

Real Time Systems

About Me



Hi, I'm Steve Bussey

- Software Architect at SalesLoft
- Elixir in production for several years
- I'm excited about Elixir





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

SalesLoft

Companies

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
- 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
- 11/17/17 10:07 AM (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-a584-9ef11ac2c47d>
Edit Call Note
- 11/16/17 4:23 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

Real-Time Phoenix

- Reinforces this course's material
- Available (final beta) at <https://sal.es/rtp>
- LonestarElixir20 for 20% off

Real-Time Phoenix

Build Highly Scalable Systems
with Channels



Stephen Bussey

Series editor: Bruce A. Tate
Development editor: Jacquelyn Carter

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?

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

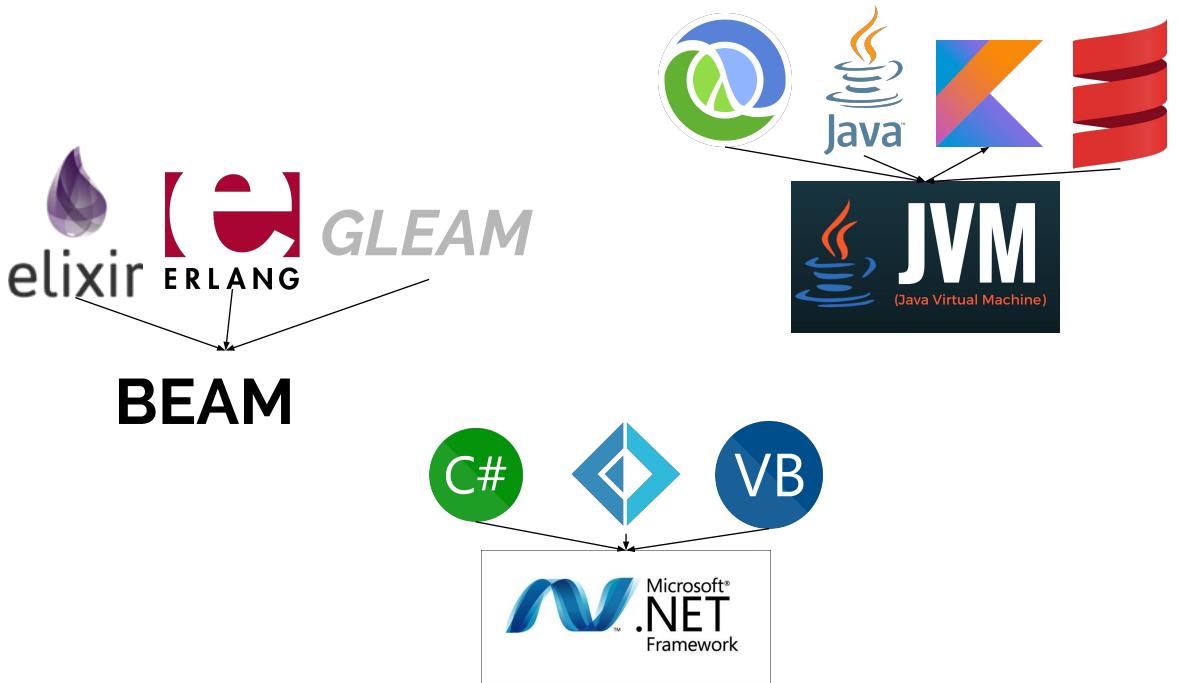
User expectations have risen

- 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



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.

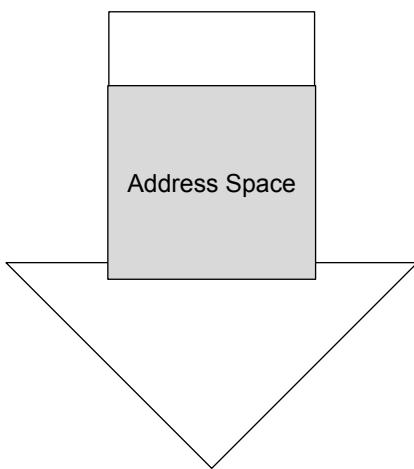
- Process shared nothing architecture is pretty important to BEAM characteristics and performance

Processes

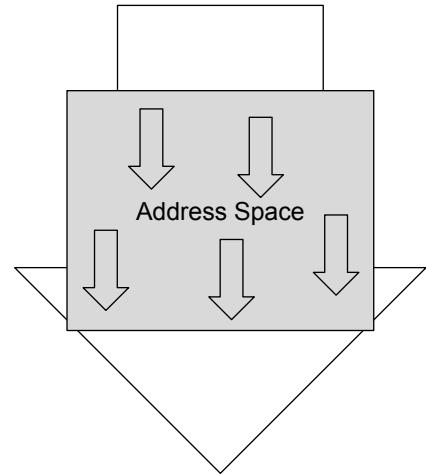
- Beam Processes are not OS processes

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
Processes or OS
Threads

BEAM Processes

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



- BEAM processes are VM provided concurrency primitives 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

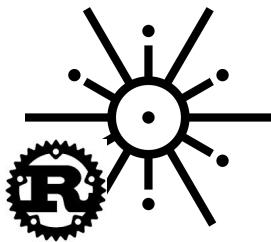


BEAM Processes



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

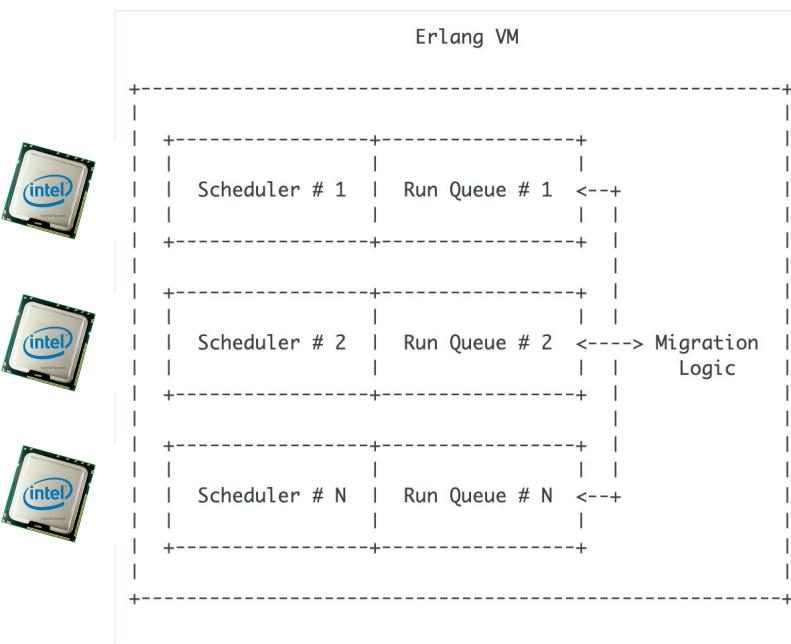
Fast Network I/O and Event Management for Ruby Programmers

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

-

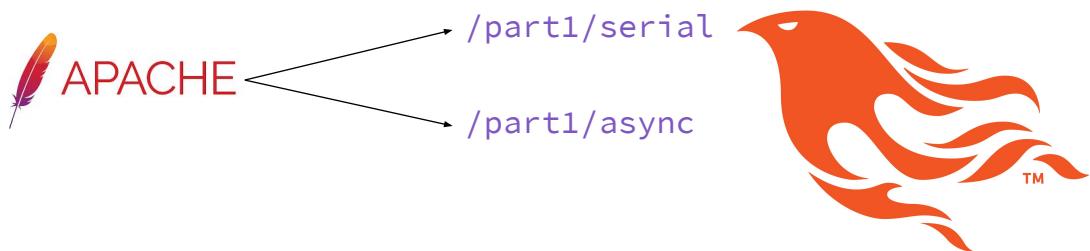


<https://hamidreza-s.github.io/erlang/scheduling/real-time/preemptive/migration/2016/02/09/erlang-scheduler-details.html>

- 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

```
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 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 <http://bit.ly/real-time-training>
Branch section1-part1

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

Caution

The Single Global Process (SGP) is easy to 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

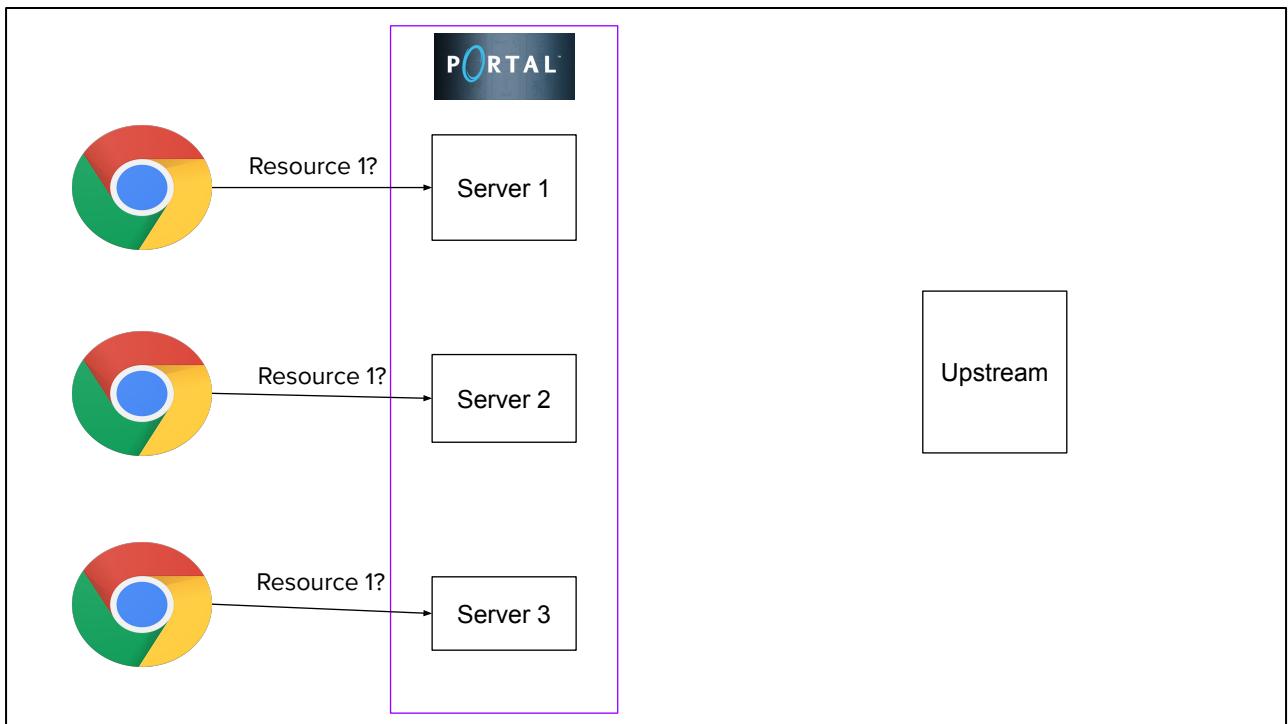
- In the BEAM processes can run in parallel
- Uncontrolled parallelism leads to blowing up downstream services
- Sometimes we have to reign in parallelism in order to be stable—stay in control
- We may want tasks to run in parallel, but maybe also serially to their parent process

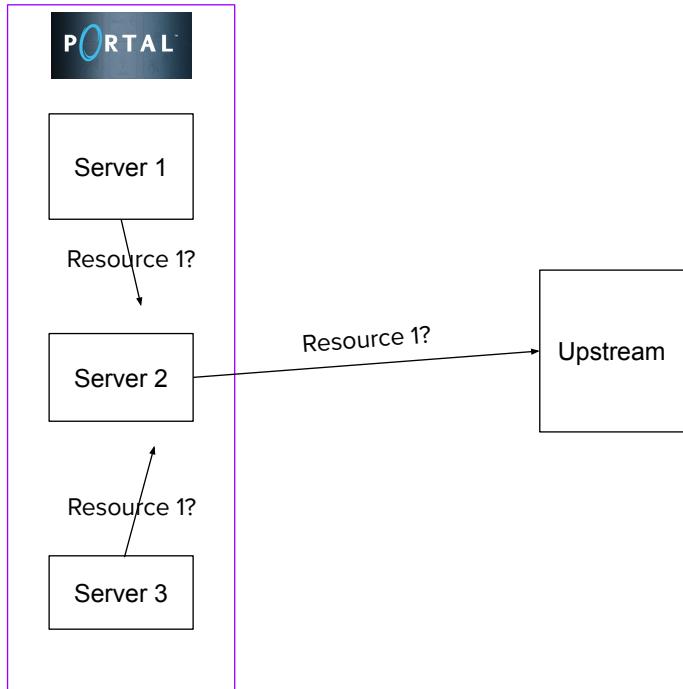
**A majority of the
performance problems
I've had with Elixir was it
going too parallel**

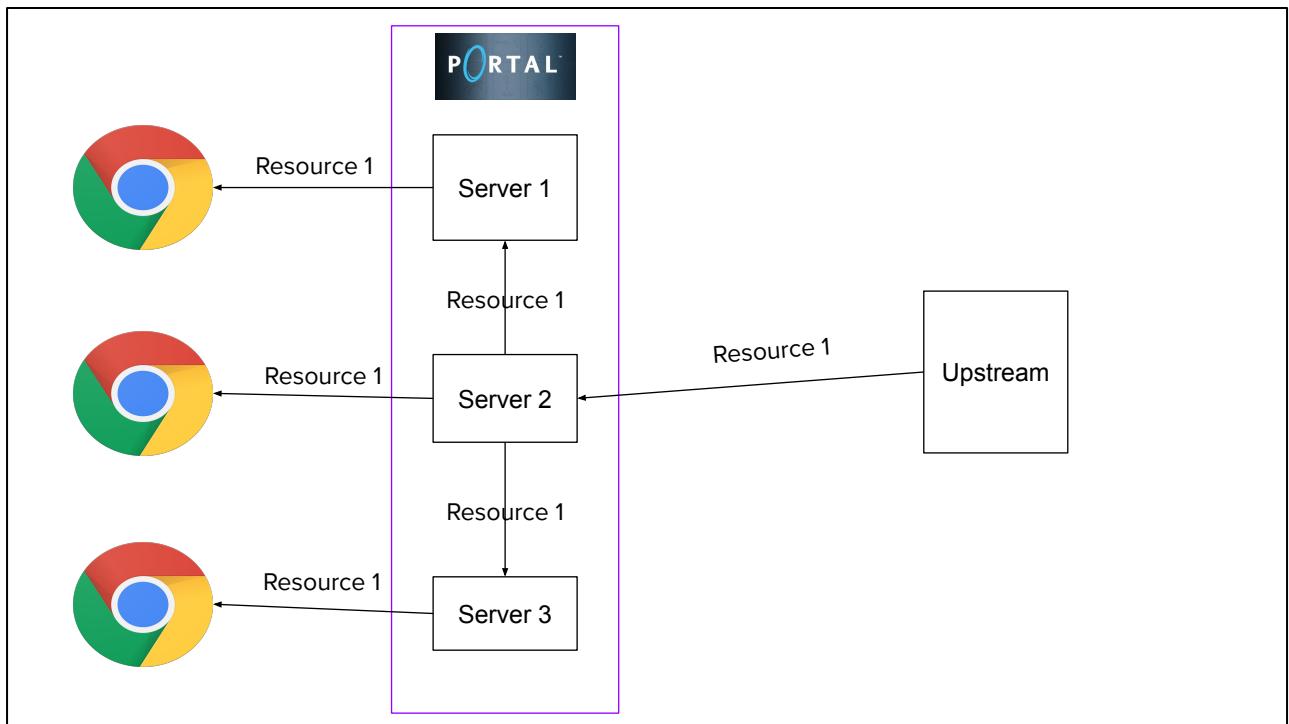
Our experience in learning to put Elixir into production at SalesLoft is that the BEAM is capable of massive concurrency, and that can put strain on other resources.

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

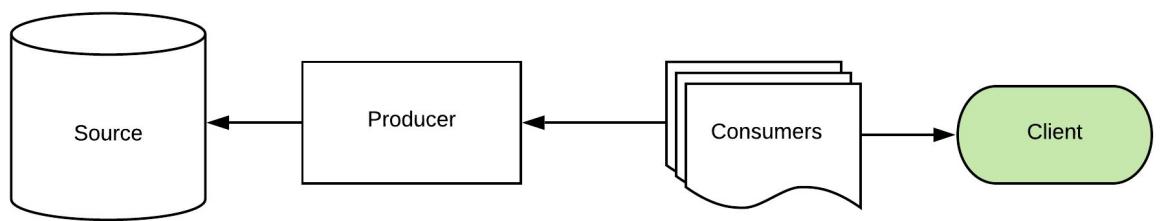






What is a data pipeline?

- 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



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

- Building blocks of your data pipeline can compose into larger pieces of functionality
- 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

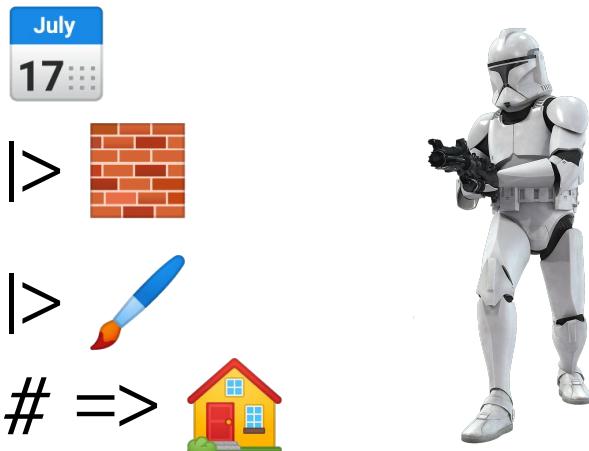
```
[1, 2, 3, 4]
|> Enum.map(&double/1)
|> Enum.map(&add1/1)
|> Enum.map(&exclaim/1)
```

```
[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

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

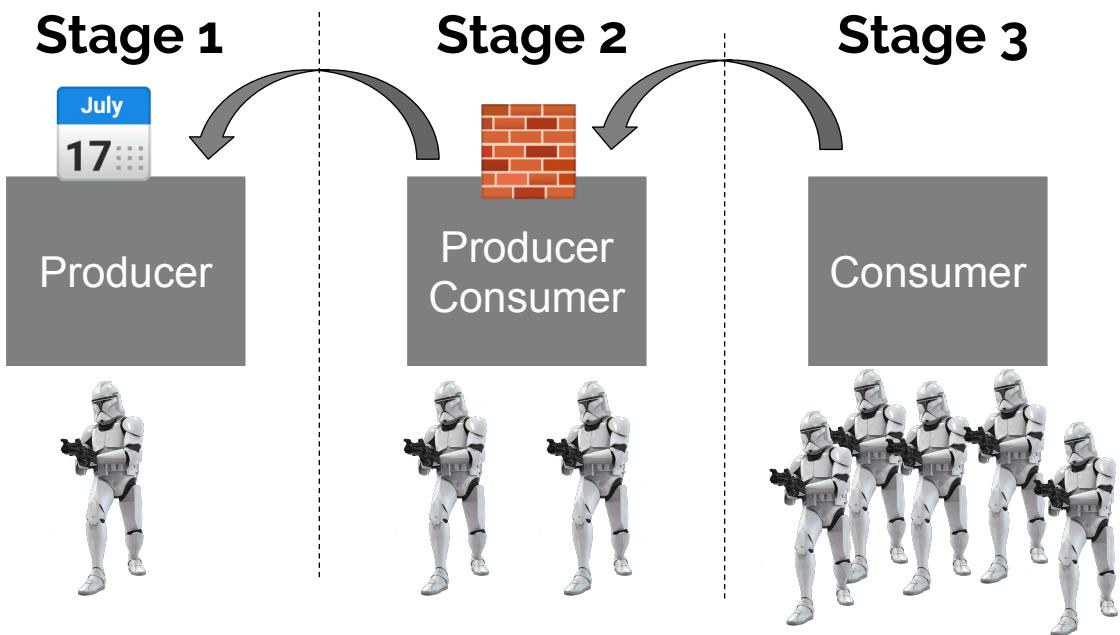
What if we can only be bricking 4 houses at a time? What happens if we can only have 2 painters at a time?

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....

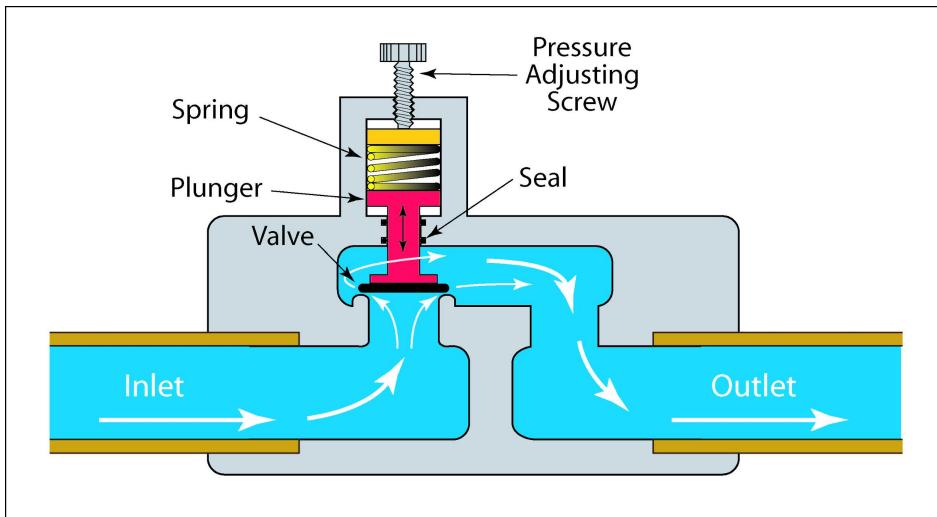
- GenStage is a flexible elixir tool to compose functions while restricting the concurrency of different stages of your data pipeline
- GenStage is **NOT** the right tool for every situation, but we find it pretty handy around SalesLoft
- Checkout the Plataformatec tool called Broadway, which is built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

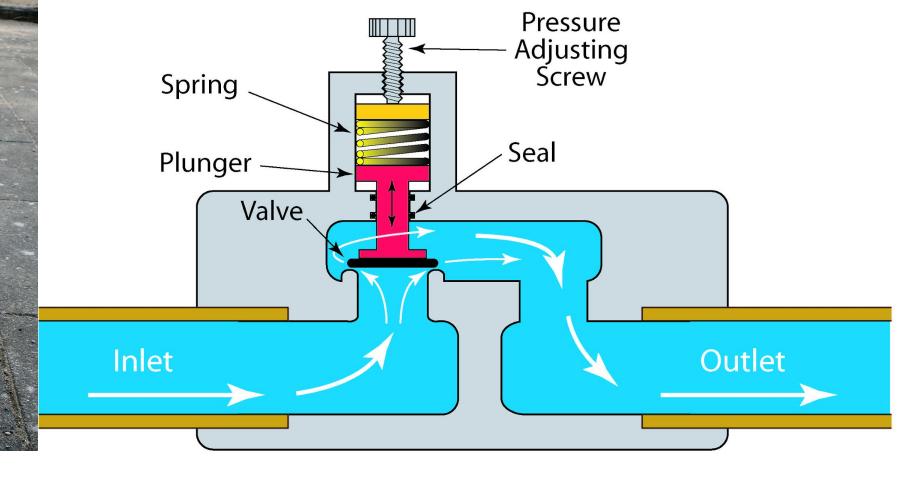


- Add new stages easily over time
- Control the concurrency of any given stage (1, 2, 5)
- Later stages **ASK** earlier stages for work, this provides back pressure to the system
- Producers can get a bunch of appointments at a time, and can be getting appointments while workers are working

Queues Vs. Back Pressure

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





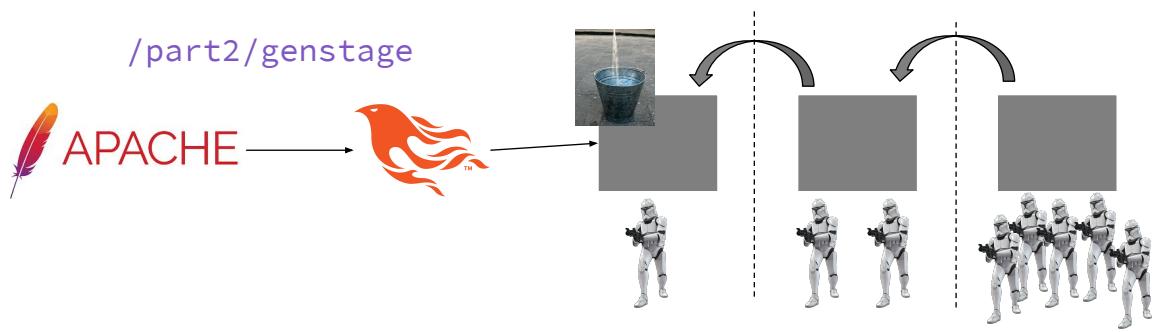
Measurement is Crucial

- 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 <https://bit.ly/real-time-training>
- Use branch section1-part2-start
- Pair up and ask questions as needed!

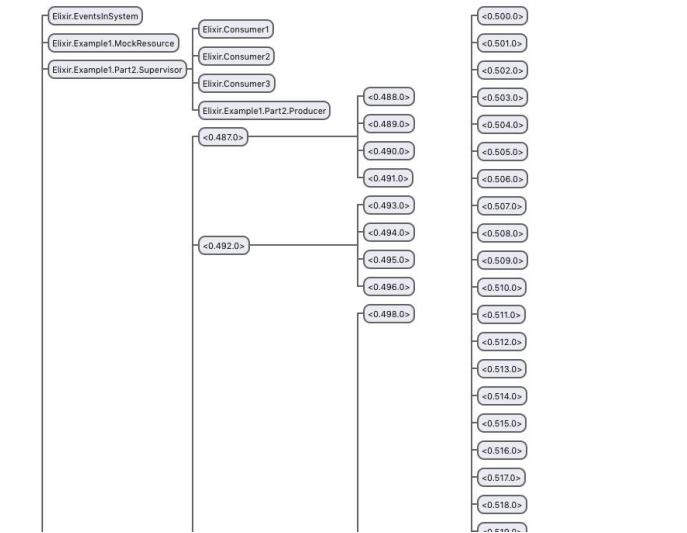
Share with Us

- 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)

Observer



```
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 — Phoenix Channels

-

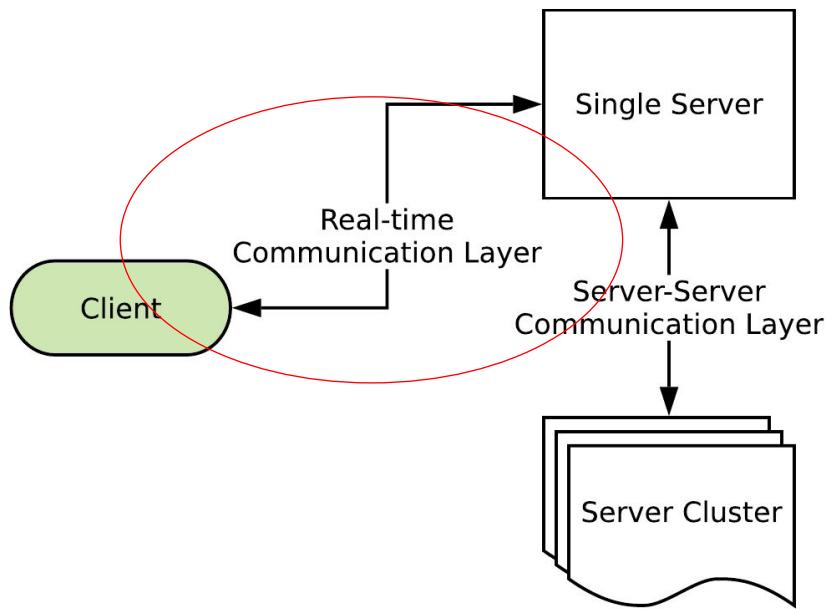
Section 2 Plan

- 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.



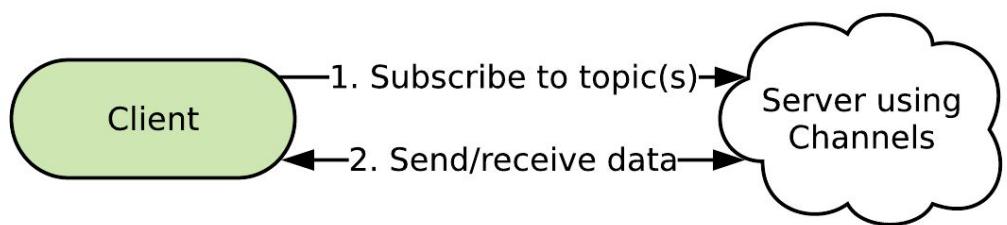
- WebSocket is a real-time communication mechanism
- Sits between the client and a server, providing “last mile” of data to the customer

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

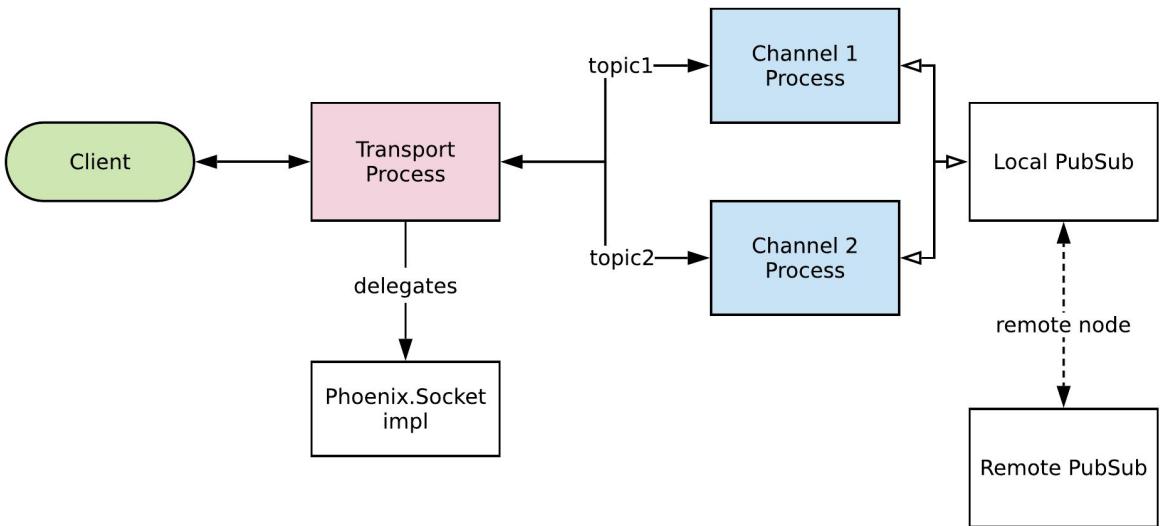


Why not use WebSockets directly?

- 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

```
1  defmodule MyAppWeb.SomeSocket do
2    use Phoenix.Socket
3
4    channel "*", HelloSocketsWeb.SomeChannel
5
6    def connect(_params, socket, _connect_info) do
7      {:ok, socket}
8    end
9
10   def id(_socket), do: nil
11 end
12
13 defmodule MyAppWeb.Endpoint do
14   socket "/some_socket", HelloSocketsWeb.SomeSocket,
15     websocket: true,
16     longpoll: false
17 end
18
```

- 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

```
1  defmodule MyAppWeb.SomeChannel do
2    use Phoenix.Channel
3
4    def join("ping", _payload, socket) do
5      if valid?(socket) do # optional
6        {:ok, socket}
7      else
8        {:error, %{reason: "an error"}}
9      end
10     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

```
12  def handle_in("ping", %{"ack_phrase" => ack_phrase}, socket) do
13    {:reply, {:ok, %{ping: ack_phrase}}, socket}
14  end
15
16  def handle_in("ping", _payload, socket) do
17    {:reply, {:ok, %{ping: "pong"}}, socket}
18  end
19
20  def handle_in("pang", _payload, socket) do
21    {:stop, :shutdown, {:ok, %{msg: "shutting down"}}, socket}
22  end
```

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

Send messages to clients

- transition

```
24  # Anywhere in your app
25  MyAppWeb.Endpoint.broadcast("topic", "message", %{data: "test"})
26
27  # In Channel
28  intercept ["message"]
29
30  def handle_out("message", payload, socket) do
31      push(socket, "message", payload)
32      {:noreply, socket}
33  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

```
10 // Join a Channel, handling any errors
11 channel.join()
12   .receive("ok", resp => { console.log("Joined ping", resp) })
13   .receive("error", resp => { console.log("Unable to join ping", resp) })
14
15 // Send a data, handling errors or timeouts
16 channel.push("ping")
17   .receive("ok", resp => console.log("receive", resp.ping))
18   .receive("error", resp => console.error("an error", resp))
19   .receive("timeout", resp => console.error("a timeout", resp))
```

- Join a Channel, send data to the server

```
21 // Receive messages from the server and process them
22 channel.on("send_ping", (payload) => {
23   console.log("ping requested", payload)
24   channel.push("ping")
25     .receive("ok", resp => console.log("ping:", resp.ping))
26 })
```

- Receive new events from a server

Channels Example



Getting Started

- Download code from
<http://bit.ly/real-time-training>
- Follow section2-part1-start only
- Complete Channels tasks listed
- Pair up, I'll walk around for help

•

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

```
1  defmodule HelloSocketsWeb.AuthSocket do
2    use Phoenix.Socket
3    require Logger
4
5    def connect(%{"token" => token}, socket) do
6      case verify(socket, token) do
7        {:ok, user_id} =>
8          socket = assign(socket, :user_id, user_id)
9          {:ok, socket}
10
11        {:error, err} =>
12          Logger.error("#{__MODULE__} connect error #{inspect(err)}")
13          {:error, err}
14      end
15    end
16
17    def connect(_, _socket) do
18      Logger.error("#{__MODULE__} connect error missing params")
19      {:error, nil}
20    end
21
22    @one_day 86400
23
24    defp verify(socket, token),
25      do:
26      Phoenix.Token.verify(
27        socket,
28        "salt identifier",
29        token,
30        max_age: @one_day
31      )
32    end
```

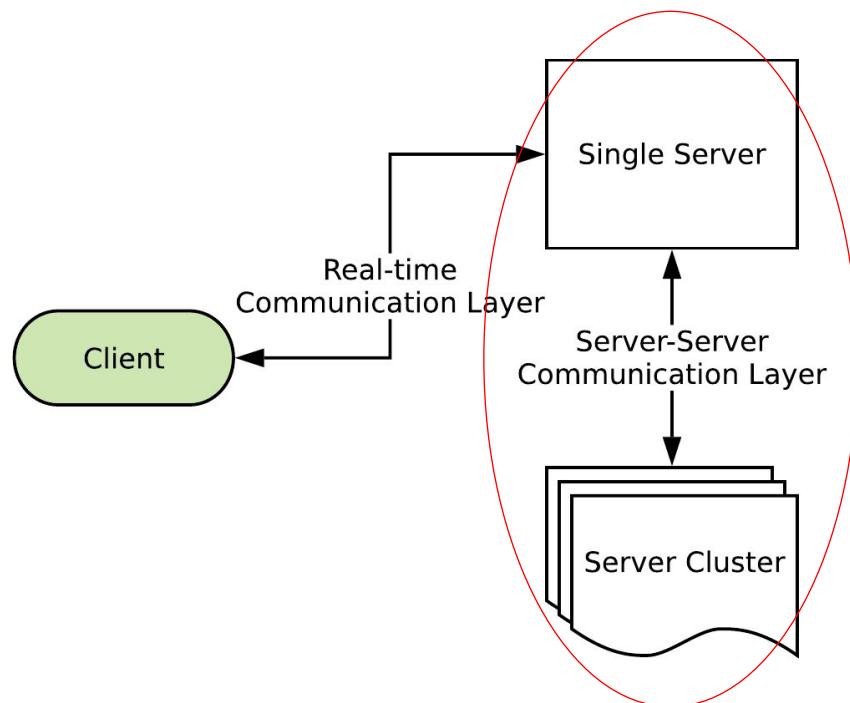
```
1 defmodule HelloSocketsWeb.AuthChannel do
2   use Phoenix.Channel
3
4   require Logger
5
6   def join(
7     "user:" <> req_user_id,
8     _payload,
9     socket = %{assigns: %{user_id: user_id}}
10    ) do
11      if req_user_id == to_string(user_id) do
12        {:ok, socket}
13      else
14        Logger.error("#{__MODULE__} failed #{req_user_id} != #{user_id}")
15        {:error, %{reason: "unauthorized"}}
16      end
17    end
18  end
19
```

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



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

```
1  defmodule MyAppWeb.RecurringChannel do
2    use Phoenix.Channel
3
4    @send_after 5_000
5
6    def join(_topic, _payload, socket) do
7      schedule_send_token()
8      {:ok, socket}
9    end
10
11   defp schedule_send_token do
12     Process.send_after(self(), :send_token, @send_after)
13   end
14
15   def handle_info(:send_token, socket) do
16     schedule_send_token()
17     push(socket, "new_token", %{token: new_token(socket)})
18     {:noreply, socket}
19   end
20
21   defp new_token(socket = %{assigns: %{user_id: user_id}}) do
22     Phoenix.Token.sign(socket, "salt identifier", user_id)
23   end
24 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
<http://bit.ly/real-time-training>
- Follow section2-part2-start only
- Complete Channels tasks listed
- Pair up, I'll walk around again

•

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

- 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
<http://bit.ly/real-time-training>
- 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
```

```
1  # Channel
2  def handle_in("work", payload, socket) do
3    |> Worker.perform(payload, socket_ref(socket))
4    |> {:noreply, socket}
5  end
6
7  # Worker
8  def handle_info({:work_complete, result, ref}, socket) do
9    |> reply(ref, {:ok, result})
10   |> {:noreply, socket}
11 end
12
```

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]]},
17 ]
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
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` in 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

Home(H) Network(N) System(S) Ets(E) App(A) Doc(D) Plugin(P) recon:proc_count(reductions, 18) Interval:1500ms							0 Days 0:0:42
Erlang/OTP 22 [erts-10.4.3] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [hipe]							
System	Count/Limit	System Switch	Status	Memory Info	Size		
Proc Count	219/262144	Version	22.0.4	Allocated Mem	77.8008 MB	100.0%	
Port Count	15/65536	Multi Scheduling	enabled	Use Mem	46.0313 MB	59.17%	
Atom Count	25975/1048576	Logical Processors	12	Unuse Mem	31.7694 MB	40.83%	
Mem Type	Size	Mem Type	Size	IO/GC	Interval: 1500ms		
Total	48.6569 MB	100.0%	Binary	2.8372 MB	05.83%	IO Output	5.5889 KB
Process	12,9936 MB	26.70%	Code	13,9754 MB	28.72%	IO Input	6 B
Atom	722,5244 KB	01.45%	Reductions	133493		Gc Count	18
Ets	1.6728 MB	03.44%	RunQueue	2		Gc Words Reclaimed	328089
1	00.27%	7	00.02%	13	00.00%	19	00.00%
2	00.02%	8	00.02%	14	00.00%	20	00.00%
3	00.03%	9	00.02%	15	00.00%	21	00.00%
4	00.02%	10	00.25%	16	00.00%	22	00.00%
5	00.06%	11	00.02%	17	00.00%	23	00.00%
6	00.03%	12	00.02%	18	00.00%	24	00.00%
No	Pid	Reductions	Name or Initial Call	Memory	MsgQueue	Current Function	
1	<0.9.0>	2725695	erl_prim_loader	66.3242 KB	0	erl_prim_loader:loop/3	
2	<0.64.0>	790667	group:server/3	2.7829 MB	0	group:more_data/6	
3	<0.49.0>	607923	code_server	2.4285 MB	0	code_server:loop/1	
4	<0.2.0>	390240	erts_literal_area_collector:start/0	2.6250 KB	0	erts_literal_area_collector:msg_lo	
5	<0.1.0>	379491	erts_code_purger	20.0078 KB	0	erts_code_purger:wait_for_request	
6	<0.43.0>	336617	application_controller	1.1482 MB	0	gen_server:loop/7	
7	<0.57.0>	332610	file_server_2	673.4492 KB	0	gen_server:loop/7	
8	<0.3.0>	50951	erts_dirty_process_signal_handler:star	2.7422 KB	0	erts_dirty_process_signal_handler:	
9	<0.62.0>	21888	user_drv	13.6758 KB	0	user_drv:server_loop/6	
10	<0.116.0>	18488	elixir.Mix.ProjectStack	416.5117 KB	0	gen_server:loop/7	
11	<0.116.0>	18488	Elixir.DBConnection.Connection:init/1	3.6016 KB	0	erlang:hibernate/3	
12	<0.658.0>	10993	Elixir.DBConnection.Connection:init/1	3.6016 KB	0	erlang:hibernate/3	
13	<0.653.0>	10993	Elixir.DBConnection.Connection:init/1	3.6016 KB	0	erlang:hibernate/3	
14	<0.659.0>	10977	Elixir.DBConnection.Connection:init/1	3.6016 KB	0	erlang:hibernate/3	
15	<0.656.0>	10977	Elixir.DBConnection.Connection:init/1	3.6016 KB	0	erlang:hibernate/3	
16	<0.654.0>	10755	Elixir.DBConnection.Connection:init/1	3.6016 KB	0	erlang:hibernate/3	
17	<0.655.0>	10746	Elixir.DBConnection.Connection:init/1	3.6016 KB	0	erlang:hibernate/3	
18	<0.650.0>	10745	Elixir.DBConnection.Connection:init/1	3.6016 KB	0	erlang:hibernate/3	

- 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
<http://bit.ly/real-time-training>
- 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

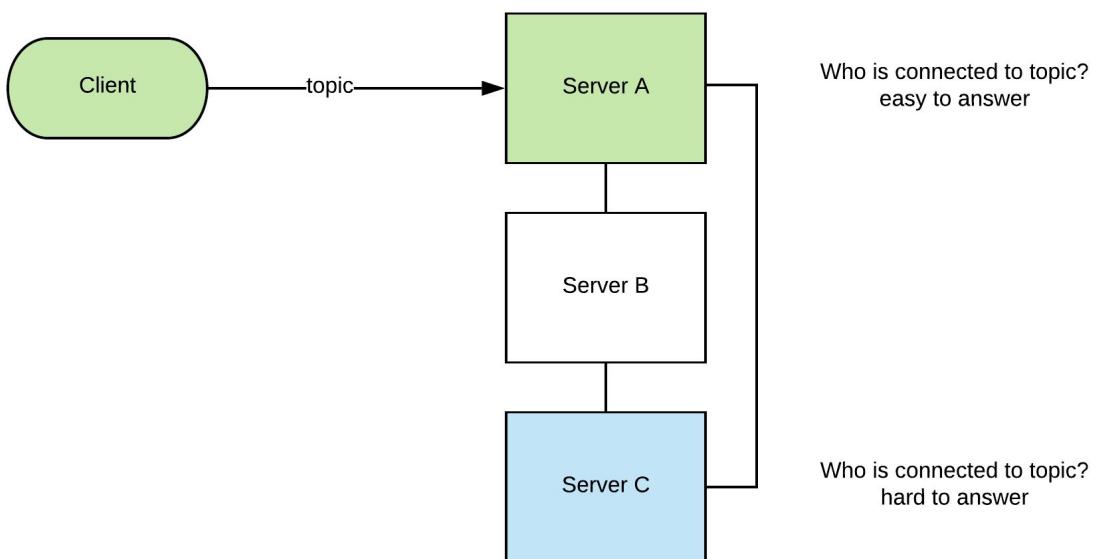


Section Plan

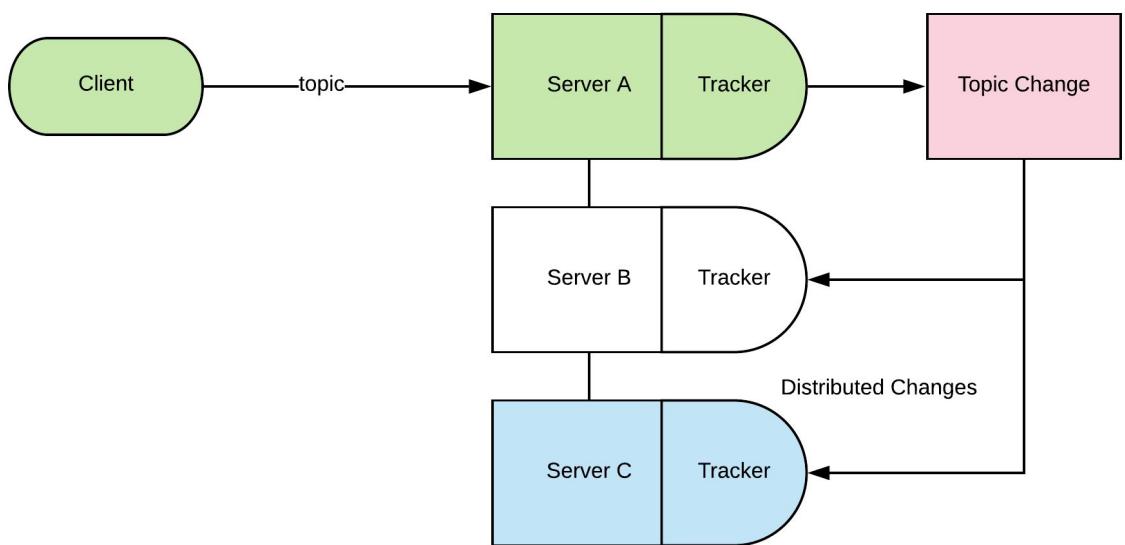
- Understand Phoenix Tracker and Presence
- See how Tracker can fit into a data pipeline
- Challenges of distributed code

What is Phoenix Tracker?

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



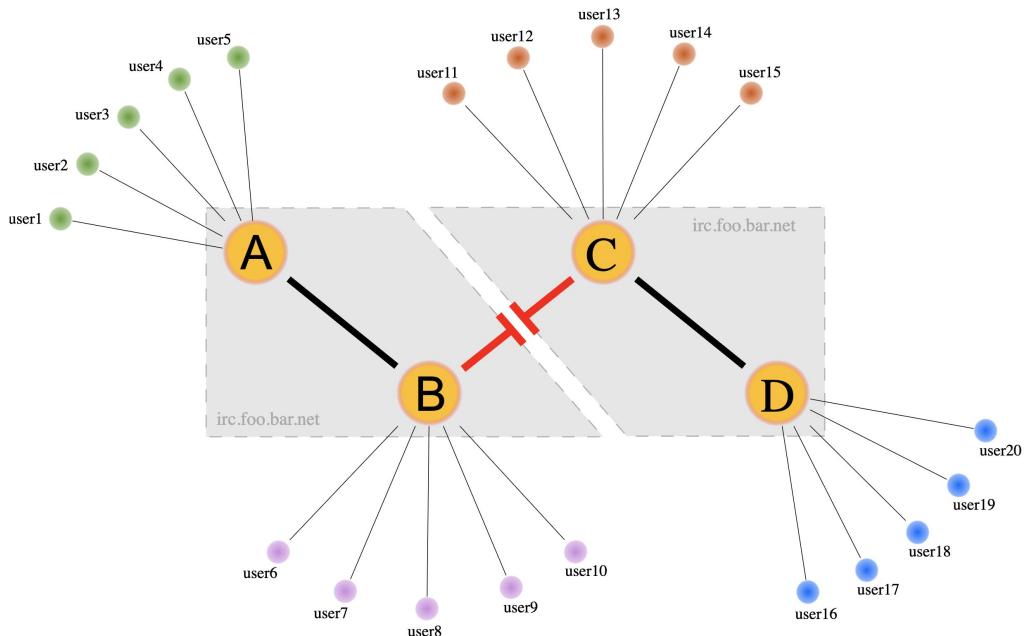
- It's easy to know if the client is connected when server A asks (just look in memory)
- It's hard to know if the client is connected when server C asks (has to jump the network)



- 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

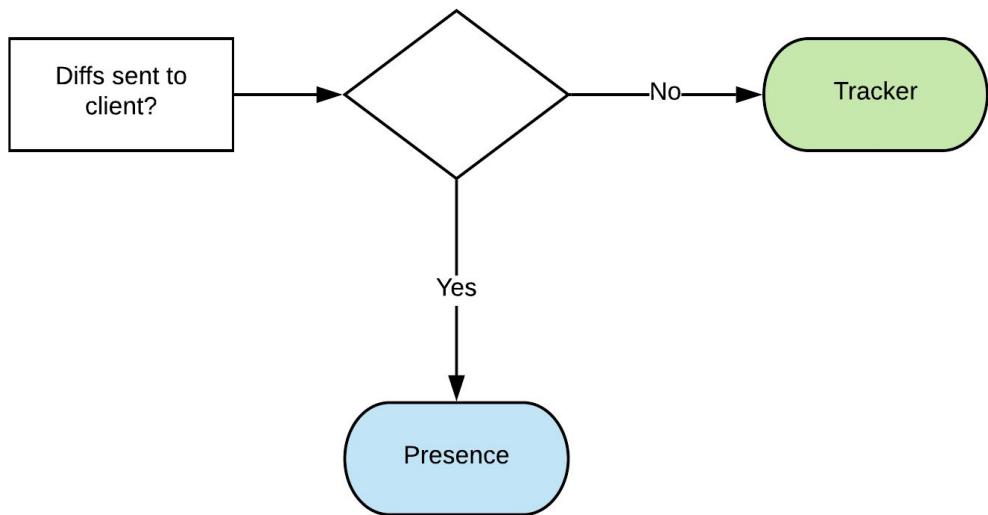
- Steve - Pet peeve to jump to Presence when you probably want Tracker
- Presence is Tracker + an additional feature. That means it is more expensive and you may not even want the additional feature

```
~~~  
...  
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?

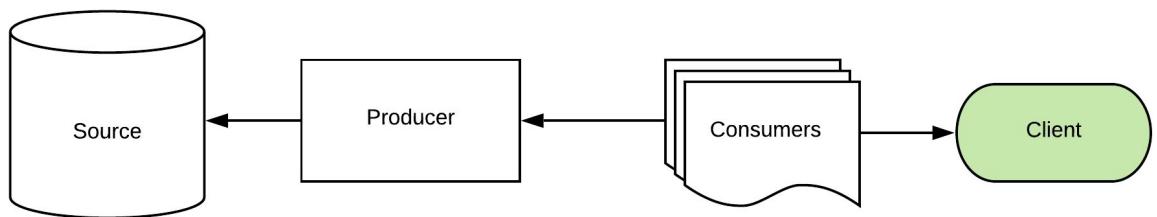
- Next slide



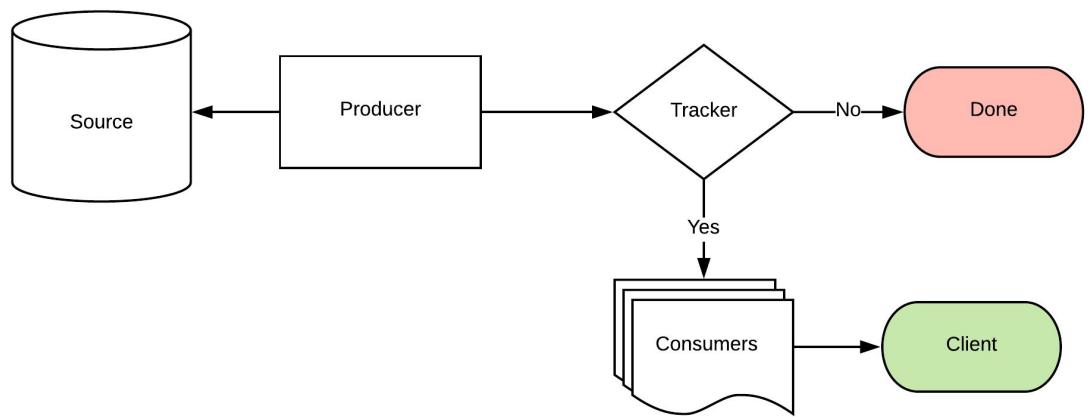
- Really easy question to ask yourself: Do you want all presence diffs being sent to the clients connected to that topic?
- If yes -> Presence. If no -> Tracker

Tracker in a data pipeline





- Data pipeline without Tracker
- Items are produced from a database, run through multiple consumers, and finally spit out to the destination (a client)
- What if no client is there? -> Then we waste 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
<http://bit.ly/real-time-training>
- 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

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

```
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

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

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

```
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 pre-1.0

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
<http://bit.ly/real-time-training>
- Checkout **section4-part2-start**
- Complete LiveView tasks listed
- Pair up, raise hands with any questions

•

Share with Us



Q&A Session



Thanks!