Incrementally migrating large apps to Phoenix

Ruby Elixir Conf Taiwan 2018

Jake Morrison @cogini

Agenda

- Why migrate to Phoenix
- How to migrate

Why?

- Rails is a great way to quickly create applications
- It hits limits when systems get larger or we need to keep persistent connections

- Do cool things
- Improve performance, reduce costs
- Improve reliability
- Reduce system complexity
- Improve maintenance and ops

My story

- Background in telecom, VoIP, and supply chain
- In 2005, started product development agency using Rails
- Bus route management system project, IoT 1.0
  - Tired of doing network communication in C++
  - Wanted real time web interfaces for maps and alerts
  - Rails process model had trouble
  - Found Erlang

Erlang

- Runtime and programming language created by Ericsson for telecom systems
- C++ systems were out of control
- Highly reliable: nine nines of availability
- Highly concurrent: tens of thousands of simultaneous calls
- Distributed: reliability requires more than one machine

Erlang

- Practical functional programming
- Isolates requests from each other
- Patterns for state management
- Patterns for fault handling
- Best-of-class system-management tools

Erlang

- Great for network communications systems
- Web development stack was not mature
- Ended up making hybrid apps: front end in Rails, real-time back end in Erlang

Examples

- Stateful web
  - Chat
  - Real-time auctions
  - Push notifications
- Ad-tech
- Bots
- Embedded systems
- Health care and financial services

Elixir / Phoenix

- In 2014, saw Chris McCord's blog post comparing Rails and Phoenix
  - https://littlelines.com/blog/2014/07/08/elixir-vs-ruby-showdown-phoenix-vs-rails
- Best of both worlds: the ease of use of Rails and the power of Erlang
  - Similar syntax, similar structure (MVC)
  - Less magic
  - Better performance
  - Ability to make next generation applications
- Switched our new projects to Elixir

Performance

Phoenix is typically 10x faster than Rails

- Compiled vs interpreted
  - Compiled views, compiled routes
  - Hot code loading makes development fast
- Uses database more efficiently
  - Explicit joins instead of automatic loading
  - Uses compile time schema, not runtime meta
- It's just math
  - 100 ms = 10 requests per second / CPU core
  - 10 ms = 100 requests per second / CPU core

Concurrency

Better concurrency = better use of resources

- A Rails process handles one request at a time
- Each needs its own RAM, not shared
- Proxying via Rails = idling resources
  - Database
  - HTTP APIs
  - Elasticsearch

Complexity

Poor performance = system complexity

- Background job handlers
- Caching everywhere
- More components
- Less reliability
- Poor load management

Elixir/Phoenix

- Single virtual machine
- Processes are light weight, effectively unlimited
- Shared resources, e.g. in-memory key/value store for caching
- Similar programming model to Rails
- Frameworks for stateful apps

Putting the band back together

- Public web
- CMS
- Back end admin
- Mobile APIs
- Real-time communication
- 3rd-party integrations
- Background jobs

Splitting up the monolith

- Microservices?
- Docker?
- Phoenix domains
- Elixir applications in umbrella apps

Incremental migration

Scenarios

- Implement real time chat back end
  - Channels, pub-sub, presence
- Split off api.example.com
- Implement GraphQL
- Protect back end
- Rate limit traffic: api, scraper, DDOS
- Proxy and coordinate communication

Process

Similar approach for most applications

Monitor, analyze

- Monitor performance and reliability
  - Hosted service like Datadog or Prometheus
  - Elasticsearch / Logstash / Kibana
- Look at the traffic
- Count things: requests, response time, errors
  - Figure out where the problems are
  - 99% time is most interesting, not average
  - Identify and classify errors

Prioritize

- User experience
- Cost
- Errors
- Maintenance or ops pain

Route traffic

- Common front end directs traffic based on URL
  - Nginx, HAProxy, AWS ALB, Varnish
  - Elixir: https://github.com/poteto/terraform
- Manage load
- Improve security
- Collect metrics

Integrate session

- Share user session / login
- Share database, memcached, JWT token

Integrate UI

- Implement common UI template
- Share navigation

Migrate, test, monitor

- Implement HTTP routes
- Test in parallel
  - Ensure new code gets same response as old
- Monitor in production
- Repeat

GraphQL

- A **great** way to build mobile APIs, replacing REST
  - Fundamentally better performance
  - Easier development and maintenance
- An ok way to build web interfaces
  - With Phoenix, CRUD is easy and fast
  - Add channels for interactivity
- Absinthe GraphQL server integrates with Phoenix
- GraphQL as proxy

Improve the architecture

- A good architecture avoids accidental complexity
- Model the natural concurrency of your system
- Splitting everything into tiny pieces doesn't make life better

Functional programming

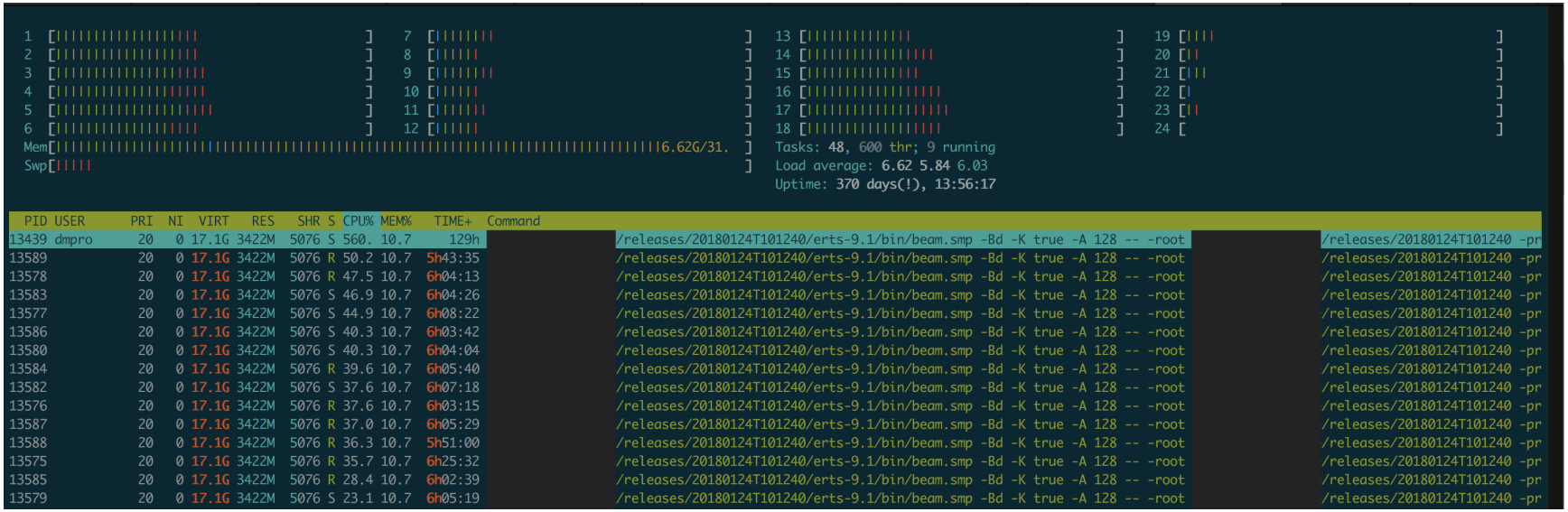
- Receive request, transform, send response
- Avoid side effects
- Avoid shared state
  - The database is usually the ultimate bottleneck
  - Cache data with smart invalidation

Improve reliability

- Use standard patterns
  - Microservice HTTP APIs are just RPC done badly
  - What do you do if it fails?
- Supervise and retry on failures
  - Restart at the beginning
  - Persist data on organization boundaries
  - Event Sourcing
- Reject traffic at the edge

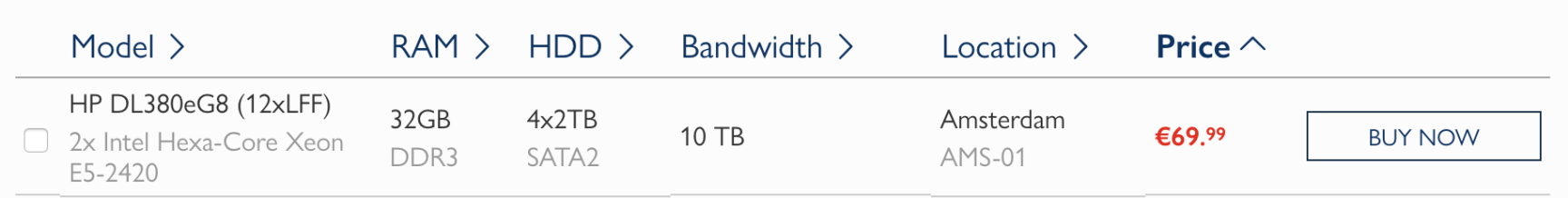
Hosting architectures

What is a good hosting architecture if your platform is great at concurrency?



Cloud vs bare metal

- Cloud providers want you to use the cloud, duh
- Dedicated hardware can perform very well with low operational complexity
  - $100/mo for 24 hyperthreads, 32 GB RAM, 10 TB transfer
  - Two front end servers + two database servers for availability = $400/month



Model	RAM	HDD	Bandwidth	Location	Price	
HP DL380eG8 (12xLFF) 2x Intel Hexa-Core Xeon E5-2420	32GB DDR3	4x2TB SATA2	10 TB	Amsterdam AMS-01	€69.99	BUY NOW

Docker

- Standardized deployment mechanism
- Low concurrency
- Operational complexity
- Elixir umbrella applications

Questions / Comments?