id, "payload" => payload}) do
    PulseWeb.EventChannel.broadcast_write(topic, event_id, payload)
    conn |> send_resp(204, "")
  end

  # resources "/events", EventController, only: [:show, :create, :index]
  # GET /v1/events/:event_id
  def show(conn, %{"id" => id}) do
    payload = %{"event_id" => id}

    {_event_ecto, event_view_model} = Pulse.Events.expose(payload)

    conn
    |> Plug.Conn.put_resp_header("content-type", "application/json; charset=utf-8")
    |> Plug.Conn.put_resp_header("cache-control", "max-age=3")
    |> Plug.Conn.send_resp(200, Poison.encode!(event_view_model, pretty: false))
  end

  # resources "/events", EventController, only: [:show, :create, :index]
  # GET /v1/events
  def index(conn, params) do
    data = Pulse.Events.index(params)

    conn
    |> Plug.Conn.put_resp_header("content-type", "application/json; charset=utf-8")
    |> Plug.Conn.put_resp_header("cache-control", "max-age=3")
    |> Plug.Conn.send_resp(200, Poison.encode!(data, pretty: false))
  end

  defp authenticate_user_by_api_key!(conn, _params) do
    user = Pulse.Users.auth(conn.params)

    if !user do
      raise "Idiot."
    end

    conn
  end
end
```

“Fat Controller? No.”

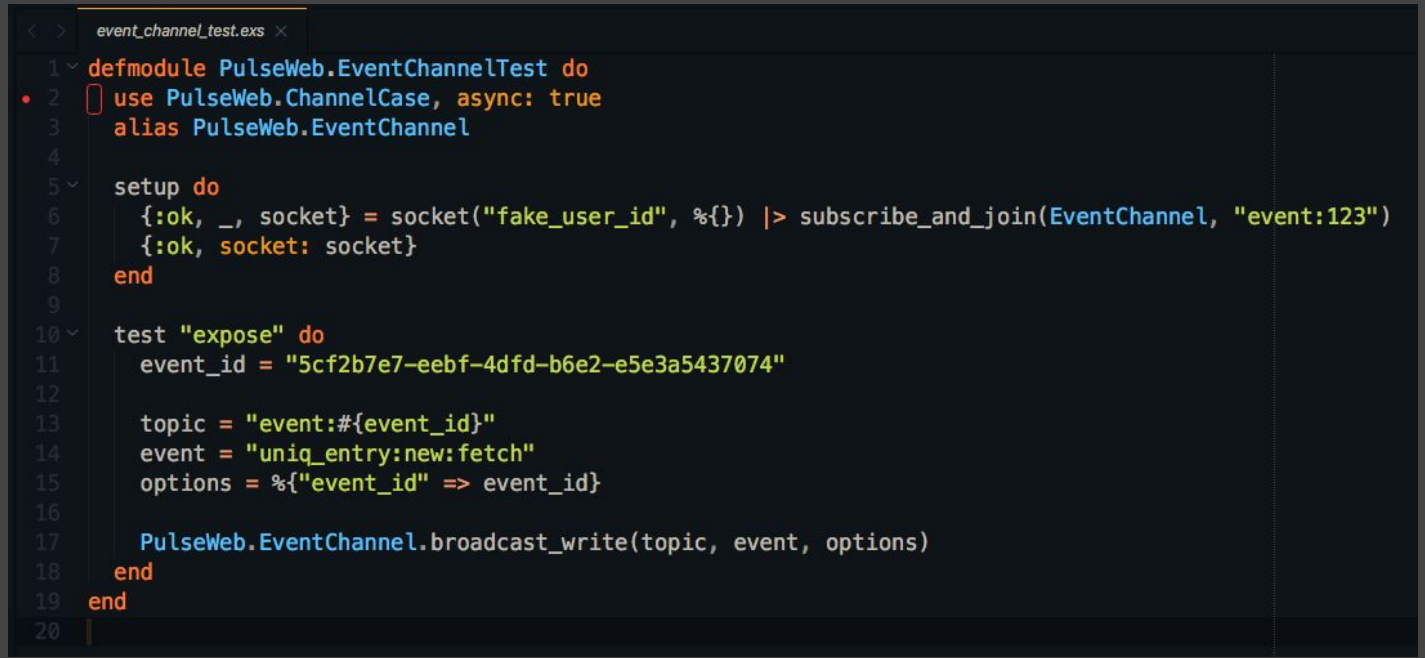
---

# Joy

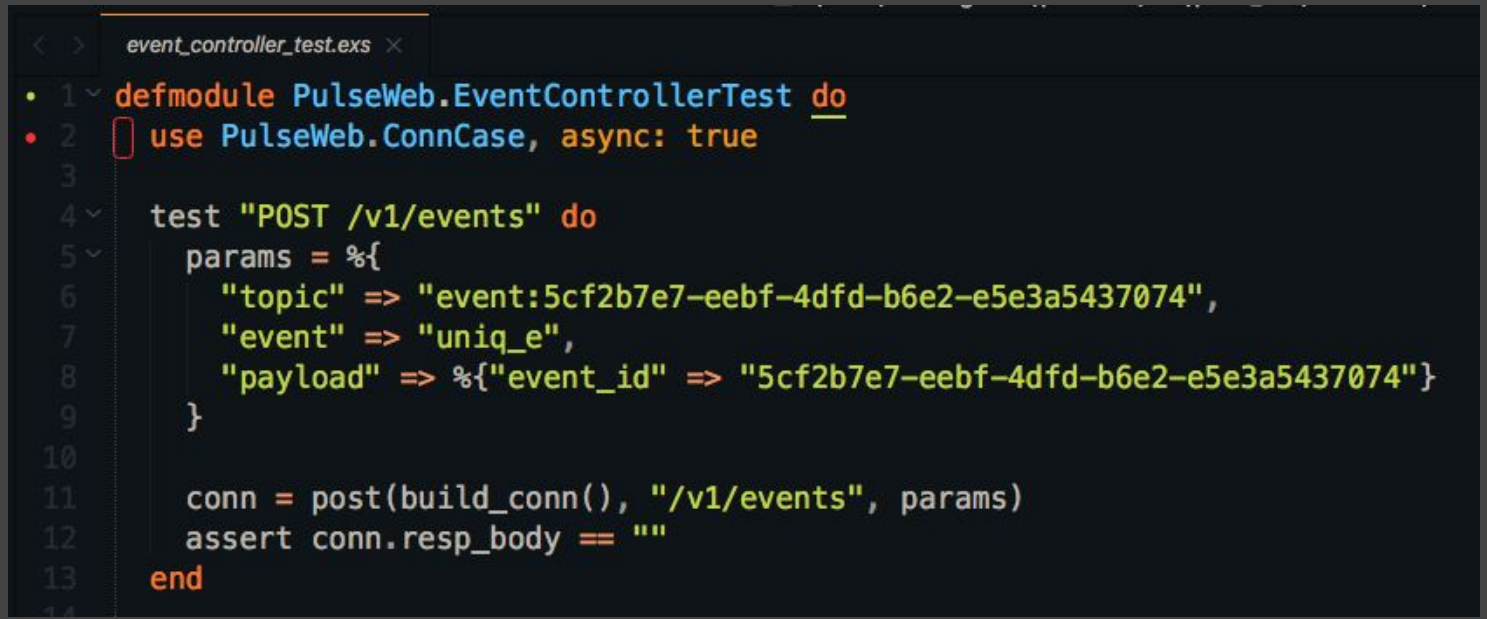
- Testing the app from the outside, complete integration testing.
- Functional: compile checking
- Parallel tests
  (by default, super fast)
- Easy to find documentation
- FUNCTIONS AND MAPS.
  THAT's IT.
- 0 bad surprises.

---

# Test Channels



# Test Controllers



“Peace.”

———

# Just starting.

- Baby steps are possibles, total time from learning to production: 10 working days. In production since August, 0 unexpected crashed.
- Background processing: large CSV files (Ruby is super bad at it), image processing, real time communication with clients.
- Failures first: helps you a lot to anticipate bad scenario and edge cases.
- Juniors: not that hard! I would love to have feedback about your experience.

# Thanks!

*All cat pictures are mine. If you want to use them, no problem, but I'll ask a LOT of money.*