Real Time Systems

About us

Grant Powell

Grant Powell

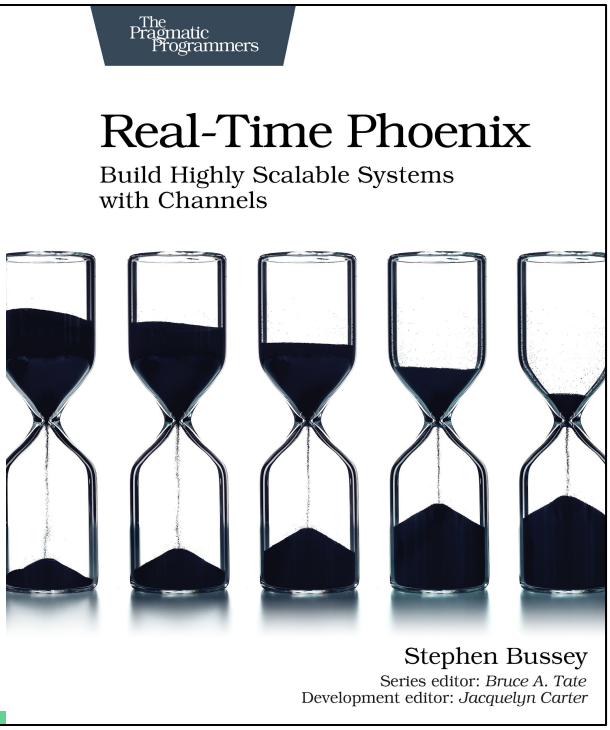
- Senior Software Engineer at SalesLoft
- Love building things people use
- Worked in research labs before coming to SalesLoft
- Enjoys all the functional programming languages
- Enjoys playing golf and working with habitat for humanity

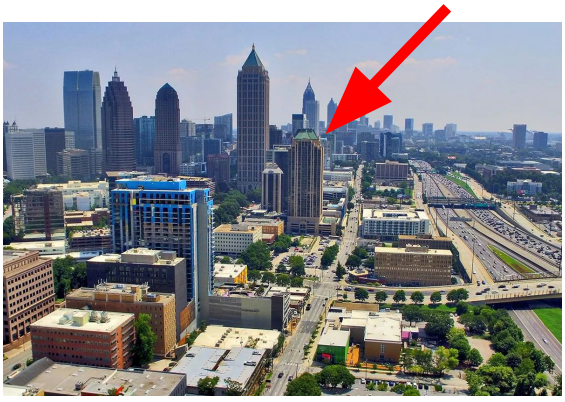
Steve
Bussey

STEVE'S BOOK JUST WENT BETA!

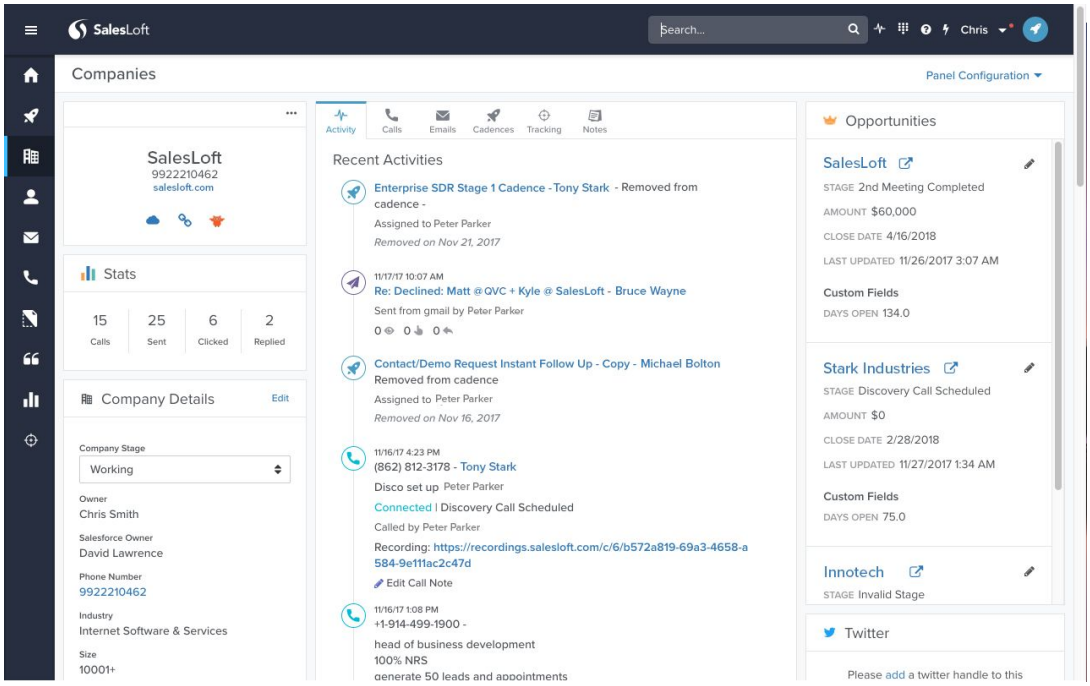
Available Wednesday

Discount code available





We're on the engineering team at SalesLoft in midtown Atlanta. Our desks are right where the arrow is pointing


SalesLoft
Search...
Chris
Companies
Panel Configuration
SalesLoft
9922210462
salesloft.com
Stats
15
Calls
25
Sent
6
Clicked
2
Replied
Company Details
Edit
Company Stage
Working
Owner
Chris Smith
Salesforce Owner
David Lawrence
Phone Number
9922210462
Industry
Internet Software & Services
Size
10001+
Activity
Calls
Emails
Cadences
Tracking
Notes
Recent Activities
Enterprise SDR Stage 1 Cadence - Tony Stark - Removed from cadence -
Assigned to Peter Parker
Removed on Nov 21, 2017
11/17/17 10:07 AM
Re: Declined: Matt @ QVC + Kyle @ SalesLoft - Bruce Wayne
Sent from gmail by Peter Parker
0 0 0
Contact/Demo Request Instant Follow Up - Copy - Michael Bolton
Removed from cadence
Assigned to Peter Parker
Removed on Nov 16, 2017
11/16/17 4:23 PM
(862) 812-3178 - Tony Stark
Disco set up Peter Parker
Connected | Discovery Call Scheduled
Called by Peter Parker
Recording: https://recordings.salesloft.com/c/6/b572a819-69a3-4658-a584-9e111ac2c47d
Edit Call Note
11/16/17 1:08 PM
+1-914-499-1900 -
head of business development
100% NRS
generate 50 leads and appointments
Opportunities
SalesLoft
STAGE 2nd Meeting Completed
AMOUNT $60,000
CLOSE DATE 4/16/2018
LAST UPDATED 11/26/2017 3:07 AM
Custom Fields
DAYS OPEN 134.0
Stark Industries
STAGE Discovery Call Scheduled
AMOUNT $0
CLOSE DATE 2/28/2018
LAST UPDATED 11/27/2017 1:34 AM
Custom Fields
DAYS OPEN 75.0
Innotech
STAGE Invalid Stage
Twitter
Please add a twitter handle to this

About Y'all

About you!

- What's your name?
- Your experience with software?
- What types of problems do you like to solve?
- What languages do you primarily use?
- Why did you want to come to Elixir Conf?

Which Database

What will you be able to take away from today?

- Foundational concepts of real-time systems
- Take-away code that you could use to get started with other projects

Roadmap

- Introduction
- Data Pipelines
- GenStage
- Phoenix 101
- Remote Nodes, Pub Sub, Channels
- Production and Deployment
- Advanced Topics

- Build a data pipeline with GenStage
- Use Phoenix Channels to receive / send data to users
- Use advanced Phoenix tools like Tracker / LiveView
- Understand general challenges with real-time systems running in production

What is a (soft) real time system?

Hard Real Time

Soft Real Time

- "Real Time System" or "Hard Real Time System" has a very strict math meaning. A hard real time system has hard deadlines in which it must return data, and is considered to be failed if that deadline is exceeded.
- Real time systems are often found in things like fighter jets and pacemakers
- The BEAM is **NOT** a hard real time system
- When we refer to real time systems we're referring to soft real time systems

Why do Real-Time Systems Matter?

- Go to next slide

User expectations have risen

- Users expect data on the screen to reflect **NOW**, before they ask for it
- Win user trust with real time features

- You want changes in a Google Doc to happen seamlessly
- New items on Twitter feed come in as they happen
- Telling a user that the shoes in their shopping cart just went out of stock before they enter all of their payment information

Why Elixir, and Why Now?

- Elixir's OTP backbone can easily model and solve the challenges of building real time systems
- Advances in Web standards have lead to new real time communication layers

10,000 foot view of the BEAM

- Next slide

BEAM

Björn's **E**rlang

Abstract **M**achine

The BEAM is a virtual machine, much in the same way the JVM and .NET framework

BEAM Vocab

A Process – a unit of execution in the BEAM. BEAM Processes share **NOTHING** with each other.

A Message – A process can send a message to another process.

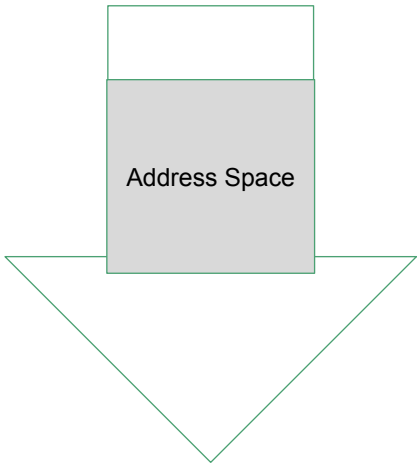
- Process shared nothing architecture is pretty important to BEAM characteristics and performance

Processes

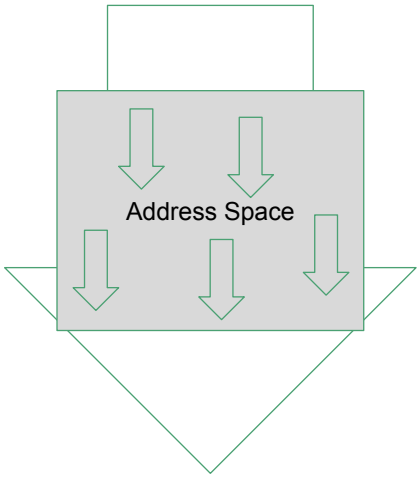
- Beam Processes are not OS processes

Understanding OS processes and Threads

A process contains one address space



A process can contain one to many threads



- Many programming languages (like ruby, python, java) rely on OS concurrency primitives in order to be concurrent
- The two OS concurrency primitives are threads and processes
- A process is an independent unit of execution in the OS, it shares nothing with other processes.
- Threads are independent units of execution within a process. Threads share memory between them.
- OS Processes are more heavy weight than OS threads
- OS Processes and OS threads can physically run in parallel on multi cpu architectures

BEAM processes are **NOT** OS Process or OS Threads

BEAM Processes

- Lightweight - (338 words * 64bit machine ≃ 2Kb)
- Don't share memory*
- Don't share garbage collection (GC)

- BEAM processes are VM provided concurrency primativatives that are optimized for the following things
  - Lightweight (being able to run many at the same time)
  - Failure tolerance (a process can fail without affecting other processes)
  - Equitable CPU distribution (a process can't hog too many resources)
  - Low latency (no large pauses, no stop the world events)
- One of the neat things that fell out of the above characteristics is that the BEAM is awesome at taking advantage of multicore architectures
- The BEAM garbage collection is very well suited to real-time systems. Having a shared nothing garbage collection allows processes to be garbage collected in parallel. BEAM processes are easier to garbage collect because memory can't be shared across processes, so all allocated memory is process local.
- One common term you'll hear to describe BEAM processes is "Green threads" this is a little misleading because erlang processes don't share state relative to the programmer, so BEAM processes are more like "Green processes"

Why does the BEAM matter?

Or as I like to explain it...

The BEAM and Star Wars

The BEAM and Star Wars

Traditional OS primitive concurrency systems

I typically break web frameworks concurrency primitives into two groups

1. The web applications that rely on OS concurrency primitives (OS threads and OS Processes)
2. The web applications that use VM provided concurrency primitives

OS processes/threads tend to be like Jedi. Jedi are powerful, but their numbers are limited.

OS processes tend to be good when there is a single task at hand and the task requires much of CPU. This is similar to how Jedi are best at fighting Sith, one at a time.

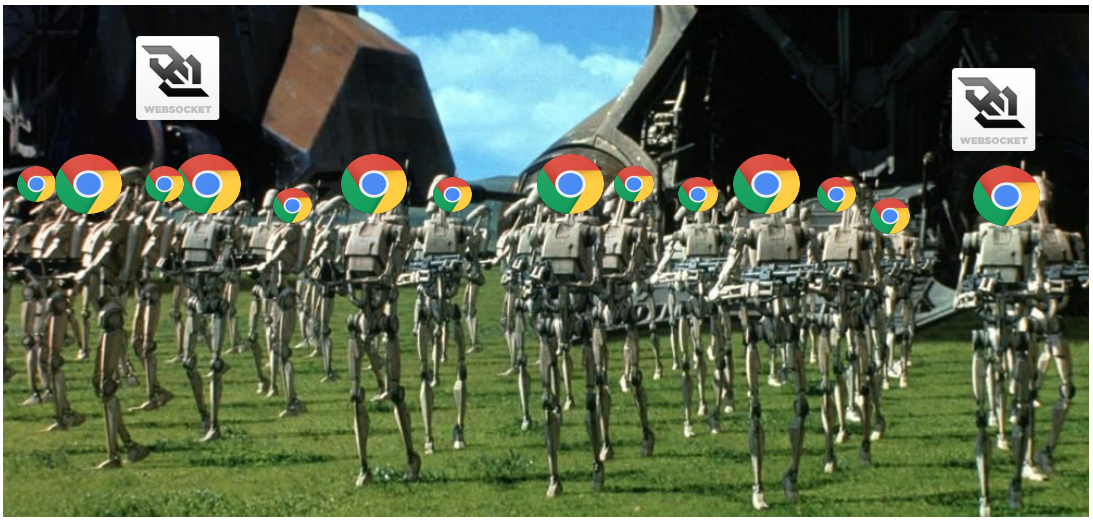
OS processes/threads take considerable time, cpu, and memory to start up. In a similar way Jedi take a long time to train and prepare

Because OS processes/threads are heavy, a system is more limited in the number of processes/threads it that can have.

Jedi Aren't Great at Concurrency

With 6 Jedi's you can serve at most 6 concurrent requests

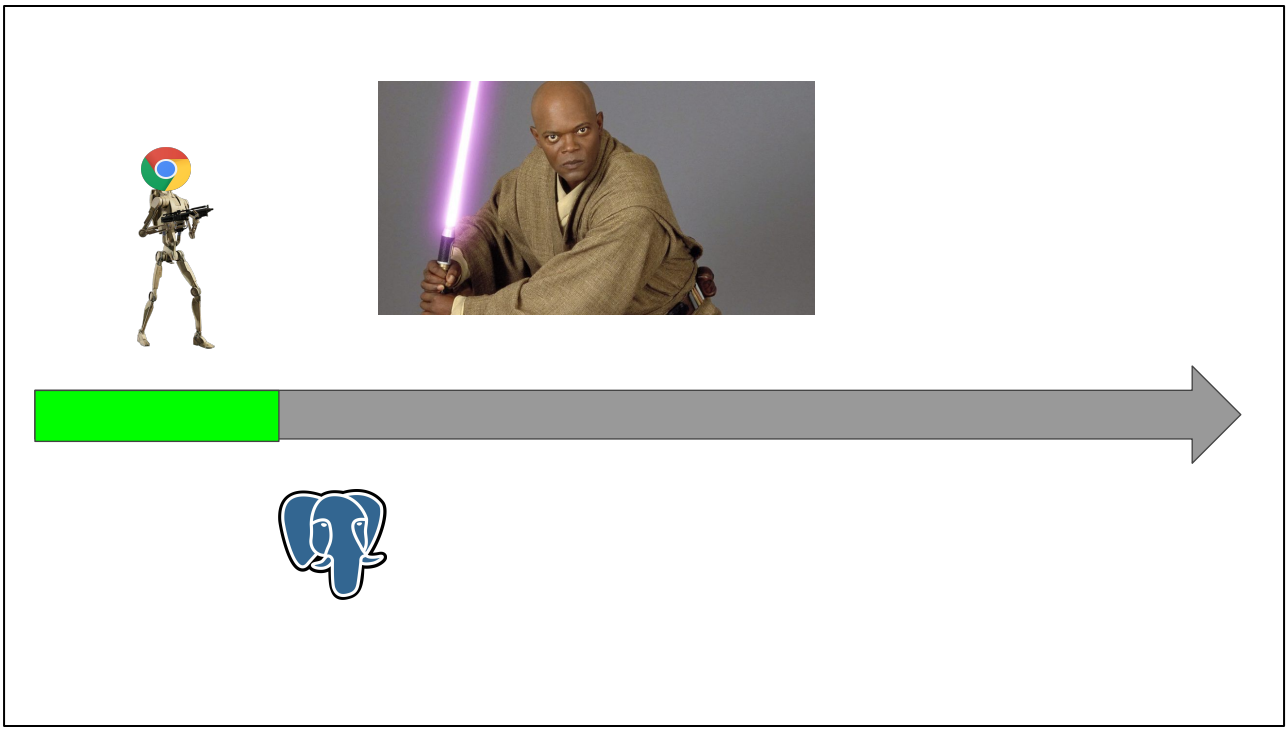
Jedi are good at fighting one opponent at a time.



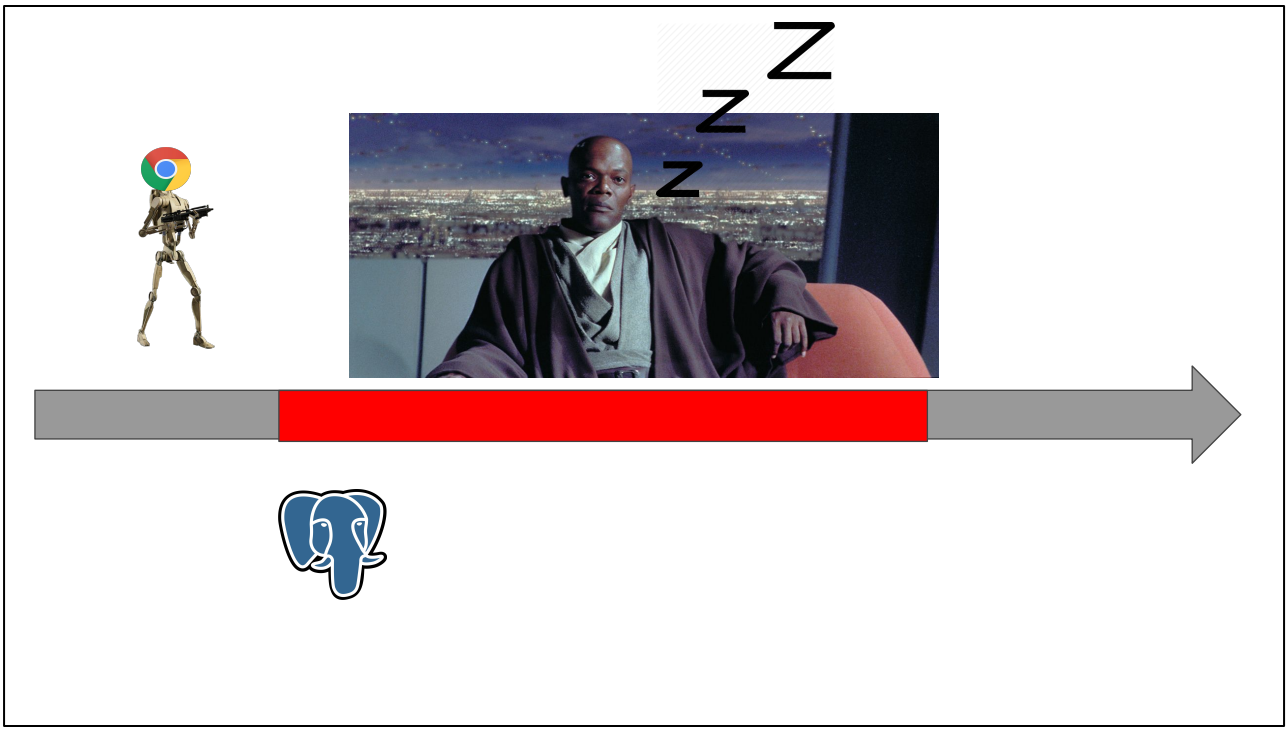
The web is a different enemy than Darth Maul, the web is more like the droid army. Each droid is not strong, but there are many many droids.

Web requests are a different beast.

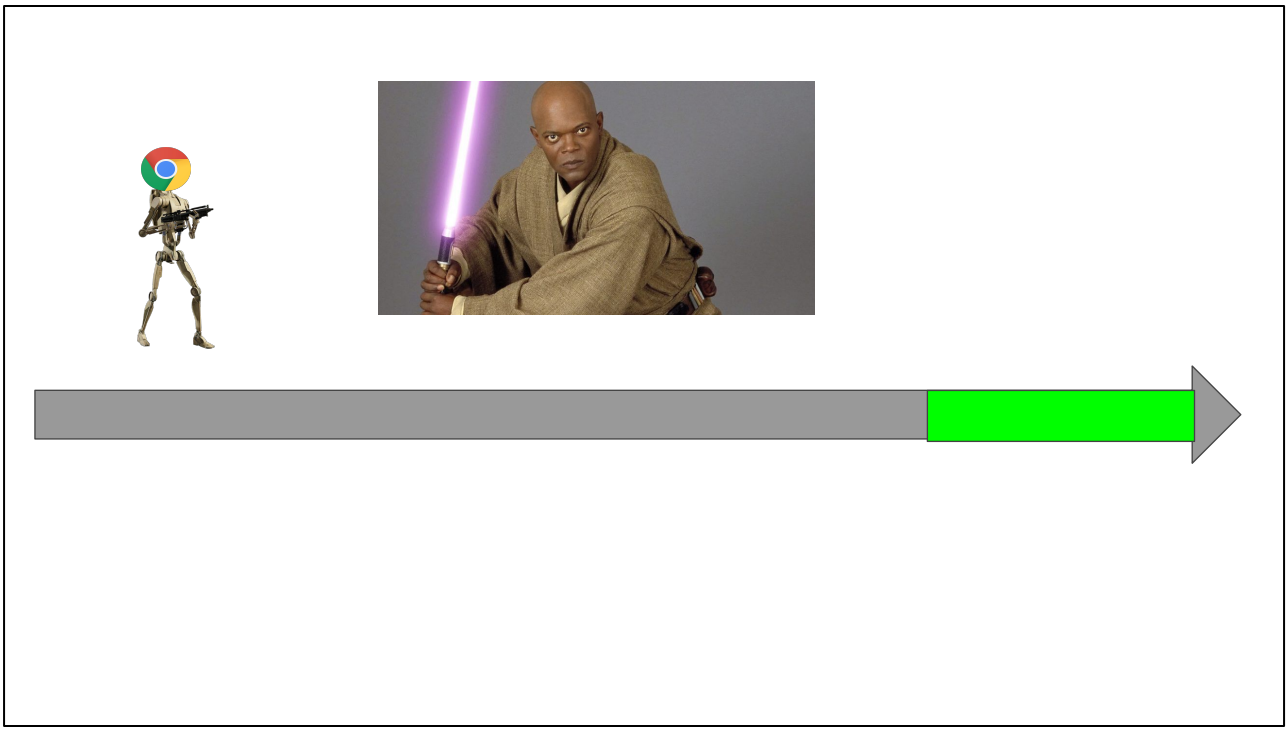
If you've seen *Star Wars Episode II*, then you know that the Jedi had trouble when fighting numerous opponents.



When serving a web request with an OS process/thread based web application, it's like having Mace Windu fight a droid. Mace does a little bit of fighting, but then he calls Postgres



While he calls Postgres, he just hangs out in the Jedi chambers waiting.



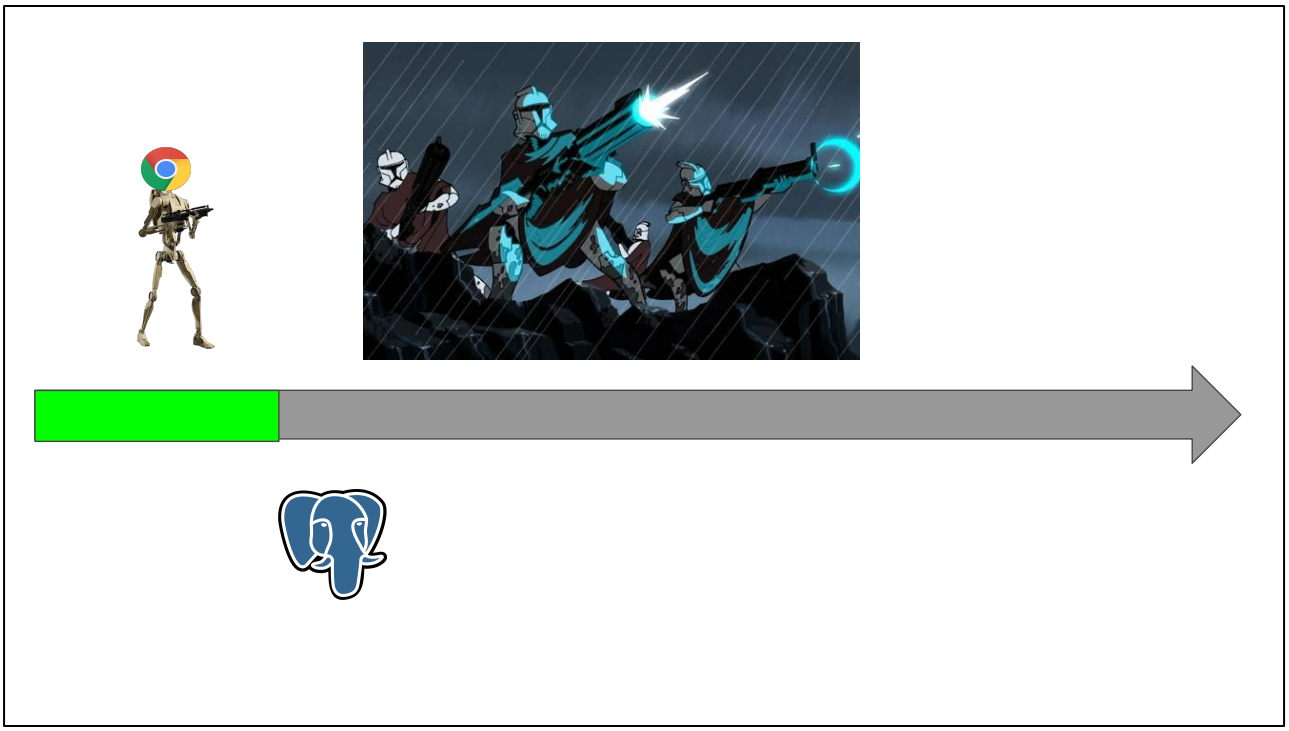
Until Postgres returns, then he can fight again. **Most** of Mace Windu's time is spent waiting! Not a great use of Mace Windu if you ask me!

The Elixir Clone troopers are a much better strategy for for dealing with the Droid army of web requests

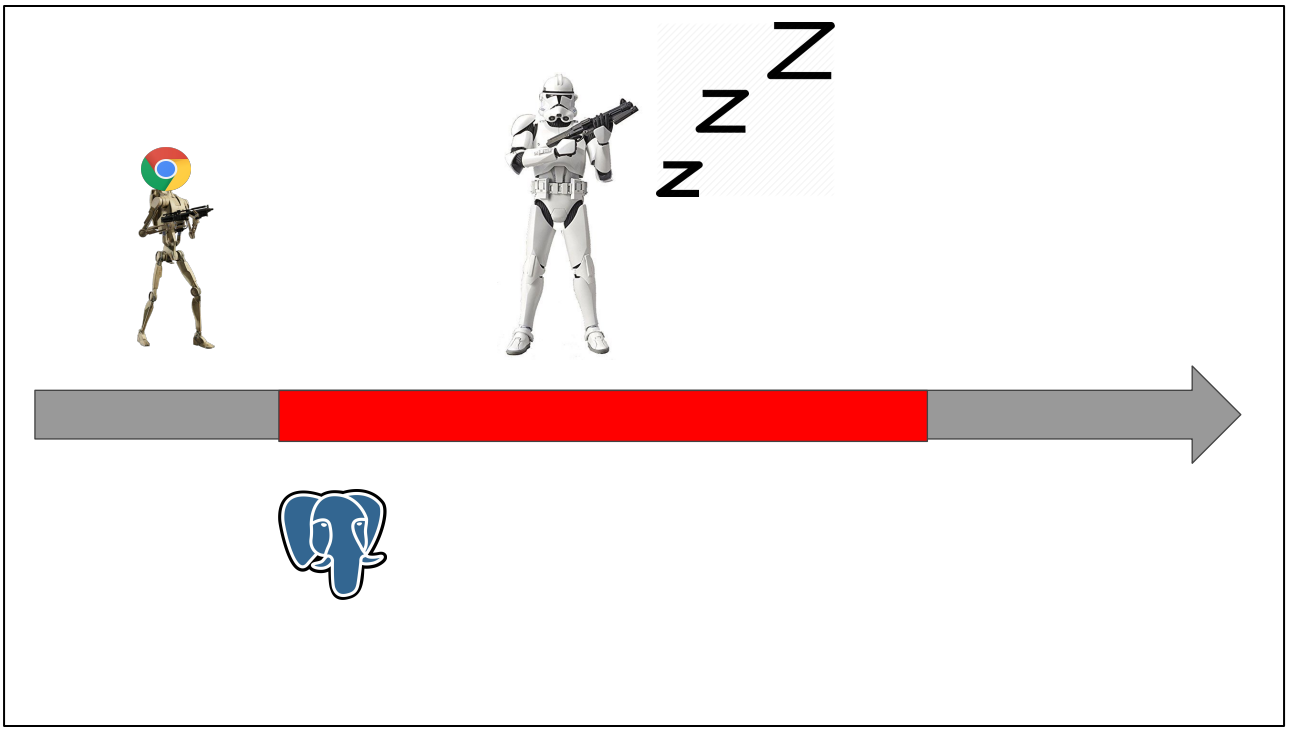
Clone troopers are bad at concurrency too

But there are many many more of them

Jedi are good at fighting one opponent at a time.



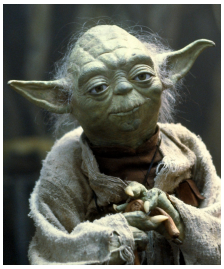
A clone trooper (or a few) do a little work before calling Postgres.



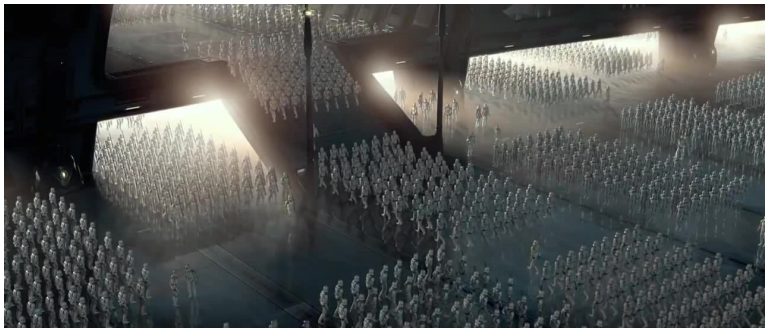
Then wait for Postgres to return.

Then do some more work. It's fine though! There are many many clone troopers.

The BEAM

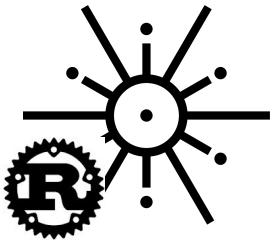


BEAM Processes



A BEAM node runs one OS processes and (usually) one OS thread per core, and runs 10s of thousands of Green Processes (clone troopers). This concurrency architecture is a fantastic way to build web apps that deal with a IO wait like talking to redis/databases/other web services

Other languages/frameworks are trying to build clone armies



Ruby/EventMachine

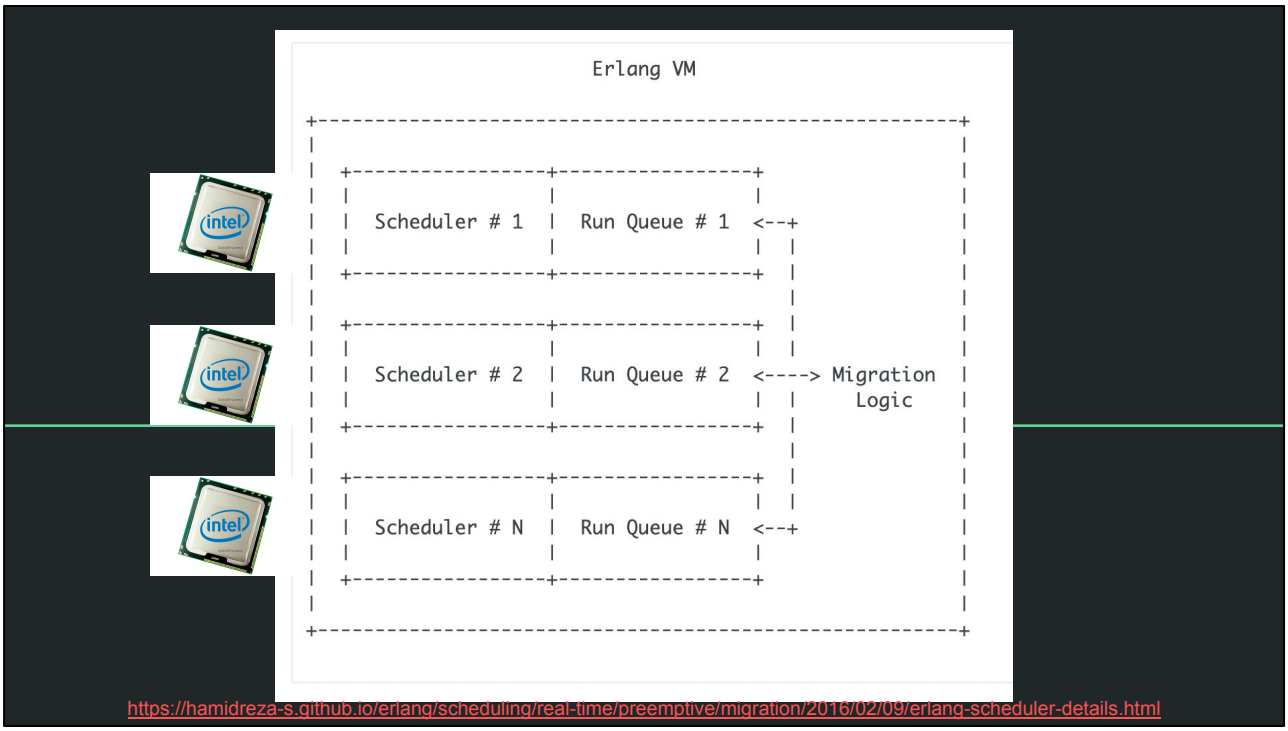
Fast Network I/O and Event Management for Ruby Programmers

Other languages are trying to build their own armies. Other languages tend to have historical reasons why they can't compete with the BEAM's outstanding army.

We're not going into this here, but I find it a fascinating topic. I'm more than happy to talk about it at the end of the day

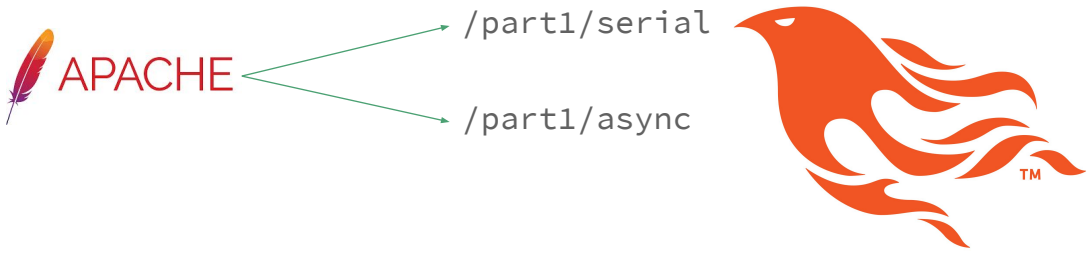
The BEAM scheduler

-



- Source: https://hamidreza-s.github.io/erlang/scheduling/real-time/preemptive/migration/2016/02/09/erlang-scheduler-details.html

Example 1



In the first example, we're going to be using the Apache Benchmark tool to hit two different endpoints on an example phoenix app.

These two endpoints do the same mock IO bound task, but they do it in different ways.

The tool we will be using is Apache Benchmark, and we will instruct it to use 50 concurrent workers and record the speed of the requests

```elixir
defmodule Example1Web.Part1Controller do
  use Example1Web, :controller
  alias Example1.MockResource

  def serial(conn, _params) do
    :ok = MockResource.use_resource(Part1)
    text(conn, "OK")
  end

  def async(conn, _params) do
    spawn(fn -> :ok = MockResource.use_resource(Part1) end)
    text(conn, "OK")
  end
end
```

1. The **`serial`** endpoint does some work and returns the answer
2. The **`async`** endpoint starts a new process to the do the work, then immediately returns to the caller
3. The work done by this endpoint is a 500ms "Web Request" (Actually a `Process.sleep/1`). You can imagine the MockResource like a database, or a web request, or some other IO wait bound task.

Questions to explore

- What will the response times of each endpoint will be?
- What will the memory usage of each of the benchmarks will be?
- What will the concurrent usage of our Mock Resource be for each endpoint?
- How can I answer these questions using the BEAM's built in **`:observer`** tool?

Repo available at bit.ly/elixir-conf-training-1

Share with Us

- Invite someone up to discuss what occurred for them

Our perspective:

- Serially ran 50 Process.sleep at a time
- Task.async essentially ran all Process.sleep within 500ms
- Notice the spike in memory with Task.async
- Notice the amount of work queued up with Task.async (uncontrolled)
- Theorize what would happen if that work used the database, another HTTP service

S + G PRO TIP

BE CAUTIOUS: The Single Global Process (SGP) is easy to over apply, but can cause performance problems in production. **`MockResource`** in the previous example is an example of the SGP antipattern

- https://keathley.io/blog/sgp.html

Section 1 — Part 2

- Breaker upper

Uncontrolled Parallelism = Instability

- In the BEAM processes can run in parallel
- Uncontrolled parallelism leads to blowing up downstream services
- Sometimes we have to reign in parallelism in order to be stable—stay in control
- We may want tasks to run in parallel, but maybe also serially to their parent process

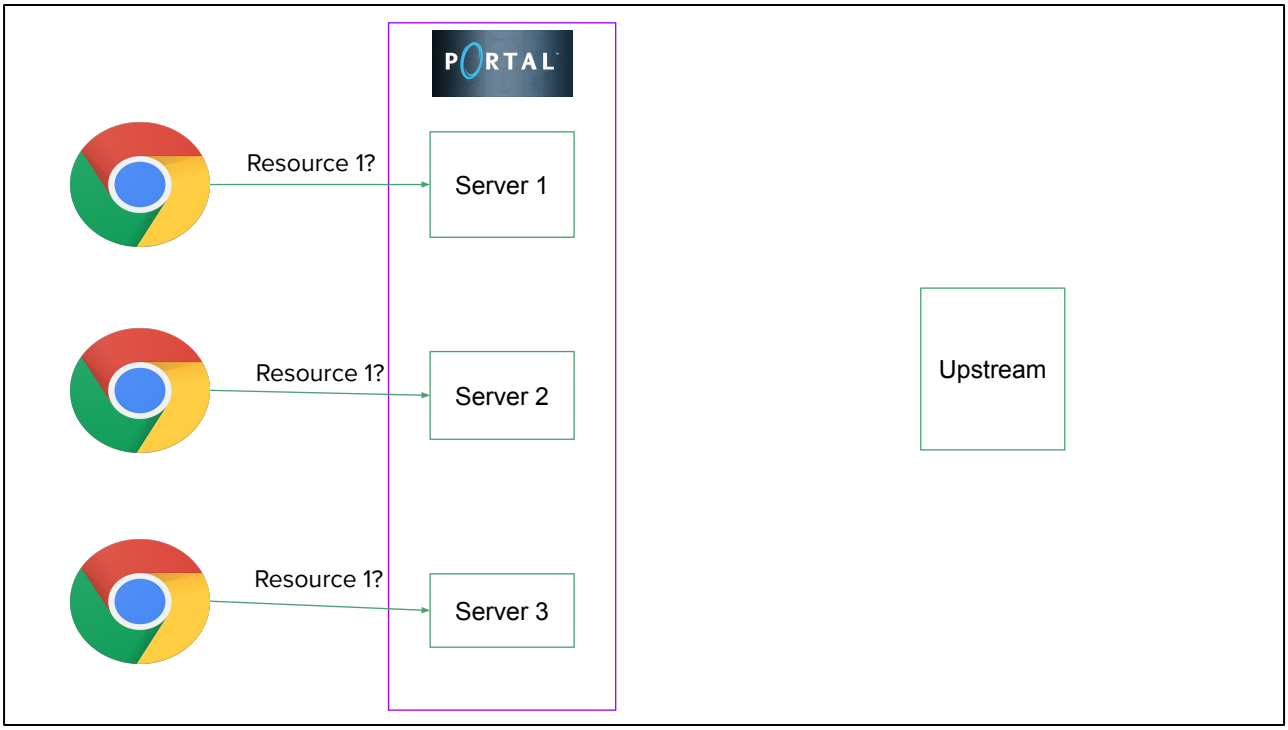
A majority of the performance problems we've had with Elixir was it going too parallel

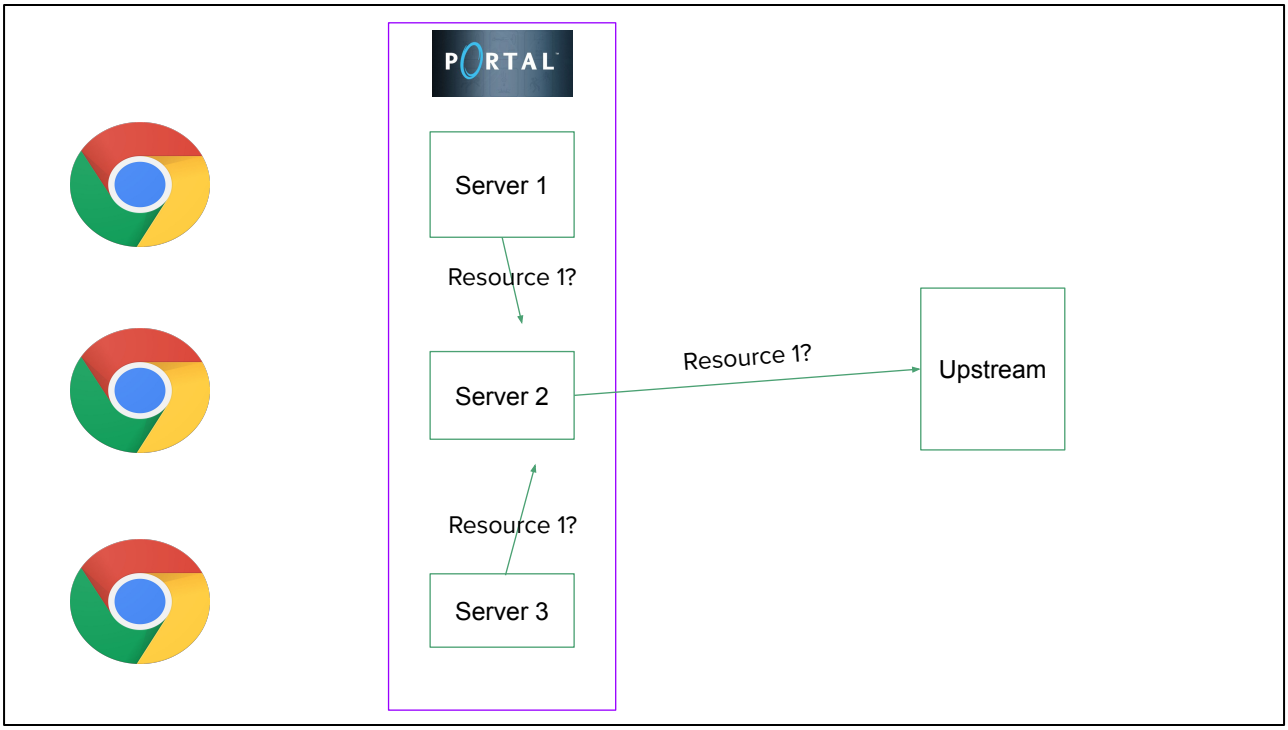
Our experience in learning to put Elixir into production at SalesLoft is that the BEAM is capable of massive concurrency, and that can put strain on other resources.

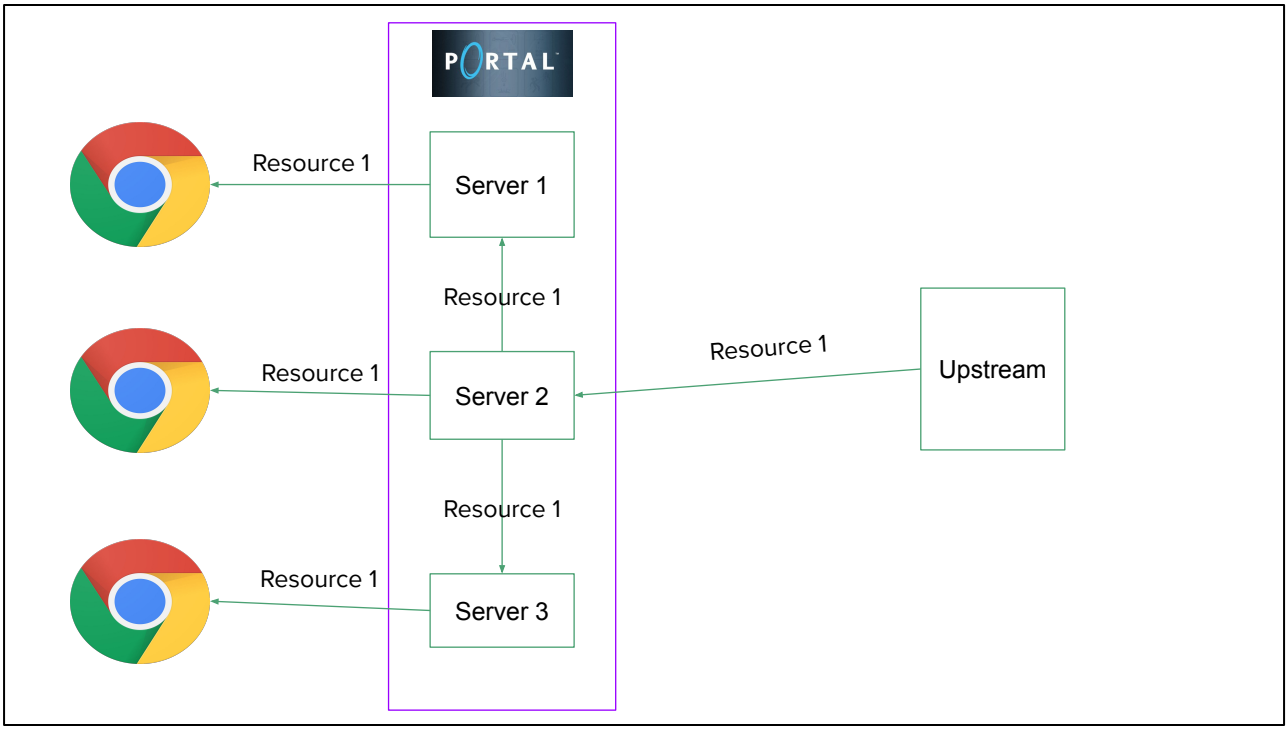
Case Study: Portal

- Unique web request made for a resource across the entire cluster
- Resource is requested exactly at the same time, not directly cacheable
- Highly concurrent across all resources
- Cache a given resource for 30 seconds

- Achieved by using DynamicSupervisor on a specific node (hash the resource ID)
- If process exists, return the data in memory
- When process is created, request the resource

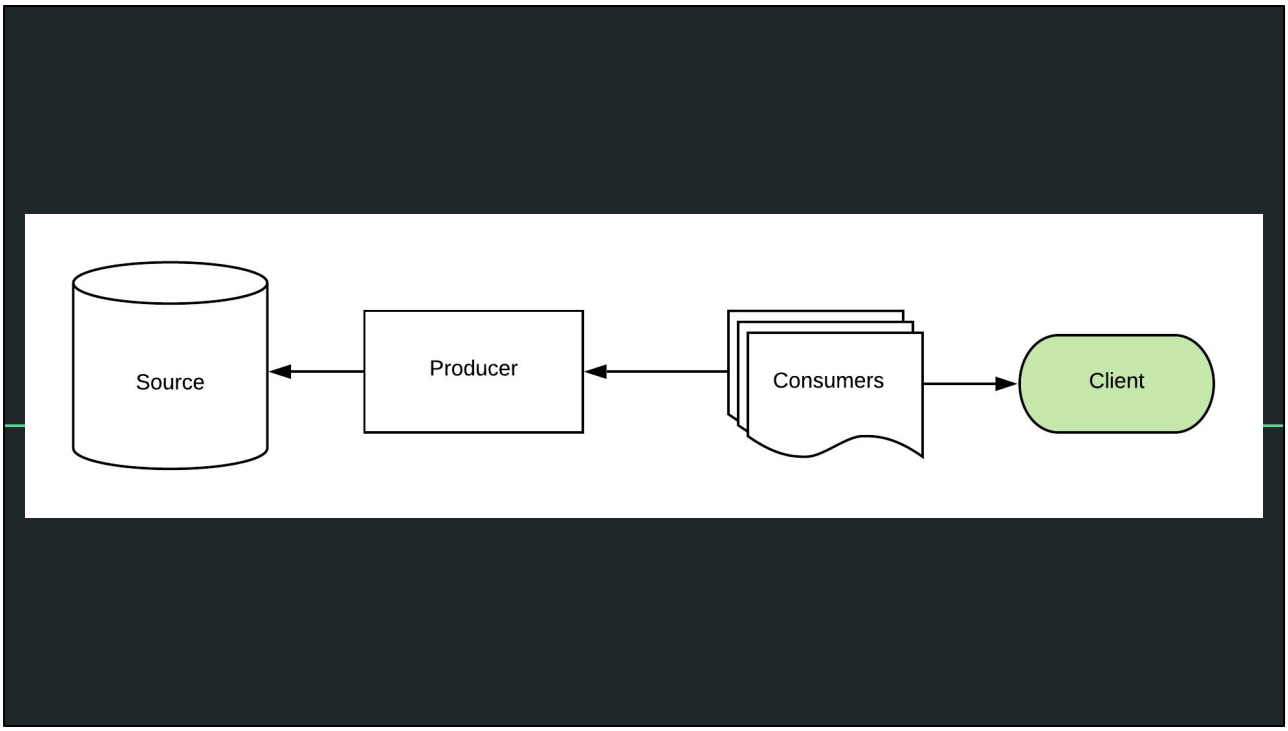

PORTAL
Resource 1?
Server 1
Resource 1?
Server 2
Resource 1?
Server 3
Upstream


PORTAL
Server 1
Resource 1?
Server 2
Resource 1?
Upstream
Resource 1?
Server 3


PORTAL
Resource 1
Server 1
Resource 1
Resource 1
Server 2
Resource 1
Upstream
Resource 1
Resource 1
Server 3

What is a data pipeline?

- Move data from source->destination throughout your system
- May or may not control the source of the data, could be user provided
- Goal of moving the data is to get it to users as quickly, and seamlessly, as possible


Source
Producer
Consumers
Client

The Function

- The function is the simplest form of mapping a values to values
- Most of your business logic should be implemented in functions

```
def double_v1(number) do
  number * 2
end
```

Composing Functions

- Once you have the building blocks of your data pipeline you want to compose them into larger pieces of functionality
- Choosing how you compose your functions will determine the runtime tradeoffs your data pipeline makes

```
def double_plus_1_exclaimer(n) do
  n
  |> double
  |> add1
  |> exclaim
end
```

Functions Can be Composed in Different Ways for Different Runtime Characteristics

```
[1, 2, 3, 4]
|> Enum.map(&double/1)
|> Enum.map(&add1/1)
|> Enum.map(&exclaim/1)
```

```
[1, 2, 3, 4]
|> Stream.map(&double/1)
|> Stream.map(&add1/1)
|> Stream.map(&exclaim/1)
```

```
[1, 2, 3, 4]
|> Flow.from_enumerable()
|> Flow.map(&double/1)
|> Flow.map(&add1/1)
|> Flow.map(&exclaim/1)
```

- Doesn’t change the actual function, just the runtime characteristics

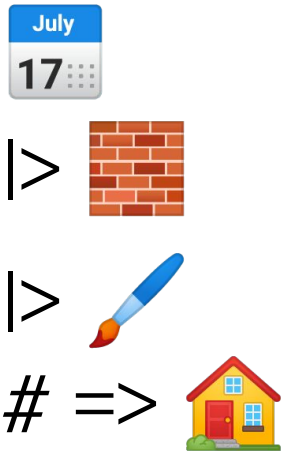
S + G PRO TIP

DO: Use BEAM processes to control runtime concurrency characteristics

CAUTIOUS: BEAM processes to separate business logic. Use modules to separate business logic into a functional core

- This is the biggest mistake that I made when I was learning Elixir, and a mistake I frequently correct as other people at SalesLoft learn Elixir
- A lot of people getting started with Elixir jump to processes like they're classes, probably a mistake

A Pipeline Composed With |> Run in One Process



📅 = Getting an order for a house
🧱 = Bricking a house

🖌️ = painting a house
🏠 = a completed house

This clone trooper takes an order for a house, and then lays bricks, and then paints the house, and the result is a house

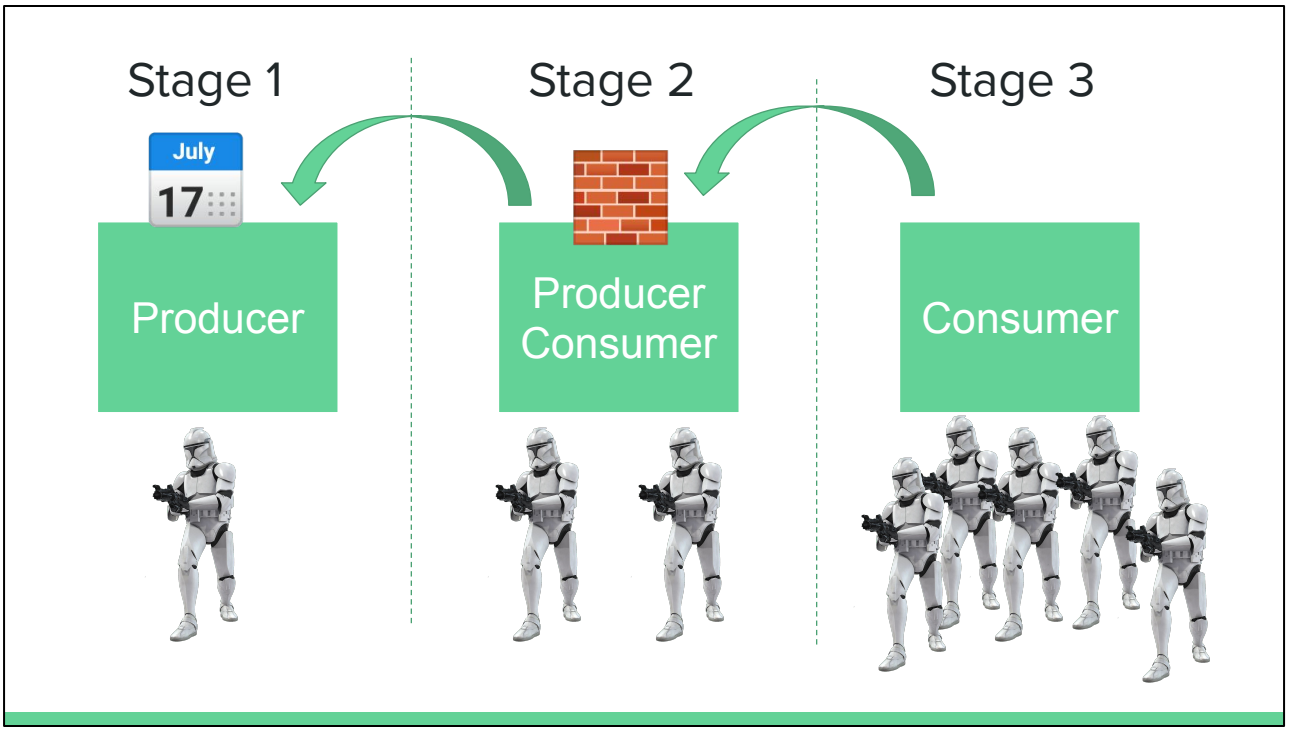
What if we can only be bricking 4 houses at a time? What happens if we can only have 2 painters at a time?

What if I told you getting 1 order for a house took as long as getting 50 orders? How would that affect this current set ups throughput?

GenStage is a toolkit to compose functions with specific runtime characteristics

GenStage is....

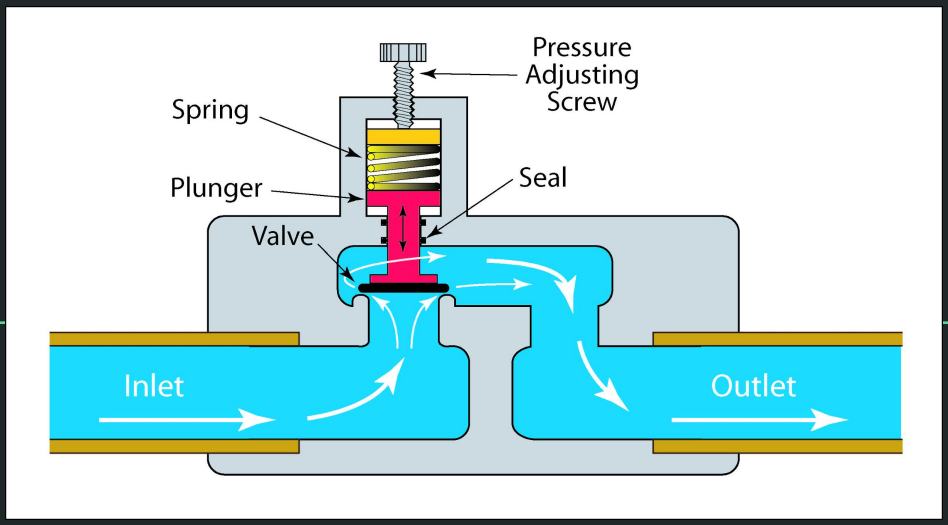
- GenStage is a flexible elixir tool to compose functions while restricting the concurrency of different stages of your data pipeline
- GenStage is **NOT** the right tool for every situation, but we find it pretty handy around SalesLoft
- Checkout the Plataformatec tool called Broadway, which is a wrapper around genstage

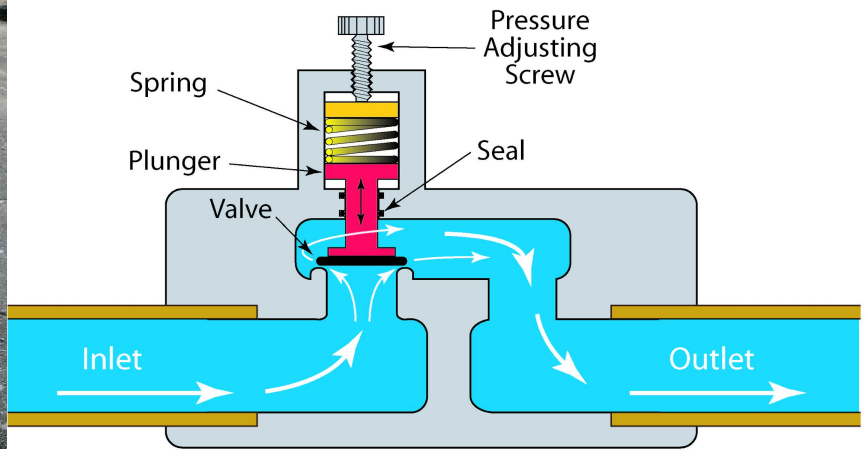


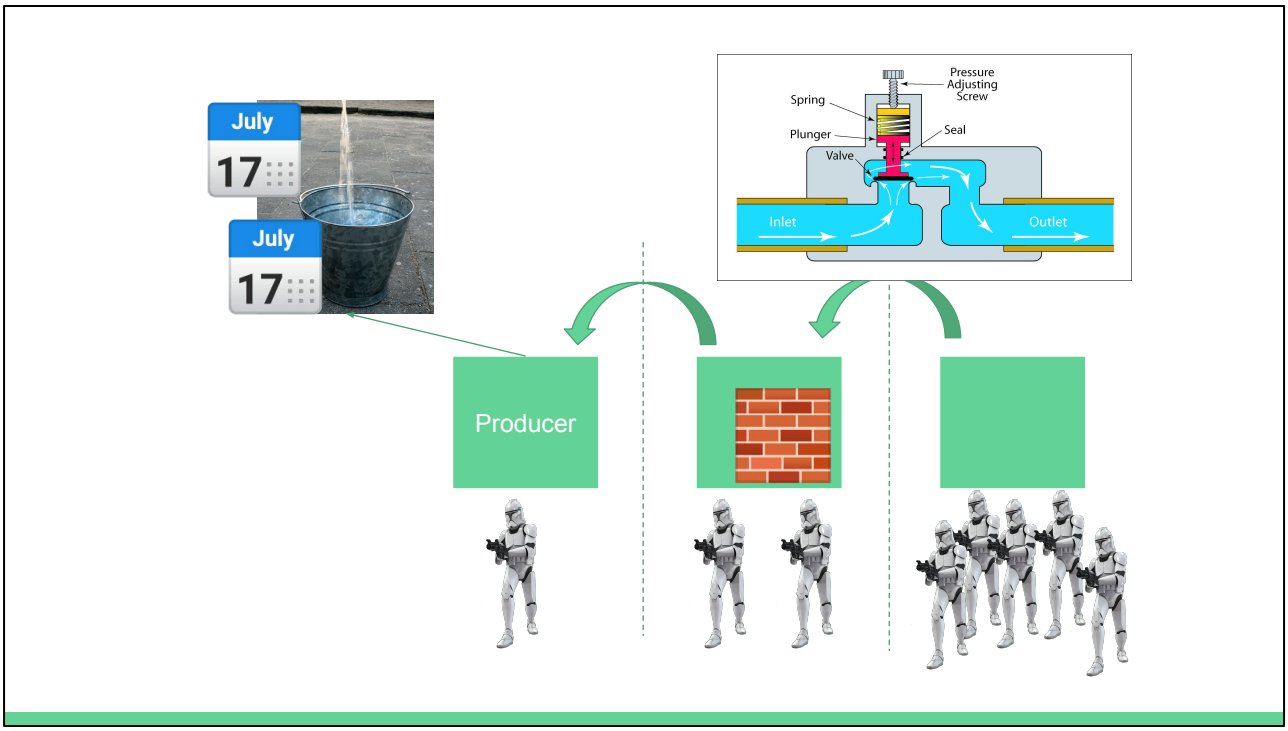
- Add new stages easily over time
- Control the concurrency of any given stage (1, 2, 5)
- Later stages **ASK** earlier stages for work, this provides back pressure to the system
- Producers can get a bunch of appointments at a time, and can be getting appointments while workers are working

Queues Vs. Back Pressure

- Did back-pressure with serial
- Essentially did Queues with Task.async
- https://ferd.ca/queues-don-t-fix-overload.html


Pressure
Adjusting
Screw
Spring
Plunger
Seal
Valve
Inlet
Outlet


Pressure
Adjusting
Screw
Spring
Plunger
Seal
Valve
Inlet
Outlet



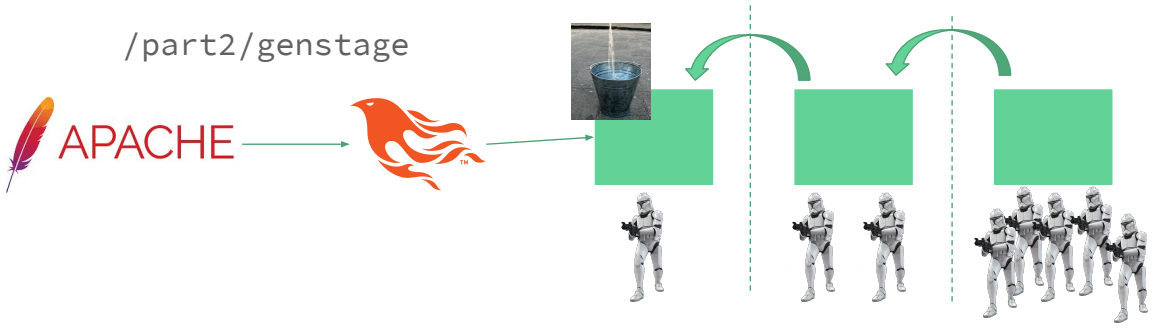
- Producer can be a queue (redis, database, memory)
- Backpressure is provided by the consumers

Measurement is Crucial

- If you set the bucket up and never look at it, how do you know if it's full?
- Errors or shutoff mechanisms when full
- Or you are dropping some data

Example 2

Example 2



- Build off the previous code

Example 2

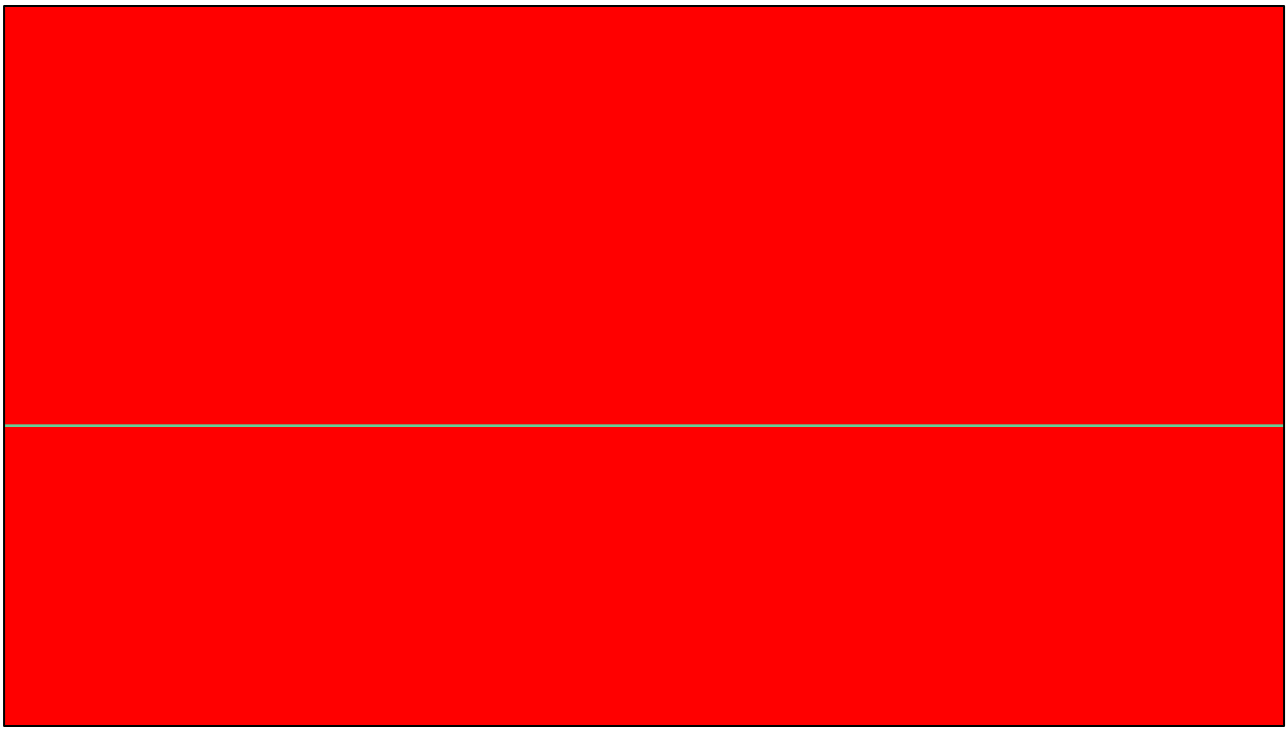
- Repo available at bit.ly/elixir-conf-training-1
- Pair up and ask questions as needed!

Share with Us

- Invite someone up to discuss what occurred for them

Our perspective:

- Same endpoint speed as Task.async (fast web requests)
- Can’t know about the end result of the data
- GenStage implementation was fairly little code compared to what it did for us
- It allowed for predictability in how many jobs would process at once (10 in our case)



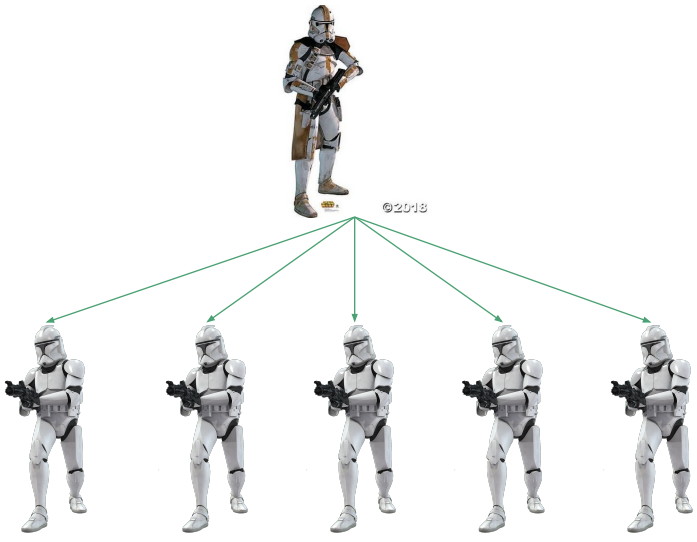
Supervision

Links and Monitors

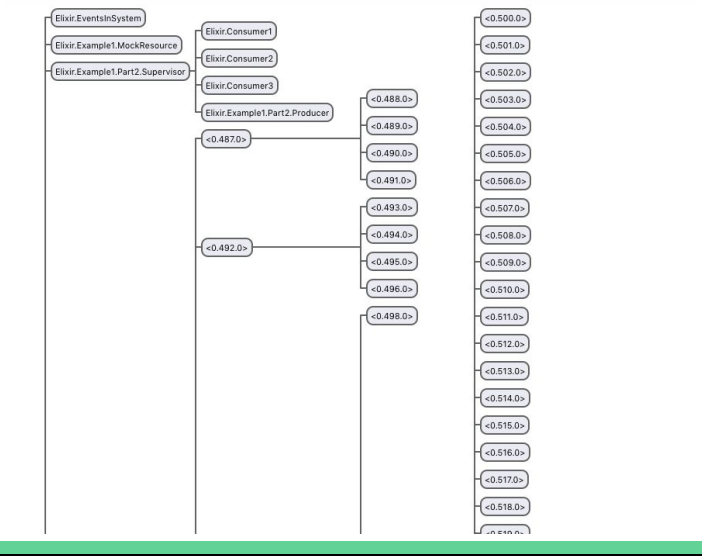
Link - If two processes are linked, if one dies, the other dies

Monitor - A process monitors another process, if the monitoree dies, the monitoring process gets a message

Clone Trooper Chain of Command



Observer


Elixir.EventsInSystem
Elixir.Example1.MockResource
Elixir.Example1.Part2.Supervisor
Elixir.Consumer1
Elixir.Consumer2
Elixir.Consumer3
Elixir.Example1.Part2.Producer
<0.487.0>
<0.488.0>
<0.489.0>
<0.490.0>
<0.491.0>
<0.492.0>
<0.493.0>
<0.494.0>
<0.495.0>
<0.496.0>
<0.498.0>
<0.500.0>
<0.501.0>
<0.502.0>
<0.503.0>
<0.504.0>
<0.505.0>
<0.506.0>
<0.507.0>
<0.508.0>
<0.509.0>
<0.510.0>
<0.511.0>
<0.512.0>
<0.513.0>
<0.514.0>
<0.515.0>
<0.516.0>
<0.517.0>
<0.518.0>

```elixir
defmodule Example1.Part2.Supervisor do
  use Supervisor
  import Supervisor, only: [child_spec: 2]
  alias Example1.Part2.{Producer, Consumer}

  def start_link([]), do: start_link(name: __MODULE__)
  def start_link(name: name), do: Supervisor.start_link(__MODULE__, %{name: name}, name: name)

  def init(_init_arg) do
    children = [
      Producer,
      child_spec({Consumer, name: Consumer1, subscribe_to: [Producer]}, id: Consumer1),
      child_spec({Consumer, name: Consumer2, subscribe_to: [Producer]}, id: Consumer2),
      child_spec({Consumer, name: Consumer3, subscribe_to: [Producer]}, id: Consumer3)
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

Section 2 — Part 1 — Phoenix Channels

-

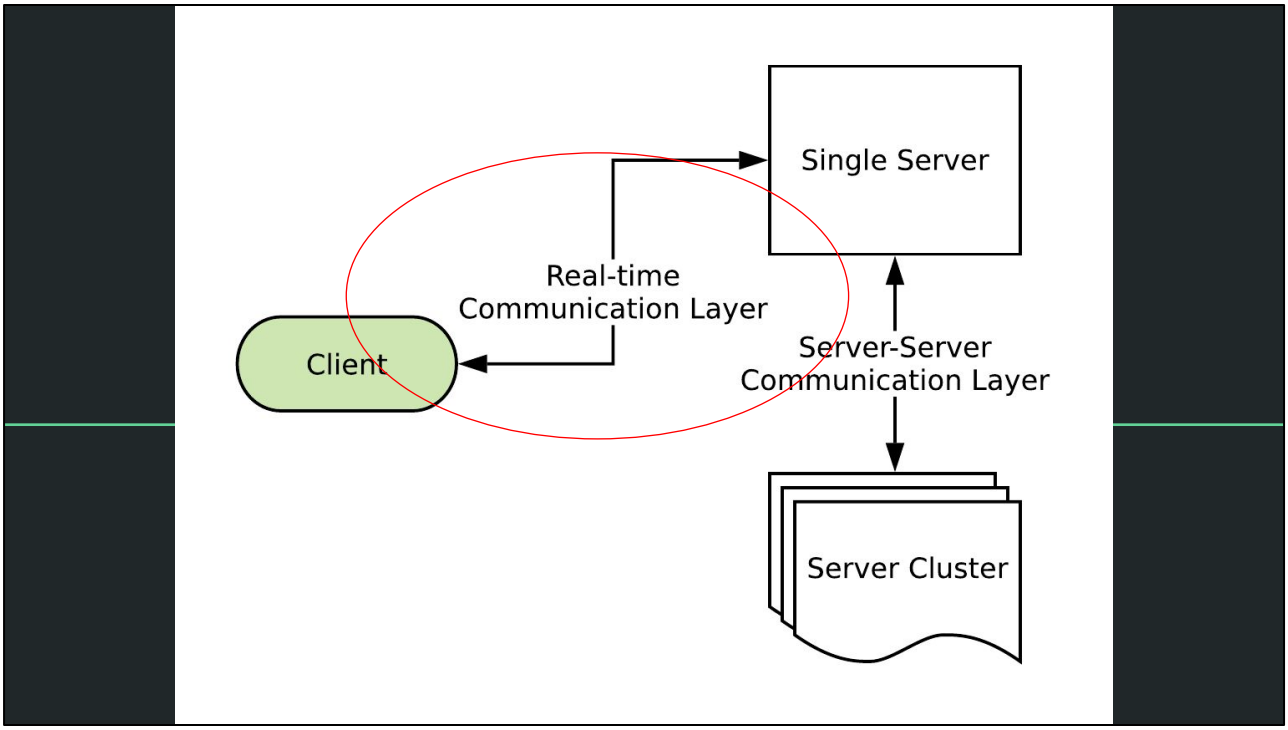
Section 2 Plan

- Structure of Phoenix Channels
- How do Channels fit into a real-time application?
- Use Channels in a demo application
- Activity feed example
- Authentication
- PubSub
- Example pt2

Foundation of Channels

Transition

- In order to get started with Channels, let’s step back and look at a more foundational concept, WebSockets.



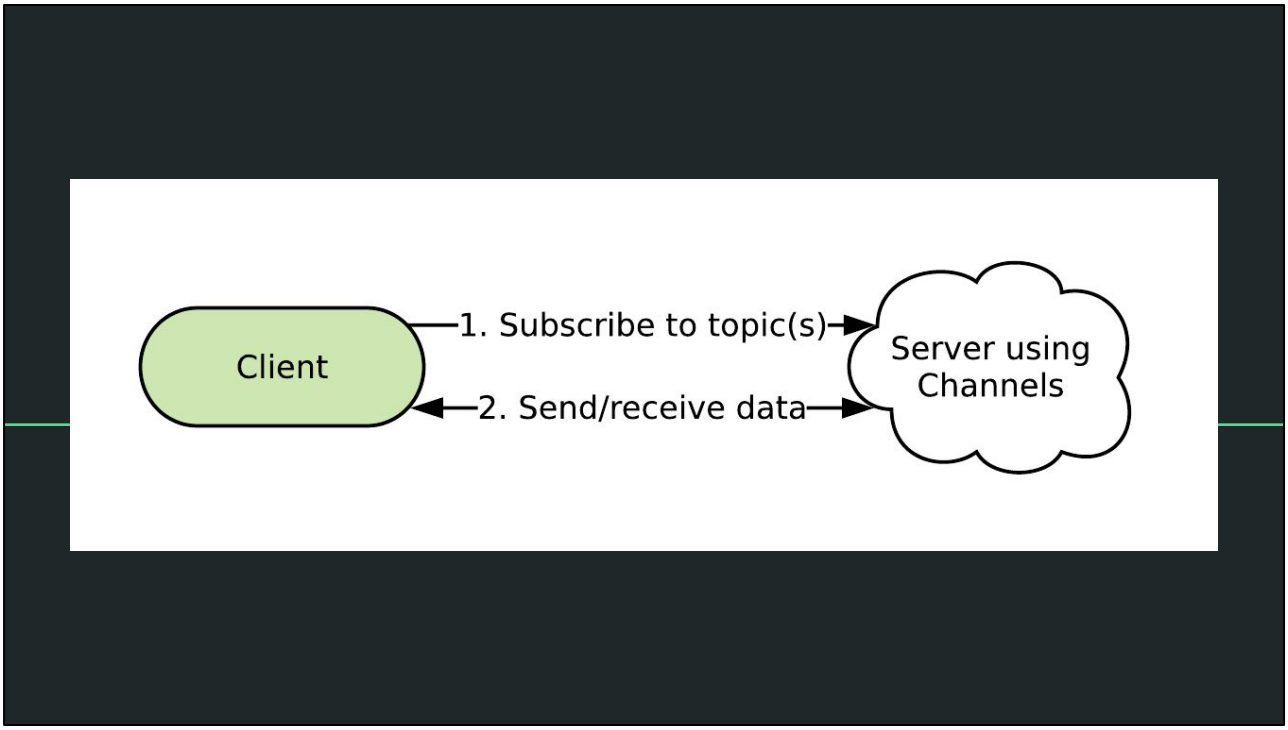
- WebSocket is a real-time communication mechanism
- Sits between the client and a server, providing “last mile” of data to the customer

WebSockets 101

- 2-way “full-duplex” communication channel
- HTTP initiation request becomes a regular TCP socket, in order to work with web products (like load balancers)
- Well supported in 2019
- More scalable than past real-time communications, like long-polling (comet)

What are Phoenix Channels?

- Abstraction for writing real-time applications
- Transport agnostic, but WebSocket is #1 right now
- Highly Scalable and easy to get started with

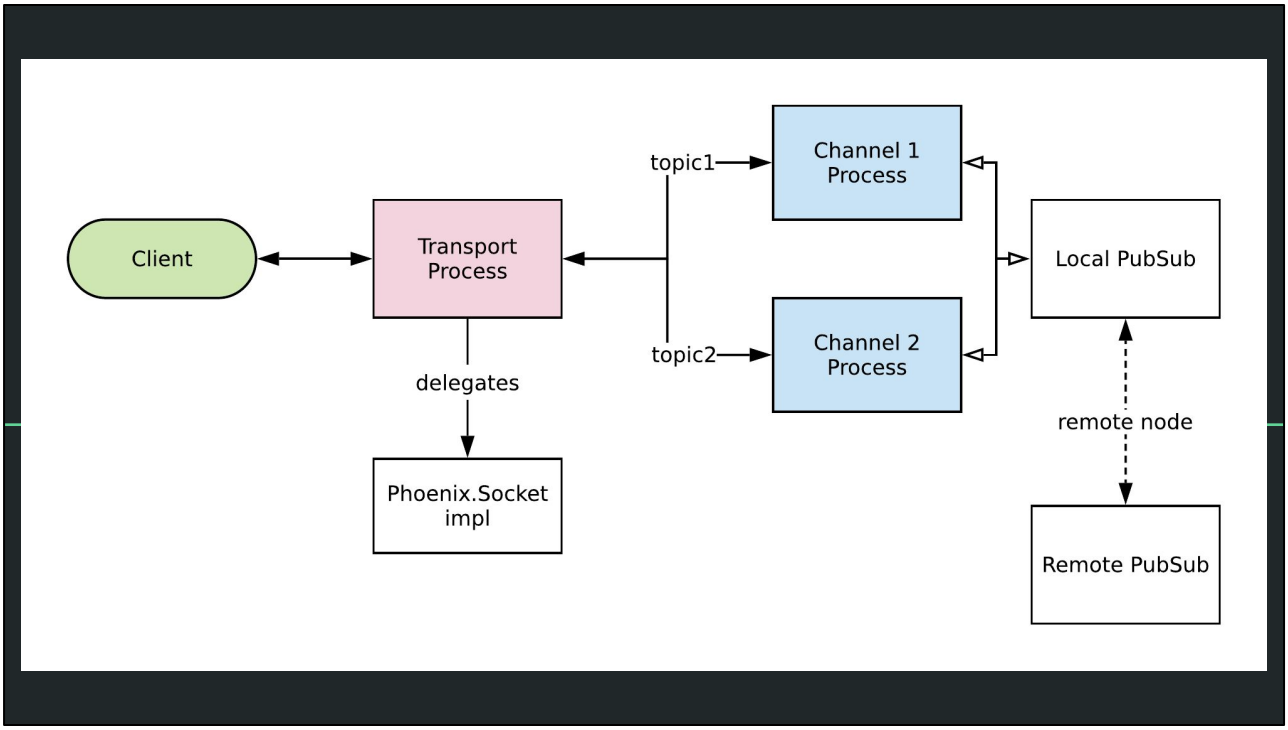

Client
1. Subscribe to topic(s)
2. Send/receive data
Server using
Channels

Why not use WebSockets directly?

- Channels are an abstraction that makes building business applications easier
- WebSockets are a tool for 2-way communication
- You would end up building what Channels gives you
- Channels take advantage of OTP structure really well

Channel Structure

- Transition slide



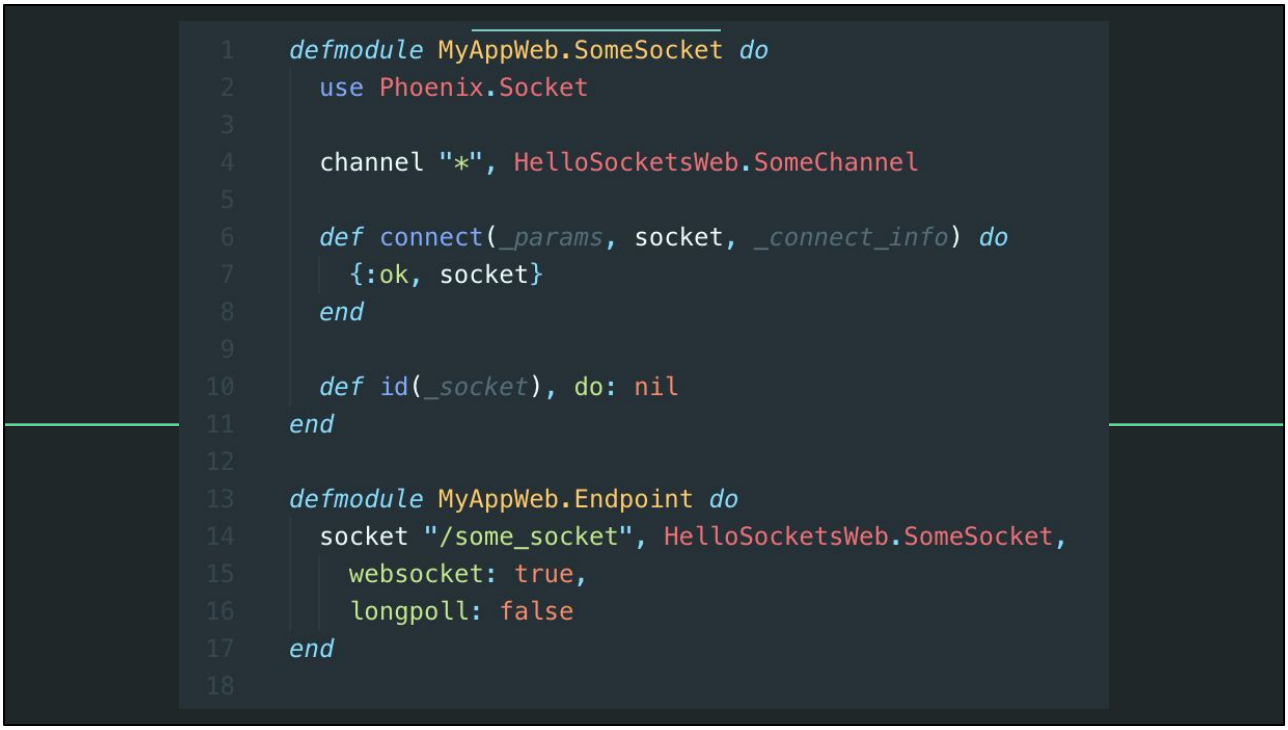
- Client
- Socket Transport
- Channel Processes
- PubSub Distribution
- Transport Process = Phoenix.Socket, but Socket is not a “real process”, it’s delegated to from the Transport process
- Topics are just strings
- Channels directly are processes
- Really elegant solution with OTP, think about how this might be represented in a non-OTP world

How do Channels fit in?

- Edge of data pipeline (to user)
- User data coming into system
- Can hold business flow state (although maybe not the best idea)

Connect a Socket

- Pretty easy to get a new Socket setup with Phoenix. We just have to implement a module with a few functions



```elixir
defmodule MyAppWeb.SomeSocket do
  use Phoenix.Socket

  channel "*", HelloSocketsWeb.SomeChannel

  def connect(_params, socket, _connect_info) do
    {:ok, socket}
  end

  def id(_socket), do: nil
end

defmodule MyAppWeb.Endpoint do
  socket "/some_socket", HelloSocketsWeb.SomeSocket,
    websocket: true,
    longpoll: false
end
```

- Define Socket
- Configure Endpoint
- Define connect/3
- Define id/1
- Add Channels
- You would add authentication or monitoring in the connect/3 function
- You would identify the Socket so you can control from the server (kill all instances of user:1 socket)

Join a Channel

- Like Sockets, it’s easy to get a new Channel setup by implementing a few functions in a module

```elixir
defmodule MyAppWeb.SomeChannel do
  use Phoenix.Channel

  def join("ping", _payload, socket) do
    if valid?(socket) do # optional
      {:ok, socket}
    else
      {:error, %{reason: "an error"}}
    end
  end
```

- Create your Channel module
- Add to your Socket as a route
- Define join/3 for a topic
- Use pattern matching!
- Add authorization here

Receive messages from clients

- transition

```elixir
def handle_in("ping", %{"ack_phrase" => ack_phrase}, socket) do
  {:reply, {:ok, %{ping: ack_phrase}}, socket}
end

def handle_in("ping", _payload, socket) do
  {:reply, {:ok, %{ping: "pong"}}, socket}
end

def handle_in("pang", _payload, socket) do
  {:stop, :shutdown, {:ok, %{msg: "shutting down"}}, socket}
end
```

- Define handle_in/3 callbacks
- Use pattern matching!
- Quickly move into “application code”

Send messages to clients

- transition

```elixir
# Anywhere in your app
MyAppWeb.Endpoint.broadcast("topic", "message", %{data: "test"})

# In Channel
intercept ["message"]

def handle_out("message", payload, socket) do
  push(socket, "message", payload)
  {:noreply, socket}
end
```

- Use Endpoint.broadcast/3 to push data
- Data sent to a topic is automatically pushed
- Intercept + handle_out/3 for custom logic
- Talk about handle_out being the only place to **know** that a client is being sent a message
- handle_out on a large topic will be

- less performant due to JSON payload serialization

Channels are just processes

- A Channel can process handle_info, even GenServer calls if you had such a use case
- Lets you build very versatile flows
- Sockets are not as versatile because they are delegated to

Channels from JavaScript

- Official JavaScript client handles a lot of challenges for you, definitely worth using!
- Works great from traditional web pages, React components, Angular, mobile, pretty much all modern JavaScript

```
1  // Join a Socket / topic easily
2  import { Socket } from "phoenix"
3
4  const socket = new Socket("/socket", {})
5
6  socket.connect()
7
8  const channel = socket.channel("ping")
9
```

- Connect to a socket / channel

```
// Join a Channel, handling any errors
channel.join()
  .receive("ok", resp => { console.log("Joined ping", resp) })
  .receive("error", resp => { console.log("Unable to join ping", resp) })

// Send a data, handling errors or timeouts
channel.push("ping")
  .receive("ok", resp => console.log("receive", resp.ping))
  .receive("error", resp => console.error("an error", resp))
  .receive("timeout", resp => console.error("a timeout", resp))
```

- Join a Channel, send data to the server

```
// Receive messages from the server and process them
channel.on("send_ping", (payload) => {
  console.log("ping requested", payload)
  channel.push("ping")
    .receive("ok", resp => console.log("ping:", resp.ping))
})
```

- Receive new events from a server

Channels Example

-

Getting Started

- Download code from bit.ly/elixir-conf-training-part2
- Follow part1 only
- Complete Channels tasks listed
- Pair up, raise hands with any questions

-

Share with Us

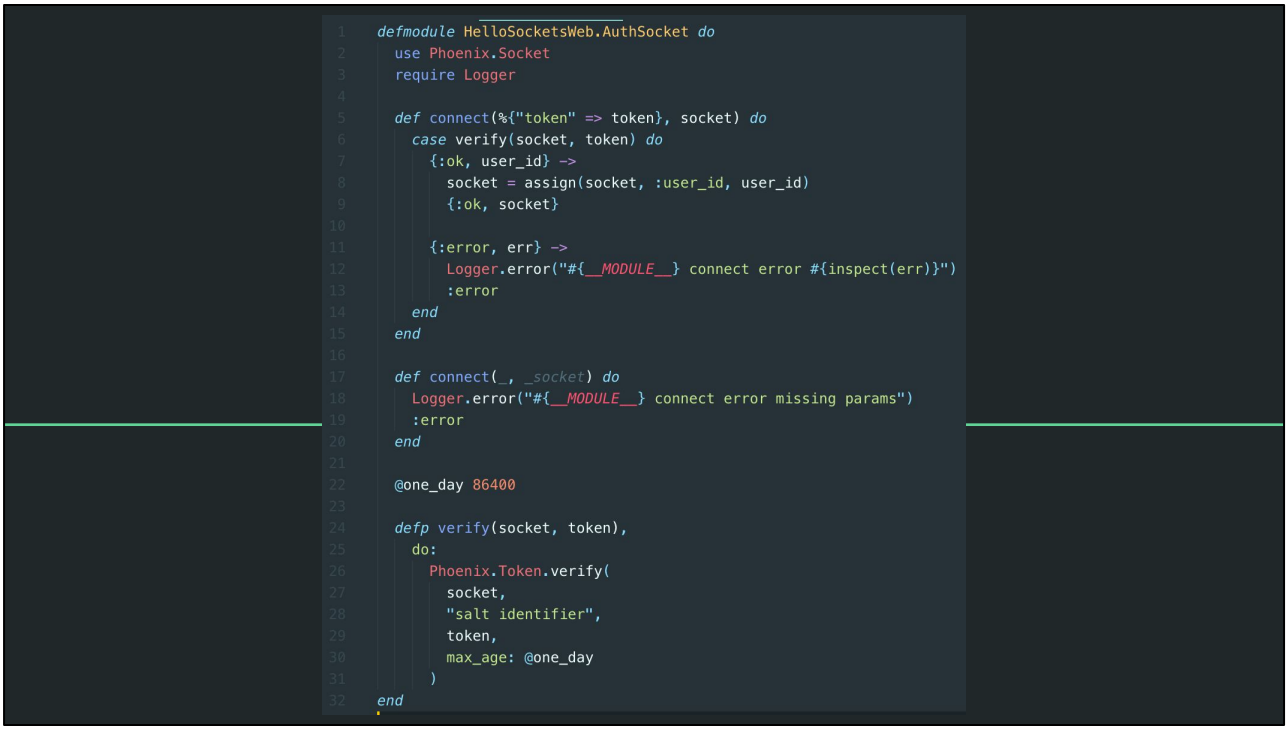
- Invite someone up to discuss what occurred for them

Section 2 — Part 2

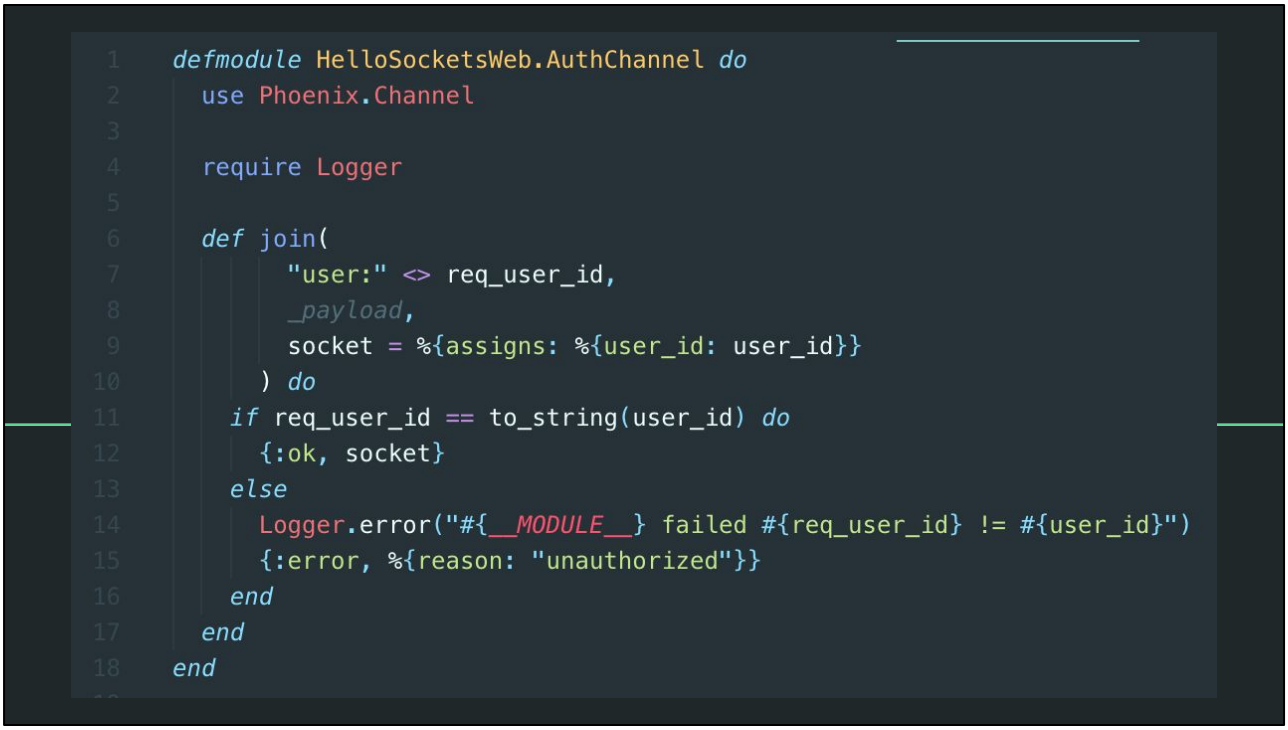
-

Authenticate your Sockets/Channels!

- Super important, but easy to overlook
- Can often *not* use the same authentication mechanism as your HTTP requests, because WebSocket requests are not as secure
- Don’t use cookies because of this, use Phoenix.Token or JWT



```elixir
defmodule HelloSocketsWeb.AuthSocket do
  use Phoenix.Socket
  require Logger

  def connect(%{"token" => token}, socket) do
    case verify(socket, token) do
      {:ok, user_id} ->
        socket = assign(socket, :user_id, user_id)
        {:ok, socket}

      {:error, err} ->
        Logger.error("#{__MODULE__} connect error #{inspect(err)}")
        :error
    end
  end

  def connect(_, _socket) do
    Logger.error("#{__MODULE__} connect error missing params")
    :error
  end

  @one_day 86400

  defp verify(socket, token),
    do:
      Phoenix.Token.verify(
        socket,
        "salt identifier",
        token,
        max_age: @one_day
      )
end
```



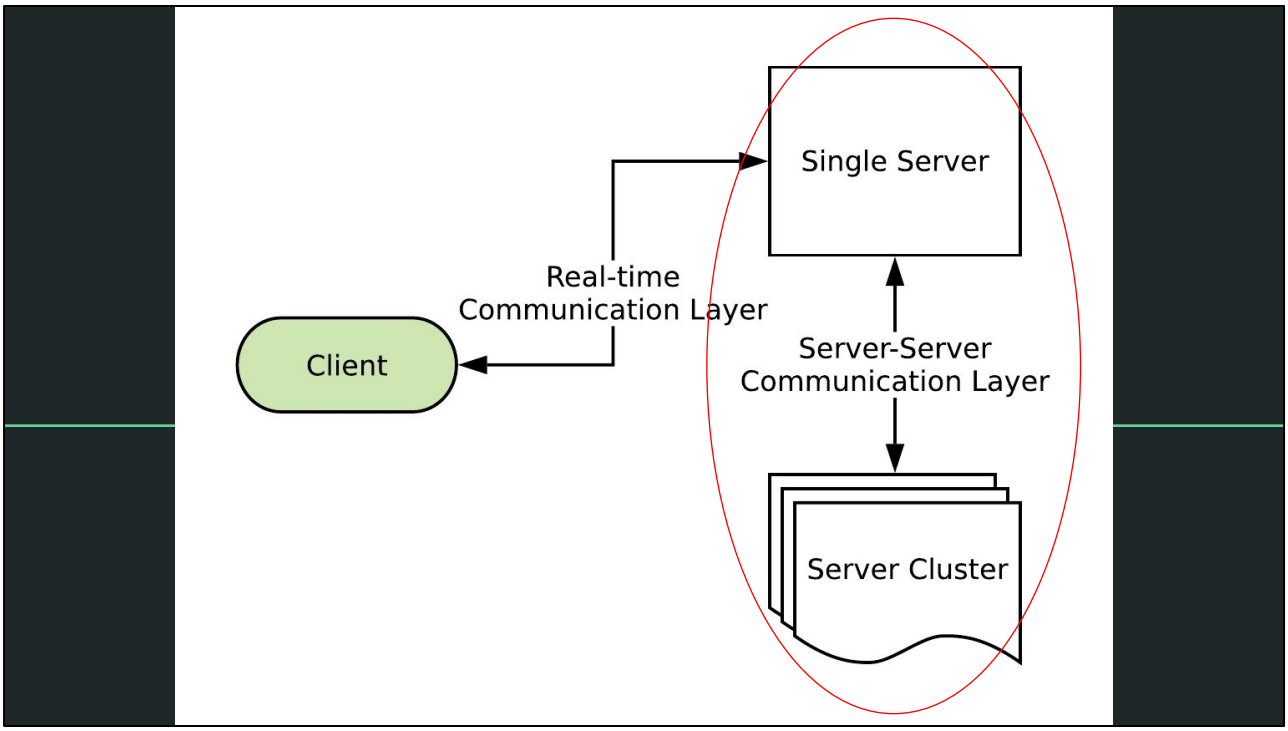
```elixir
defmodule HelloSocketsWeb.AuthChannel do
  use Phoenix.Channel

  require Logger

  def join(
        "user:" <> req_user_id,
        _payload,
        socket = %{assigns: %{user_id: user_id}}
      ) do
    if req_user_id == to_string(user_id) do
      {:ok, socket}
    else
      Logger.error("#{__MODULE__} failed #{req_user_id} != #{user_id}")
      {:error, %{reason: "unauthorized"}}
    end
  end
end
```

Socket vs Channel Authentication

- Use both, different purposes
- Socket authentication can prevent access to the connection at all, which is good for your resources
- Channel authentication can prevent unauthorized access to a sensitive channel, such as a private user channel
- I like to name channels like **private-user-1** or **public-app** so it’s exceptionally clear

Phoenix PubSub

- Transition slide


Single Server
Real-time
Communication Layer
Client
Server-Server
Communication Layer
Server Cluster

Phoenix PubSub

- Communicates between the nodes of the system using pg2 or redis
- Necessary because a user can be connected to server A but new real-time data originates on server B, B needs to get the data to A
- By default, PubSub will use a mesh-like broadcast pattern. Every node gets each broadcast
- Prefer pg2 if you're able to configure node networking, may not be possible depending on your environment
- Redis as a backup if you can't use pg2

Delivery guarantees of Channels

- At most once
- Broadcasts out events over PubSub and doesn’t try any further
- This is good for most people
- You can customize with your own code to handle that, if needed

Channels are just Processes

- Super duper important, re-iterated twice in this section
- You can link or monitor Channels to other parts of your system
- You can send messages to a Channel including from itself
- You can store state in the Channel memory

```elixir
defmodule MyAppWeb.RecurringChannel do
  use Phoenix.Channel

  @send_after 5_000

  def join(_topic, _payload, socket) do
    schedule_send_token()
    {:ok, socket}
  end

  defp schedule_send_token do
    Process.send_after(self(), :send_token, @send_after)
  end

  def handle_info(:send_token, socket) do
    schedule_send_token()
    push(socket, "new_token", %{token: new_token(socket)})
    {:noreply, socket}
  end

  defp new_token(socket = %{assigns: %{user_id: user_id}}) do
    Phoenix.Token.sign(socket, "salt identifier", user_id)
  end
end
```

- Here we're using Process.send_after to start a flow once a Channel is joined
- We can push data to the client from a message handler in the Channel

Channels are ephemeral

- Channel Processes linked to the Socket
- If the Socket dies, the Channels also die
- Internet connections are flaky, so clients will come and go unexpectedly

How to handle ephemeral Channels

- Utilize a persistent source of truth that the Channel interacts with, such as a database, for business data
- Create a functional core that maintains boundaries between the communication layer and the business logic, like in Designing Elixir Systems with OTP by James and Bruce
- Consider the lifecycle of any processes linked to or otherwise associated with your Channel process

Advanced Channels Example

-

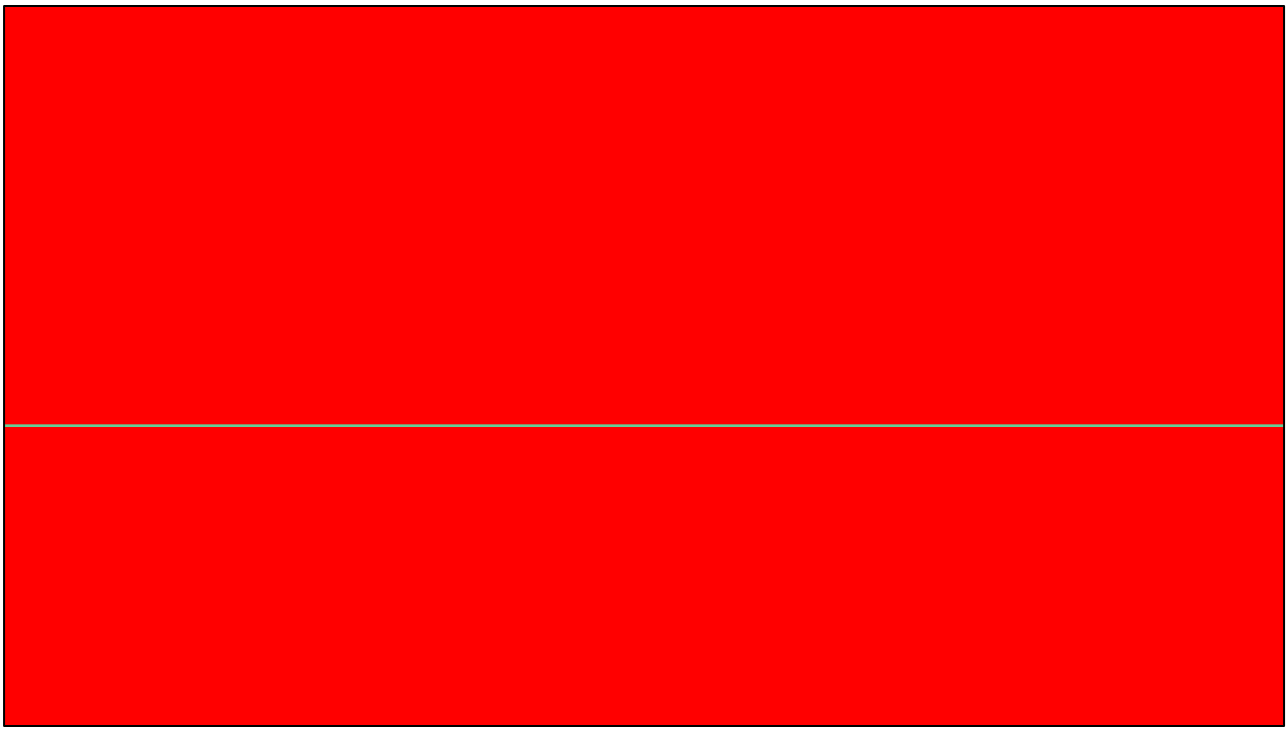
Getting Started

- Same repo as last time
  bit.ly/elixir-conf-training-part2
- Follow part2 only
- Complete Channels tasks listed
- Pair up, raise hands with any questions

-

Share with Us

- Invite someone up to discuss what occurred for them



Q&A Session

20 - 30 Minute AMA with Steve and Grant. Goal is to talk on any topics that the audience has questions on

Section 3 — Part 1

-

Part 1 Goals

- Understand telemetry
- Dispatch metrics to StatsD
- Understand server resource usage
- Understand Garbage Collection
- Example

Telemetry

- Standardized way to emit and collect metric events in Erlang/Elixir applications
- Set of tools to aggregate and emit metric events to different tools
- Erlang Foundation supported direction for the future of metrics on the BEAM

Finding Available Telemetry Metrics

- Many libraries will document what metrics they support
- Can **grep -R “telemetry\.execute” deps** in an Elixir project

Types of TelemetryMetric Events

- Counters
- Counts at points in time
- Timing of an operation

- TelemetryMetrics allows aggregating of events
- Telemetry is just a way to invoke functions when events run, so there's not "type" of metric for it

Producing StatsD Events with Telemetry

- Supported library is TelemetryMetricsStatsD
- Integrate telemetry based stats very quickly

Server Resource Usage

- Transition slide

Memory

- Long running processes can be very susceptible to memory leaks
- You have to monitor your memory usage in order to be confident that there's no issue over time

CPU

- Generally, CPU is not a big issue for this type of application
- Pre-emptive scheduler will prevent a CPU heavy operation from hogging the CPU
- If you’re serializing a lot of data or sending it to nodes, you could run CPU pretty hot
- Phoenix Tracker and other distribution techniques may be more CPU intensive

Network

- Non-obvious resource that can be consumed quicker than expected
- Mesh networking approach leads to linear network bandwidth growth to support the same amount of traffic
- Bigger, fewer servers may help keep this low

Understanding BEAM Garbage Collection

- Each process gets its own garbage collection life cycle
- Processes can share “large binaries” which uses a reference based counting GC lifecycle. There’s no stop the world pauses across the whole system.
- An individual process has both minor and major GC life cycle

Generational GC (minor)

- Happens often, only on the “young heap”
- Young heap is objects allocated since last minor GC
- An item that survives a generational GC moves into the “old heap” which is less likely to be garbage collected in the near term

Fullsweep GC (major)

- Happens less often (sometimes much less often)
- Catches all garbage, which is more expensive
- Can end up getting stuck in a sweet spot of extra memory usage, due to the non-linear growth of the memory allocation

Memory with Channels

- New OTP concept called “hibernation” is used automatically by Channels
- This forces Channels into a major GC, low-memory state until it is re-hydrated when a message comes in
- This can significantly reduce memory overhead
- 4k vs 50k memory per Channel is 12x more memory for the same amount of users
- Hibernation is not automatic for your GenServers. Be careful for long-life GenServer memory

Telemetry Example

-

Getting Started

- Same repo as last time
  bit.ly/elixir-conf-training-part2
- Checkout branch section3-part1-start
- Complete Telemetry tasks listed
- Pair up, raise hands with any questions

-

Share with Us

- Invite someone up to discuss what occurred for them

Section 3 — Part 2

-

Part 2 Goals

- Network nodes together
- Leverage PubSub for Nodes to talk to each other
- Observe production from the CLI

Maintain Asynchronous Processes

- Every Process is serial by design—only 1 message can process at a time
- Avoid having any contention with your processes by making them respond to long duration requests asynchronously
- Biggest hit here is against Channels, because it’s common to fetch external resources from them.

```
14    # Channel
15    def handle_in("work", payload, socket) do
16      result = expensive_lookup()
17      {:reply, {:ok, result}, socket}
18    end
19
```

```elixir
# Channel
def handle_in("work", payload, socket) do
  Worker.perform(payload, socket_ref(socket))
  {:noreply, socket}
end

# Worker
def handle_info({:work_complete, result, ref}, socket) do
  reply(ref, {:ok, result})
  {:noreply, socket}
end
```

Node networking

- Elixir comes with powerhouse networking features out of the box
- Really all you have to do to take advantage is to connect the nodes together
- Elixir is very optimistic about connecting all nodes together. This is fine for most people

Setting up libcluster

- Libcluster is a tool that makes it easy to connect nodes together
- At its core it is just service discovery, the actual Node connection is just a single function call

```
15      children = [
16        {Cluster.Supervisor, [Application.get_env(:libcluster, :topologies), [name: __MODULE__.ClusterSupervisor]]},
```

```
56  config :libcluster,
57    topologies: [
58      main: [
59        strategy: Cluster.Strategy.Kubernetes.DNS,
60        config: [
61          service: EnvMap.fetch!("HEADLESS_SERVICE_DNS_NAME"),
62          application_name: "salesloft_pusher",
63          polling_interval: 10_000
64        ]
65      ]
66    ]
```

Sending Messages

- GenServer.call and cast work across the cluster out-of-the-box
- Need to be careful about network issues while a call is going on (try/catch more often)
- Tools like PG2 help orchestra “where does this process live on a given node?” across the entire cluster

```
110     defp collect_counts_in_cluster(tick, topic) do
111       on_remote_nodes(topic, fn pid ->
112         GenServer.cast(pid, {:collect_request, tick, self()})
113       end)
114     end
115
116     defp on_remote_nodes(topic, func) do
117       topic
118       |> :pg2.get_members()
119       |> Kernel.--(:pg2.get_local_members(topic))
120       |> Enum.map(func)
121       |> length()
122     end
```

Power of PubSub

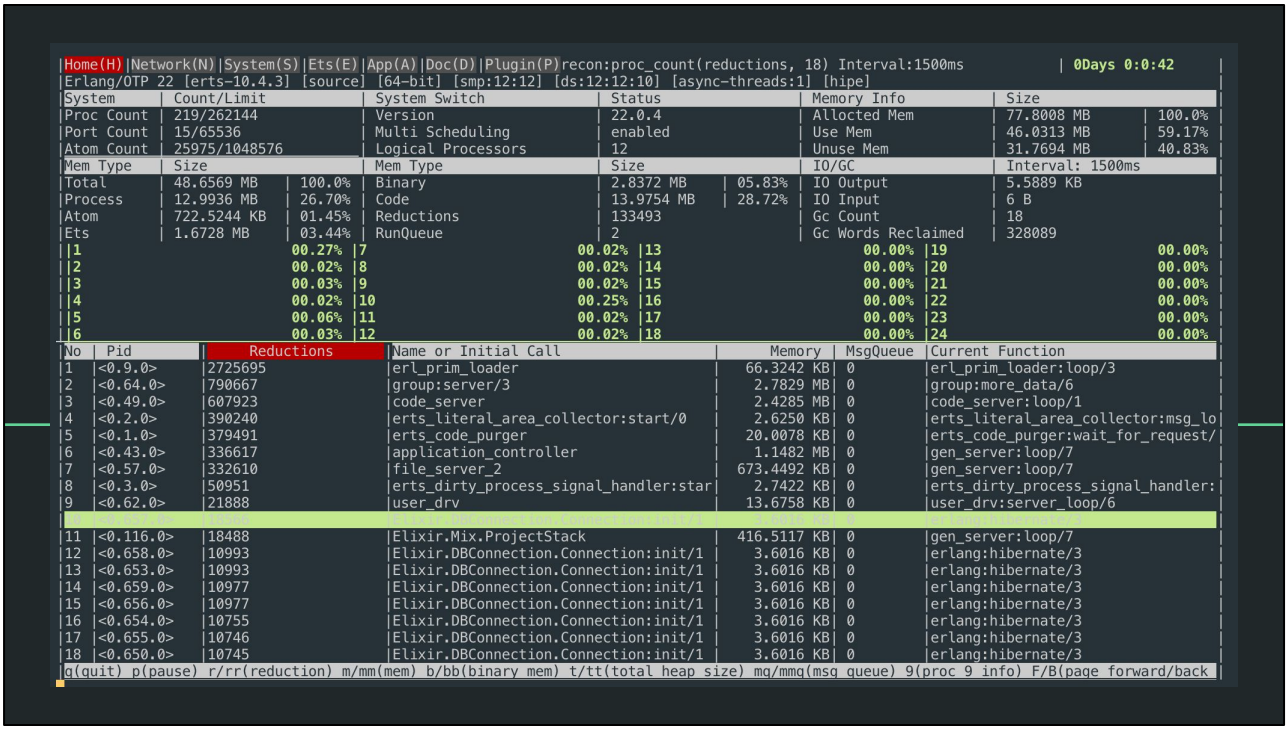
- PubSub encapsulates the pg2 + message passing paradigm into 1 consistent library
- Can swap out the backend (node networking) for other types of communication (redis pubsub)
- Comes already setup on Phoenix apps
- We used it several times today

Observing Production

- We’re used to tools like :observer.start locally, as we saw today
- You can run :observer is production, but it’s much more difficult. Common to not run GUI sessions there
- Non-interactive tools also exist for observation, like StatsD or Application Performance Monitoring tools

Observer CLI to the Rescue

- Super neat library



- Sort by different fields
- View the details of a process
- View Network usages
- Really powerful tool in general

Networking / Observation Example

-

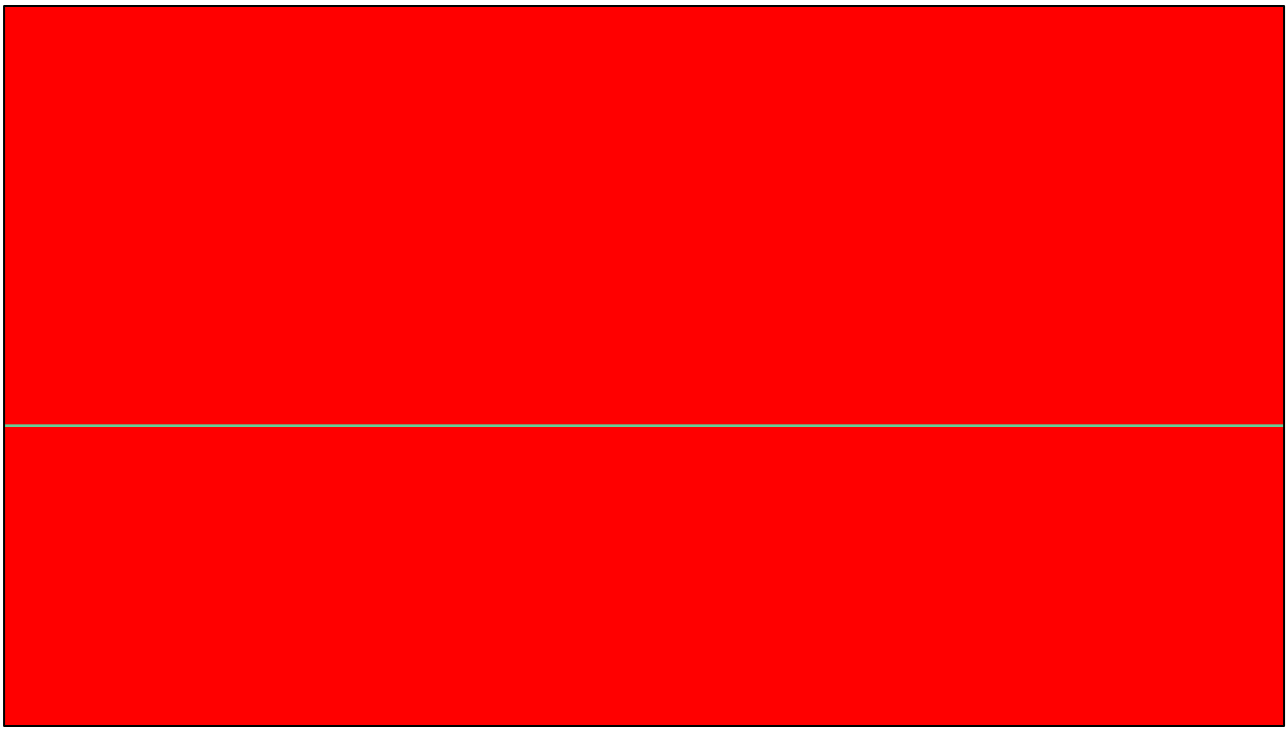
Getting Started

- Same repo as last time
  bit.ly/elixir-conf-training-part2
- Checkout branch section3-part2-start
- Complete tasks listed
- Pair up, raise hands with any questions

-

Share with Us

- Invite someone up to discuss what occurred for them



Section 4 — Part 1

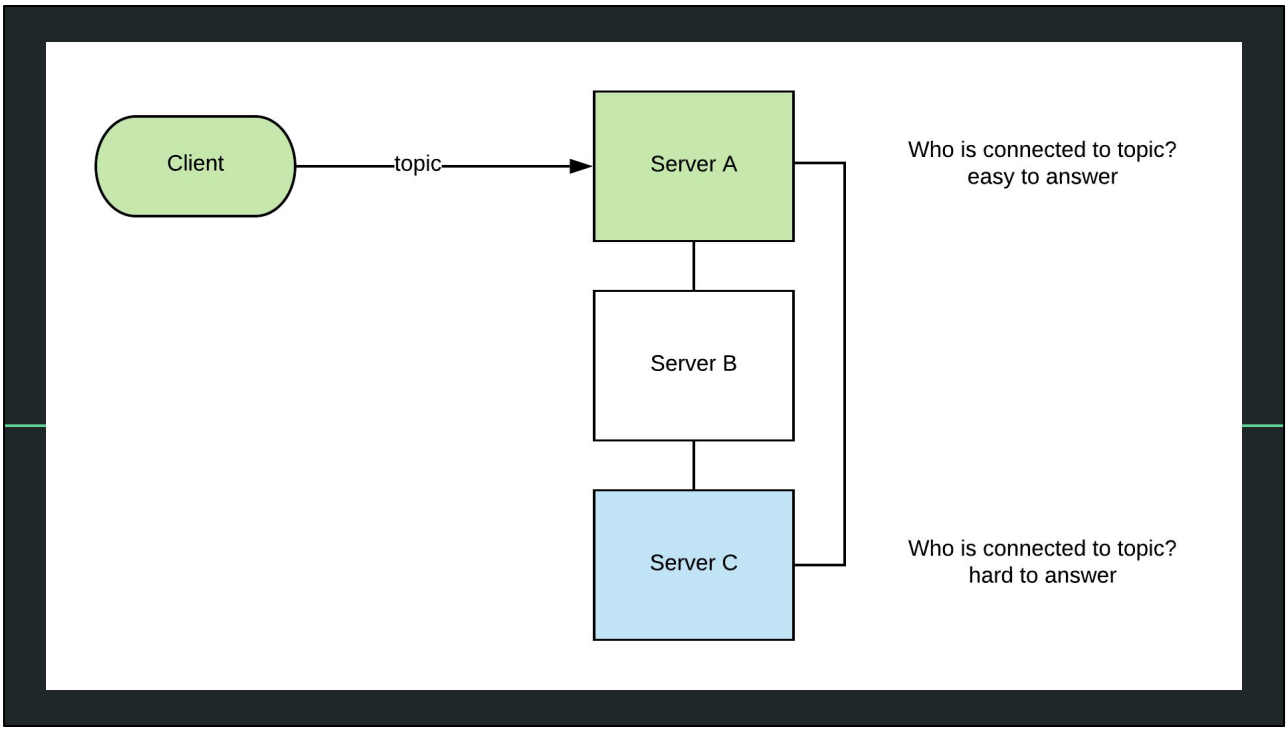
-

Section Plan

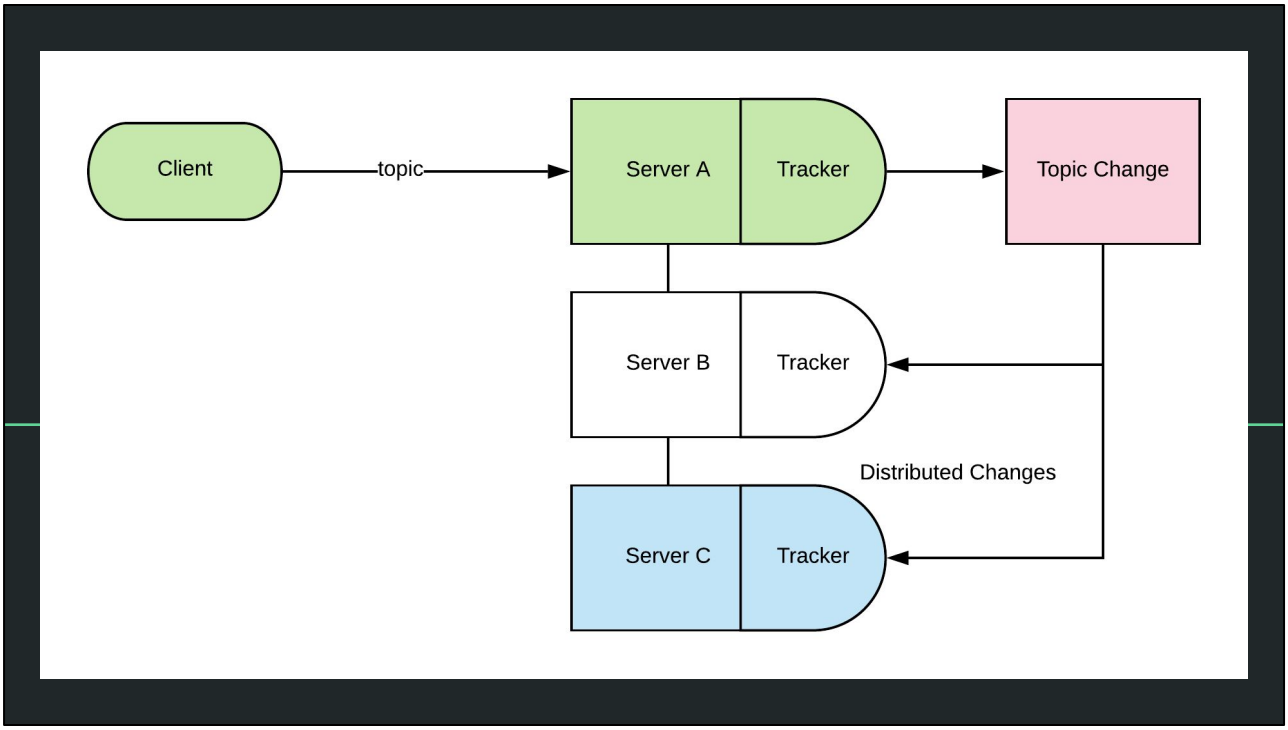
- Understand Phoenix Tracker and Presence
- See how Tracker can fit into a data pipeline
- Challenges of distributed code

What is Phoenix Tracker?

- Tracker solves the problem of knowing what processes are alive across a cluster of servers
- Next slide has illustration



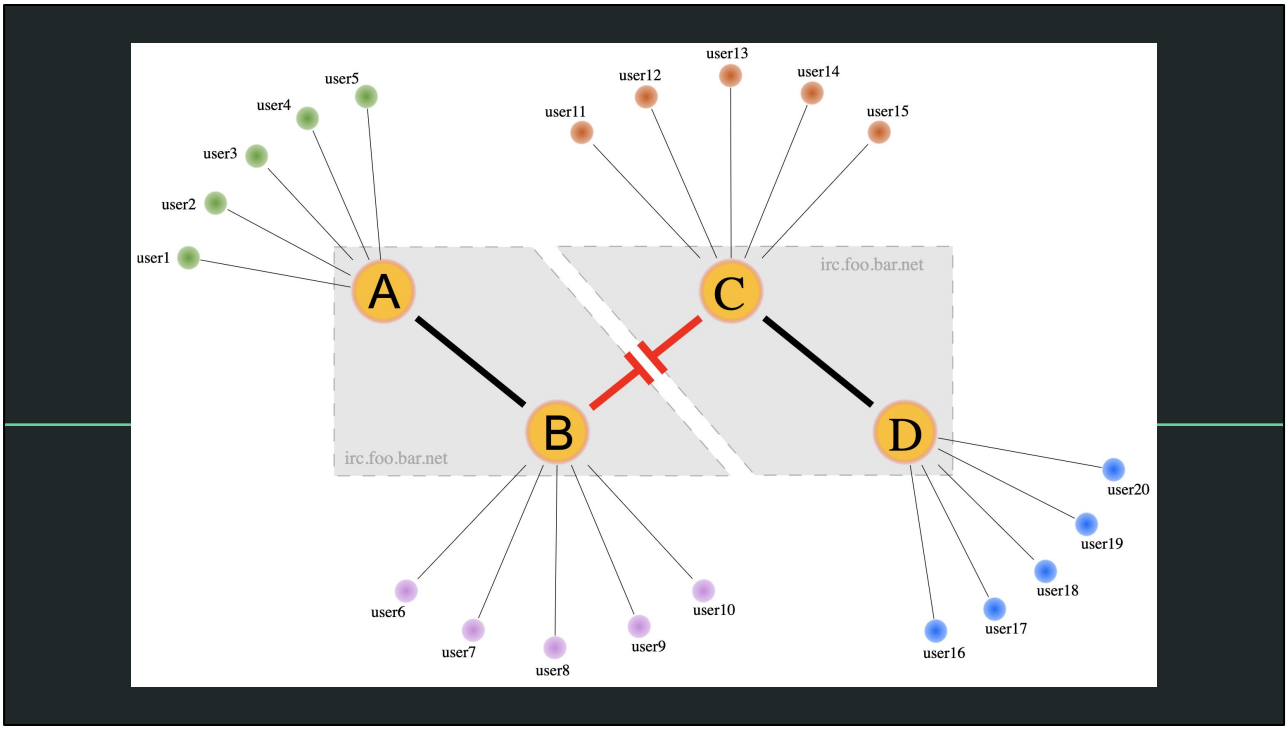
- It’s easy to know if the client is connected when server A asks (just look in memory)
- It’s hard to know if the client is connected when server C asks (has to jump the network)



- Each Tracker receives changes from the topic
- Topic Change = join/leave
- The Tracker can do whatever it wants with those changes (nothing, send them back to clients, custom use cases)
- The question “who is on topic X” becomes much cheaper to answer, just ask the current node

The challenge of distribution

- Let’s look at a scenario that can throw a wrench in basically any distributed problem



- A netsplit occurs when nodes of your cluster are healthy, but cutoff from each other
- This might be temporary, but you'd need to consider any missed messages during that time
- Re-establishing truth ASAP is critical

- Tracker leverages a special data structure called a CRDT in order to deterministically apply join/leave across a cluster
- This approach is very difficult to do otherwise, so lean on tools like Phoenix Tracker that does it for you
- You can always Process.link your Channel processes, but then you end up re-inventing the solutions Tracker has to do

Presence is a special type of Tracker

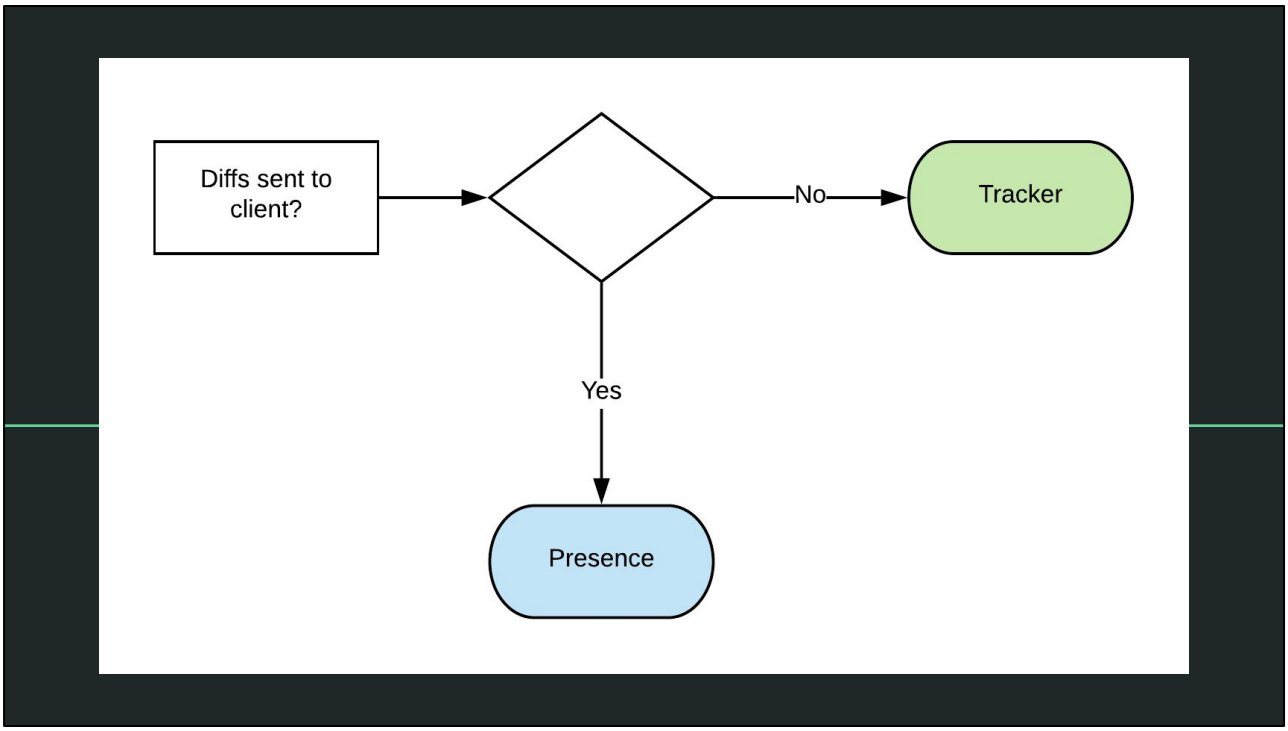
- Steve - Pet peeve to jump to Presence when you probably want Tracker
- Presence is Tracker + an additional feature. That means it is more expensive and you may not even want the additional feature

```
339     def handle_diff(diff, state) do
340       {module, task_supervisor, pubsub_server} = state
341
342       Task.Supervisor.start_child(task_supervisor, fn ->
343         for {topic, {joins, leaves}} <- diff do
344           Phoenix.Channel.Server.local_broadcast(pubsub_server, topic, "presence_diff", %{
345             joins: module.fetch(topic, Phoenix.Presence.group(joins)),
346             leaves: module.fetch(topic, Phoenix.Presence.group(leaves))
347           })
348         end
349       end)
350
351       {:ok, state}
352     end
353   end
```

- Presence is a Tracker that implements handle_diff so that it broadcasts “presence_diff” to all connected channels

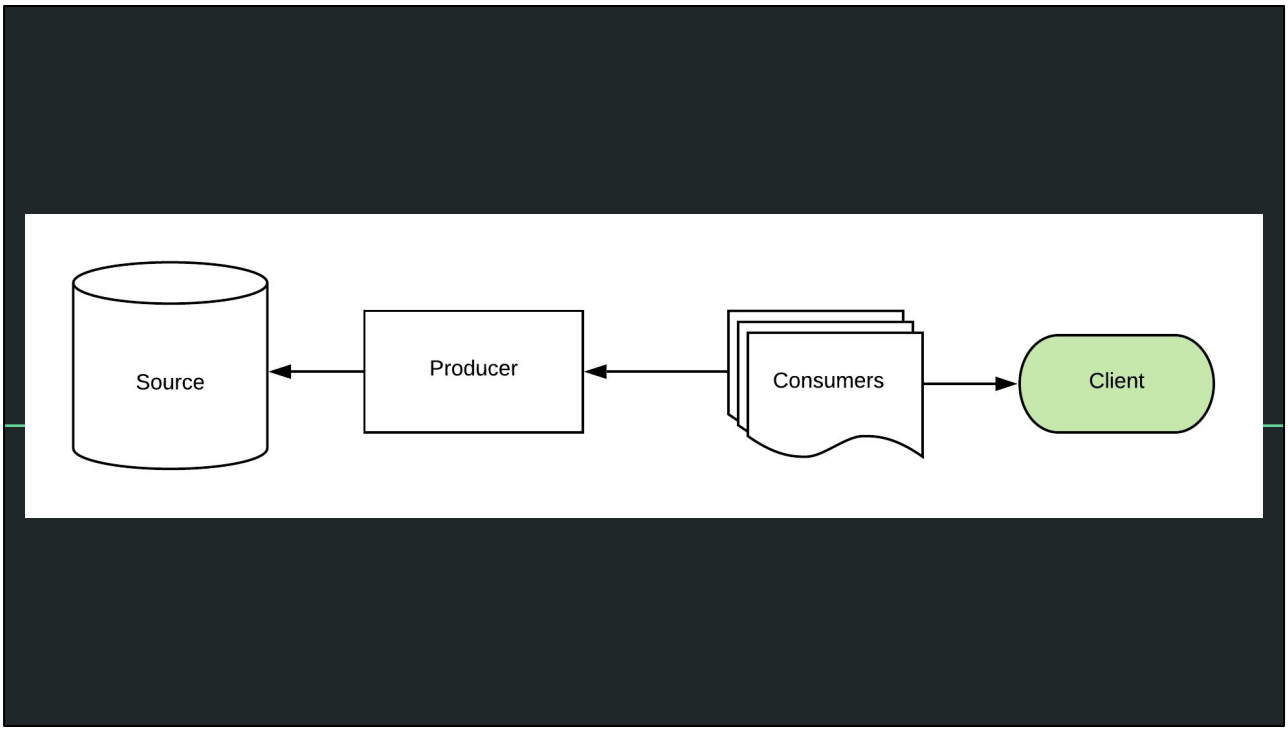
Do you want Presence or Tracker?

- Next slide

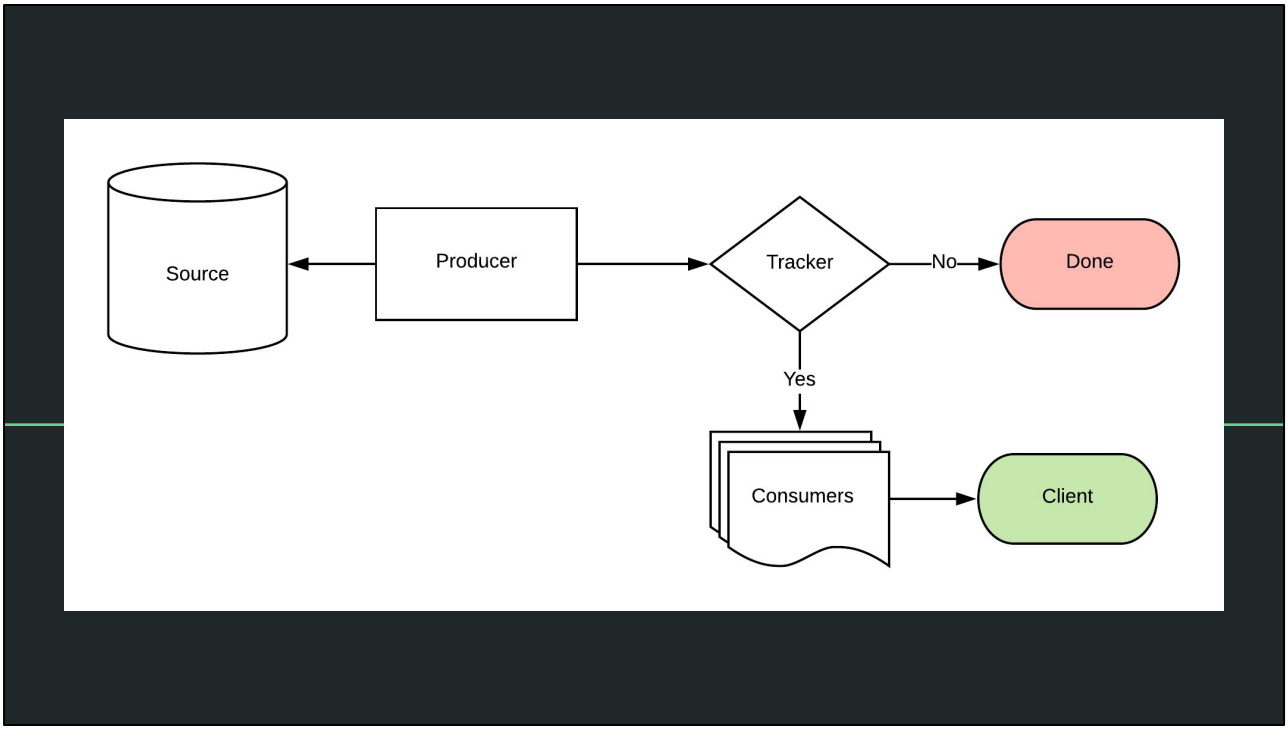


- Really easy question to ask yourself: Do you want all presence diffs being sent to the clients connected to that topic?
- If yes -> Presence. If no -> Tracker

Tracker in a data pipeline



- Data pipeline without Tracker
- Items are produced from a database, ran through multiple consumers, and finally spit out to the destination (a client)
- What if no client is there? -> Then we wasted work / resources



- Tracker lets us ask the question “Is there a client connected for the topic I will be pushing to?”
- If yes, then let’s go ahead and do the work to push the data
- If no, then let’s get out of the data pipeline

Tracker's big impact

- If your topics are only X % utilized during a point in time, you may be able to significantly reduce throughput of the data pipeline
- This has a direct impact on scalability. Less work done = more work available to get done

If your data pipeline has an expensive resource call, consider Tracker to reduce calls to it.

-

Tracker Example

- Configure a Phoenix.Tracker module
- Play around with handle_diff to see what comes back and what you could do with that
- Understand performance of Tracker calls

Getting Started

- Same repo as earlier
  bit.ly/elixir-conf-training-part2
- Checkout **section4-part1-start**
- Complete Tracker tasks listed
- Pair up, raise hands with any questions

-

Share with Us

- Walk through code that powering Phoenix.Tracker module
- Ask about handle_diff and what they saw
- Discuss performance testing of Tracker

Section 4 — Part 2

-

Section Plan

- Understand LiveView
- LiveView compared to Channels
- Example: Implement section 2 as a LiveView

What is LiveView?

- Library built on top of Sockets/Channels that allows you to build interactive frontends without writing JS
- Biggest buzzword in Elixir right now (and at this conference probably)

Foundations of LiveView

- Everything you've learned today about Channels and Sockets applies to LiveView, so it will be nearly instantly familiar
- You can build the same custom-flows on top of LiveView
- Obviously has some differences in the actual features added—user interface

Getting started with LiveView

- Transition

```elixir
defmodule AppWeb.ThermostatLive do
  use Phoenix.LiveView

  def render(assigns) do
    ~L"""
    Current temperature: <%= @temperature %>
    """
  end

  def mount(%{id: id, current_user_id: user_id}, socket) do
    case Thermostat.get_user_reading(user_id, id) do
      {:ok, temperature} ->
        {:ok, assign(socket, :temperature, temperature)}

      {:error, reason} ->
        {:error, reason}
    end
  end
end
```

- Define the module as a LiveView
- Implement a mount/2 callback, this is like join/3 in a Channel. You can do authentication and initial state there
- Implement render/1 function—This is the magic for how the frontend will be rendered
- A template file can be used instead of the string

```elixir
defmodule AppWeb.Endpoint do
  use Phoenix.Endpoint

  socket "/live", Phoenix.LiveView.Socket
  ...
end
```

```elixir
defmodule AppWeb.Router do
  use Phoenix.Router
  import Phoenix.LiveView.Router

  scope "/", AppWeb do
    live "/thermostat", ThermostatLive, session: [:user_id]
  end
end
```

- Add the LiveView socket to your Endpoint, note that there is not an application specific Socket
- Define your live route in your Router (different than a Channel). This is because a LiveView is both a real-time application and a server rendered one

What can LiveView do?

- Transition slide

LiveView offerings

- Process client-side events on the server (button clicks, links, focus, blur, key events, etc)
- Implement business flows in 1 language (form validations, custom “widgets”)
- pushState based “single-page” apps
- Lots more stuff coming, it’s still 0.1.0 release

LiveView release

- Just released into a public 0.1.0 release
- I’d wait a bit for a 1.0.0 release before taking it into a core business application, just from a risk mitigation perspective
- Great for smaller widgets or internal apps

How LiveView fits in with today's class

- LiveView is a continuation of Channels—You can do everything Channels can do, but it becomes easier to get the frontend up and running. They are at the end of your data pipeline
- If you're pushing data to different LiveViews (over PubSub), you would want to think about your data pipeline and how it all fits together
- LiveViews are just processes, just like Channels

LiveView Example

-

Getting Started

- Same repo as earlier
  bit.ly/elixir-conf-training-part2
- Checkout **section4-part2-start**
- Complete LiveView tasks listed
- Pair up, raise hands with any questions

-

Share with Us

-

Q&A Session

Thanks!