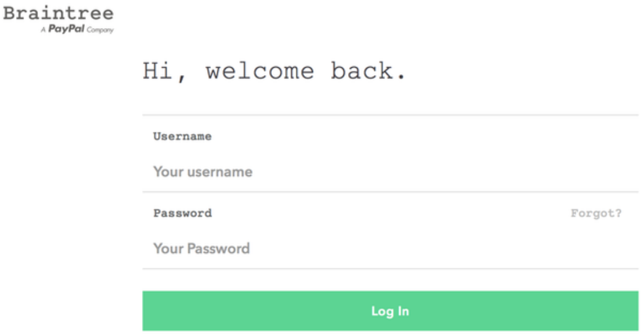
# Decomposing a Highly Available Rails App

Adam Forsyth

github.com/agfor

- I'm Adam Forsyth
- Software Engineer & Community Lead @ Braintree Payments
- We make it easy for you to accept credit cards, PayPal, other payment methods online and in mobile apps.
- This talk is Decomposing a Highly Available Rails App
- Slides + Speaker notes will be posted on my github



Braintree
A PayPal Company

Hi, welcome back.

Username

Your username

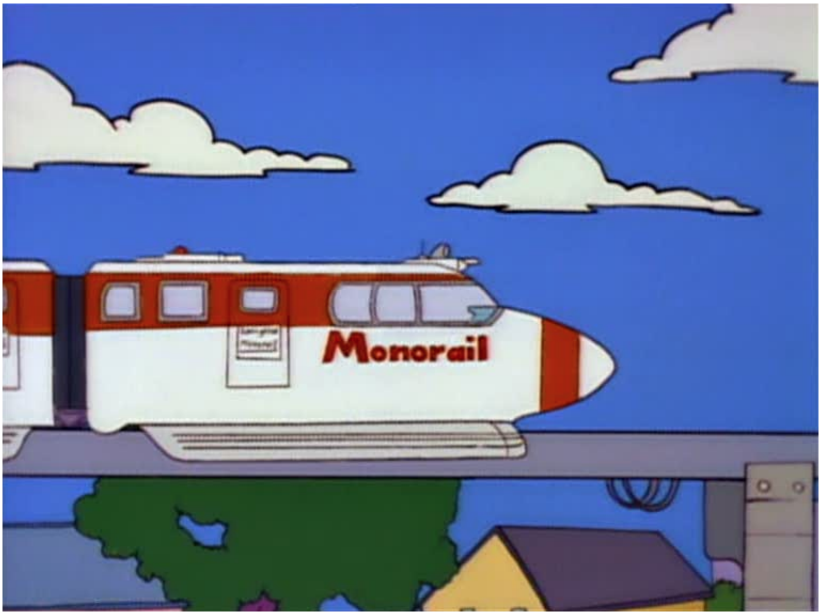
Password

Forgot?

Your Password

Log In

- My talk is about what we did to extract a service from the Braintree Payment Gateway
- I've got some general tips for building modern rails API apps
- Then in a little more depth, I'll talk about three somewhat unique things we did that may be useful to you
- Going to move through this material pretty quick
- I'll let you decide what you want to hear more about by leaving time at the end for questions



- First, some context
- The Braintree gateway is historically a monorail
- Codebase dates to 2008 / Rails 2.1
- On the order of 100,000 LOC

- By 2013 that monolithic architecture was slowing us down
- Limited language / framework choice to Ruby / Rails
- Long ramp up time for new developers
- Features not well separated

$2H_2O_2$
hydrogen peroxide
$2H_2O$
water
+
$O_2$
oxygen

- Started down the road of decomposition
- Started building new features in separate services
- Refactored the gateway to be more modular
- Started extracting existing features into their own services

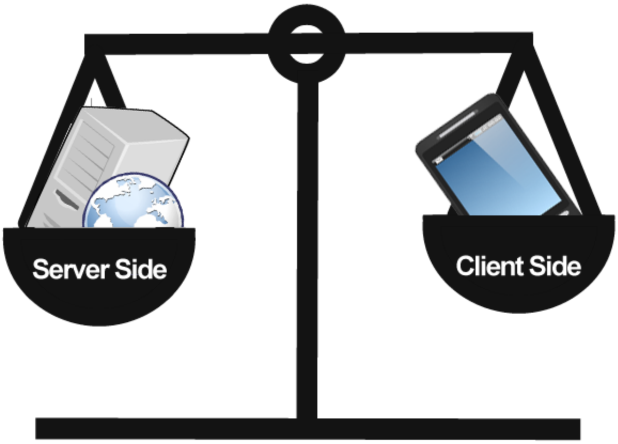
- However, we ran into some problems
- External facing features still had to be build into same app
- Because they needed to share a particular subset of code
- That code is Identity, Authentication, Authorization, Configuration
- Basically, who you are and what you can do

- So in 2014 we decided to extract the code that was forcing us to build new features into the monolith
- Into a new service
- Confusingly, called "Authy", like the two factor auth app
- Enable more decomposition
- High risk -- all traffic depends on that code
- Because All traffic authenticated
- Other people rely on us for their uptime
- We cannot take downtime or risk breaking API authentication

- What was our High-level strategy?
- I can tell you right now, not "Move fast and break things"
- More like "Move Deliberately and don't break things"
- Wanted maximum confidence in code correctness
- Needed a gradual rollout to minimize impact of problems

- Also focused on mondernization
- Could give a whole talk just on this
- Rails 3.2 -> 4.1 (Couldn't use 4.2 at the time because of a bug in rails)
- RSpec 2 -> RSpec 3 (should syntax -> expect syntax)
- Custom serialization to ActiveModel serializers
- Custom proxy / request queueing layer to nginx / unicorn
- AASM -> StateMachine -- Transition failure behavior in AASM wasn't what we wanted, caused problems
- At the time we started this project, we weren't using Database connection pooling for everything in old app
- New app PGBouncer from the start
- Wrote fake, in memory version of service to speed up & simplify testing
- Where 100% fidelity wasn't important

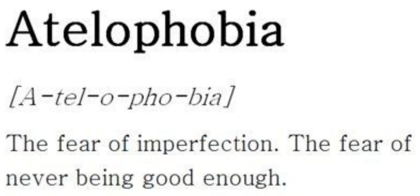
- Of course had performance concerns about going from an in-app call to network call
- No premature optimization here, it was necessary
- Actually going to be going through another round of optimization soon
- Could give a whole talk just on this too
- First thought is caching.
- Have some high-volume idempotent endpoints we could safely cache client side, eliminating request entirely
- Some nginx-level server side caching as well



- Next thought, server-side performance
- oj for faster JSON serialization
- rails-api for good defaults & faster requests
- Most requests in single-digit milliseconds
- Bullet to make sure our ActiveRecord queries were sane
- Ended up writing SQL for a couple high-volume, uncacheable endpoints
- Where we needed to collapse several queries into one to save database roundtrips

- Third thought, connection overhead
- So we tried persistent connections
- Had to choose between slow server restart time waiting for connections to close
- Or dropped requests when they're closed unexpectedly
- Ended up using persistent connections for idempotent methods only
- Where we can use automatic retrying if connection closed
- Because we don't have to worry about whether it succeeded the first time
- Post body compression, since we saw bad performance with large requests to the server

- Back to code correctness
- Normal ways of developing confidence in our code
- TDD
- Pairing or Pull requests, no solo commits
- Emphasis on shared context -- everyone on team reads every commit
- Create a seam within the application, then extract functionality
- So contract explicit when building the new service
- Once rollout started, everyone on call -- shared ownership
- Rolled out to production before application complete to work out operational problems early



# Atelophobia

*[A-tel-o-pho-bia]*

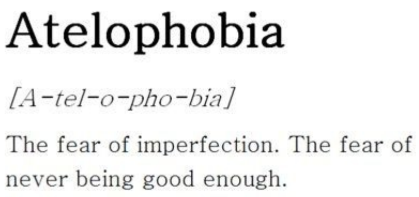
The fear of imperfection. The fear of never being good enough.

- Decided standard stuff not good enough
- Still didn't have enough confidence in correctness of new service
- Because we wanted to rewrite 6 year old Rails 2.1 code to make it more maintainable
- But test coverage on oldest code not great
- So our confidence wasn't high new code would be correct even with passing tests

- First thing we did to increase confidence:
- Wrote a proxy layer called Quackery
- Operates at the seam we introduced within the application, not as a separate service
- Configured with a tree representing hierarchy of endpoints
- Knows which endpoints are idempotent
- Allows you to call both old and new code for idempotent endpoints

robin williams
reality... what a concept
NBL5-7162
REALITY...WHAT A CONCEPT
ROBIN WILLIAMS
PRODUCED BY BROOKS ARTHUR & NEIL BOGART
Consultant to the Producers: Bennett Tramer
℗ 1979 Casablanca Record & FilmWorks, Inc.
PLEASE NOTE THAT CERTAIN WORDS IN THIS ALBUM MIGHT BE CONSIDERED OBJECTIONABLE BY PARENTS OF CHILDREN UNDER 16
STEREO CASSETTE
REALITY...WHAT A CONCEPT
ROBIN WILLIAMS
NICKY LENIN • POP GOES THE WEASEL • KINDERGARTEN OF THE STARS •
A TOUCH OF FAIRFAX • REVEREND EARNEST ANGRY
NBL5-7162
℗ 1979 Casablanca Record & FilmWorks, Inc.
SIDE 1

- This is important because we wanted our testing to be as close as possible to reality
- Intelligently calculates differences in results from both code paths, ignoring small timestamp diffs etc.
- Allows us to compare exact results, using real requests
- Ultimate in automated testing *
- In addition, it allowed us to easily switch which code path different routes were using
- Lets us specify which result to trust in case they differ, when using both code paths



Atelophobia

*[A-tel-o-pho-bia]*

The fear of imperfection. The fear of never being good enough.

- Still not good enough
- Lots of non-idempotent endpoints
- Want more detailed testing than handwritten assertions can provide on those endpoints too

- So, we said, "if we can double-read, why not double write?"
- Basically, is there a way to do the same thing, but with non-idempotent methods?

- Depends on a dirty little secret I haven't mentioned
- Since we're still running old code in old application, we're still using the old database
- Since we need strong consistency, and new code needs same state, it is also using the old database
- Yes, that's right, shared database!
- Google "shared database" and first result is "shared database is an anti-pattern"
- Talk more about this later

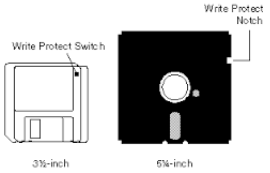
- Back to double writing
- Actually pretty simple, once double-reading in place
- Because all state in a single database
- Add metadata to requests to service, so we could tell it when we were in the process of double-calling non-idempotent methods
- If that's received, start a db transaction as soon as the request starts
- Roll it back after request returns
- So we get exact result of calling non-idempotent method, but without changing database state
- No longer limited to double-sending idempotent methods!
- Can actually compare exact results from old and new code for all endpoints

- Unfortunately, long-running database transactions are a bad idea
- Can kill db performance
- Our DBA wouldn't let me even try it live
- Even just using it in testing, lets us check far more state than we can with assertions
- Still can't check everything, because not all state is serialized back to the client

SHARING
IS CARING

- Let's get back to the shared database
- Not acceptable long term
- Even though it was super useful during the transition
- Once we were 100% using the new service
- Had to split data for the service into own database
- Two basic ways to do that without downtime
- One Duplicating the cluster would require new hardware because cluster size is very large
- Didn't want new DB on separate cluster because this db is small
- Other is Db dump / reload while traffic is paused, either explicitly or implicity because tables are locked
- Combination of db size, db load, and request volume too high to do safely
- The reason we can't do dump / reload while taking traffic because need strong consistency

- Knew from performance, double reading work that most high-volume endpoints were idempotent
- Many in fact were pure -- returned a result, had no side-effects on state
- Turns out the non-pure endpoints fell into one of three categories
- Either not used for transaction processing, so *could* take downtime
- As long as it's announced ahead of time
- Example would be creating a new user or changing settings
- Second category, non-critical, like analytics, could skip for short periods
- Finally, "nearly pure" endpoints
- Meaning primary function could be done in a pure manner, with non-pure functionality non-critical



- Added a config option for readonly mode
- Switched to a readonly database user in that case
- Trap db permission errors
- Made client handle that type of failure gracefully
- Modified "nearly pure" and non-critical endpoints to skip database state changes
- Leaving us with every endpoint either working or failing gracefully in readonly mode

- Allowed us to use db dump / reload method without pausing traffic
- Actual op was fairly complicated, had some minor issues
- Had minutes of degraded functionality
- All transaction processing unaffected
- Because the app was in readonly mode, rather than down

# Questions?

Mobilize your payments.
Globalize your business.

Start accepting PayPal, cards, and whatever's next -- all with a single integration.

www.braintreepayments.com
@braintree

- That's all I have
- Slides with speaker notes will be posted on my github,
- github.com/agfor
- Questions?