Speedshop Tune: Beyond Finance

@nateberkopec

Prepared with care

@nateberkopec

Contents

Hello Beyond Finance!

Thanks for having me take a look at your application (again!). I've identified several areas for investment in performance. Some of the things I've discussed in this report are actually just simple configuration changes you can make today that will save you a lot of money. Others are more long-term projects and skills that you can improve on over the next six months.

This document is organized at the top level by our desired Outcomes, which are my goals for your performance improvements over the next six months. Underneath that are specific Recommendations to achieve those outcomes. Each Recommendation has an associated cost and benefit, rated subjectively on a 5 point scale.Ratings are designed mostly to be relative to each other, i.e. a 5 is always harder or more valuable than a 4, etc. Even cost/benefit roughly means I think it's a toss-up whether or not you should do it, while a cost higher than the benefit rating means I think it's not worth doing in the near term future.

At the end of the document, I will present again the Outcomes and Recommendations without commentary. This is intended to be a quick reference to help you turn this document into action and assist during planning your sprints, etc.

I hope you enjoy this document and find it a useful guide for the next 6 months of performance work on your application.

Nate Berkopec, owner of The Speedshop

Outcome: Reduce Outage Incidents Caused By Salesforce Integration

This was established as the primary objective of our collaboration. In a lot of ways, Glue is practically a Salesforce proxy: over 50% of it's total wall time is spent waiting on Salesforce API calls. The Salesforce ratelimit situation is really idiosyncratic with a lot of things that make it atypical versus other types of ratelimits:

1. **Most rate limits are 24 hours**. This means it can be difficult to recover quickly when you trip the limit on hour 8 of 24.
2. **There are a LOT of different limits**. Tons of different resource types have their own limits.
3. **There are several not-really-documented limits**. You've hit a couple of these already!

This is a very difficult foundation to build a robust and reliable service on top of.

You've already tried and moved off of Heroku Connect, and apparently Salesforce is trying to sell you on an alternative with Mulesoft. I've never used Mulesoft and can't really comment on it, so this report is going to proceed from the assumption that you're not going to use any additional third party integrations.

Overall my recommended strategy is:

1. **Reduce Salesforce load during peak hours**

The most dangerous time is when agents are up and about and doing work. An outage at this time is far worse than an outage outside of peak. We should manage and preserve all possible headroom during peak hours.

2. **Increase Salesforce load outside of peak hours** Any work that doesn't need to be done immediately should be put into queues and carefully managed.
3. **Gracefully degrade** when Salesforce load is exceeding internally-set limits. Far better to exceed our own internal rate limits than Salesforce's, and internal "soft limits" will allow us to degrade quality of service for non-essential services and maintain it for essential ones.

You showed me a slide deck which contained an inventory of 6 outages related to the Salesforce integration over the last few months. One observation: though you asked me to look at the *web* side of the Glue app, 5 out of 6 of these incidents were *caused* by background jobs. This isn't that surprising to me: background jobs are a common source of sudden load overwhelming limited resources. Typically, that's something like "CPUs on the DB" but in your case it's this very idiosyncratic Salesforce integration.

Recommendation: Create a long-query lock

One of the types of outages you encountered was due to a limit on the **number of queries which have been active for 25 second or longer**. This is kind of a difficult thing to lock around, as:

1. Potentially *any* query can suddenly take 25 seconds.
2. We don't know if a query will take 25+ seconds until that event has already passed.

Yet, I think there's a "soft" heuristic we can use that will greatly reduce the likelihood of such events in the future. As I mentioned in the overview of the strategy for this section, we can **gracefully degrade** service when there are many "long" queries in flight.

Consider the following algorithm:

1. For each Salesforce request, start a timer.
2. If that timer reaches 25 seconds, take out a Sidekiq::Limiter concurrent lock. The lock is released when the query finishes.
3. At the same time, add a key to a dictionary (probably in Redis) and increment by one. `long_queries[query_signature] += 1`. Each Salesforce op must be able to have a "query signature" (you'll have to come up with a clever way to define this, in SQL it would be the prepared statement w/no bind params). You might also consider decrementing this dictionary key if the query_signature takes less than 25 seconds. I'm not sure - the heuristic will have to be tuned.

4. For each Salesforce request, if the query_signature is present in the `long_queries` dictionary.

Hopefully, consulting the dictionary doesn't feel like this.

, you must acquire the same concurrent lock as in step.

This basically ensures that queries which have, in the past, caused 25 second+ operations must acquire a concurrency lock (which you would limit to ~80-90% of the real Salesforce limit).

It's not a perfect solution of course. A never-before-seen query (from a new deploy) which always takes >25 seconds and has extremely high volume (like, say, 25+ background job threads all firing it off at once) would evade this. But I think it's a good heuristic to avoid ~90% of the incidents you might encounter, and it adds just a single Redis call to every Salesforce query (and only locks rarely).

Luckily, any query which routinely takes more than 25 seconds should be in a background job anyway (right?!? RIGHT?!?!) so the effect of not obtaining the lock is an easy retry.

Cost: 3, Benefit: 4 I think the difficult thing here will be tuning that heuristic about what query_signatures must obtain a lock before starting. The actual code to implement all this isn't very complicated.

Recommendation: Pass all Salesforce API requests through a single, rate-limited service

In general, when it comes to rate-limiting, the key is to have **just one rate-limiter, as centralized as you can possibly get it**. Having multiple internal rate-limits is a recipe for disaster, as you will run into one of two problems:

1. They're not coordinated with each other, so you still end up exceeding the external service rate limit.
2. They're set extremely low, which means you leave throughput on the table.

A lot of people can get away with #2, but Salesforce is far too central for the application's function for that to work. The mistake in your case is to start allowing many ratelimits to multiply: typically the first mistake is to start locking Sidekiq jobs. Do not do this!

Even though you do have a nice "single point of entry" for all Salesforce requests via your Faraday middleware, this recommendation is about the fact that Salesforce load can also come from two of your other Rails applications. This is still a recipe for either over-or-under throttling.

Here are your options:

1. Each of the applications can use the same rate limiting middleware, and then use the same Redis-powered locks. For example, implement the rate limiter from the previous recommendation, distribute it as a gem, and make every app use the same `salesforce-long-concurrent` lock key in Redis.
2. Rewrite your other apps which interact with Salesforce directly to instead go through Glue.
3. Connect to Salesforce through a proxy, which maintains your internal ratelimits. You could use something like nginx, HAProxy, or a custom proxy (use Go or something, this is not a job for Ruby!).

1 and 3 are probably the most reasonable approaches. It would probably depend on what ratelimits you exactly decide to implement.

Cost: 4, Benefit: 5: This could involve a lot of code (particularly if you build your own proxy), but I cannot stress this importance of "one lock for one (external, Salesforce) ratelimit".

Recommendation: Maintain internal ratelimits for all relevant SF limits

The `limits` endpoint is extremely extensive. What makes it so difficult is that a number of these ratelimits are *daily*, which means that an outage could potentially last up to 24 hours!

As with the long-query ratelimit, I think you can and should maintain internal ratelimits for anything that you find yourself often getting warnings for. You

could even have a kind of "global" ratelimit that kicks in once *any* ratelimit hits 90% of its maximum on the `limits` endpoint. We'll talk in the next recommendation about what behavior is reasonable once that limit kicks in.

It can feel like a lot of global locking, and it is, but these requests each cost about 50ms, so checking 2 or 3 locks per request is really not the most expensive part of the operation, and the consequences of getting this wrong (complete outage, or outage of a particular resource) are really quite dire. I think you're going to learn, as you start to implement this stuff, what kinds of locks are expensive, which are cheap, and which are CPU intensive (on Redis) and which are not. I think the number is somewhere between 2 and 10 per request, but YMMV.

Each query can/should carry with it a piece of metadata that says what, if any, limits from `/limits` that will be impacted by this query. For each of those limits, internally ratelimit or lock appropriately.

Example:

```
{ request_url_path: "/mass_email", payload: ..., lock: "MassEmail" }
```

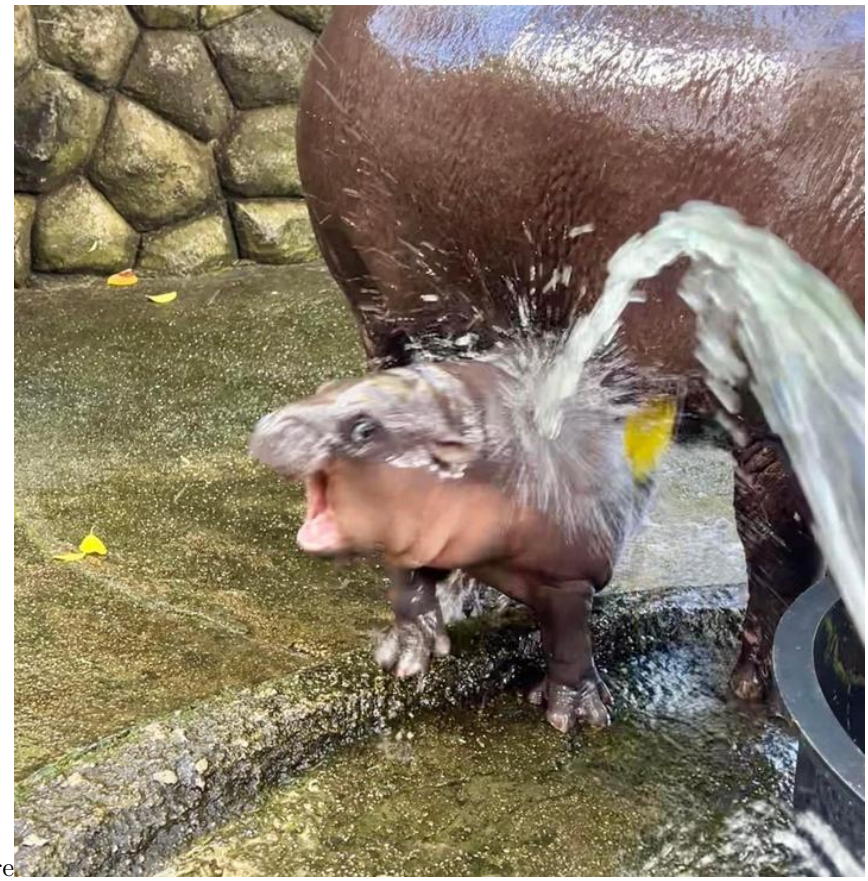
More global limits, such as `DailyApiRequests`, could simply increment a counter until they reach a certain ~% of maximum, at which point they can start to introduce locking or QoS degradation (in the next recommendation).

Cost: 2, Benefit 3: I think once you "get good" at implementing ratelimits, adding one more becomes easier.

Recommendation: Attach QoS expectations to every SF request, let the ratelimiter decide

Currently, you have two levels of service:

1. Unfettered successful access.



2. Immediate failure
   Don't let your poor, poor Glue feel like Moo Deng.

   , enforced by external Salesforce ratelimit.

This does not have to be the case. One can imagine a world where several levels of service exist:

1. Always unfettered, successful access.
2. Try again later.
3. Try again much later, if you are not an important request.
4. Immediate failure, enforced by internal ratelimit.

Each Salesforce request in your system has a varying level of importance. Requests created by web requests are generally very important. A real live human agent is on the other end of that request. However, what if a request is coming from a `within_5_minutes` background job, and that queue's latency is currently 5 seconds? Well, if you needed to throttle your Salesforce calls for a while, you could tell that query to come back in 3 or 4 minutes and try again.

This is another piece of metadata you can attach to each request:

```
{ request_url_path: "/mass_email", payload: ..., lock: "MassEmail", qos: "within_5_minutes"
```

Let's say you're within 10% of exhausting a 24-hour/daily ratelimit (let's make it a Big Deal and say it's the `DailyApiRequests` ratelimit!). Any query with a `qos` of `realtime` goes through with no additional checks. But any query with any other qos can be handled in a variety of different ways:

1. Hit with a Sidekiq ratelimiter for a certain number of queries per second.
2. Immediately rejected, say in the case of a `within_24_hours qos`.

This allows various levels of queueing to ensure system availability.

Cost: 4, Benefit: 5 Probably one of my more complicated suggestions to do right. This could introduce a lot of complex system behavior. I think it depends on "how badly do you need it?". How far do you feel you need to go to prevent these issues?

Recommendation: Create QoS degradation playbooks

At any point when you start implementing an automatic Quality of Service degradation, that should probably start paging somebody. I think the heuristics what causes these degradations will always be imperfect, because several of the ratelimits you're dealing with on the Salesforce side are so imprecise (the "suddenly a lot of concurrency" one comes to mind).

A human will probably have to be in the loop for two reasons:

1. To manage QoS and have the ability to "turn the dial" up or down on QoS degradation.
2. To be able to post-mortem the QoS degradation and set different "default" thresholds for the future.

This process doesn't need to be ad-hoc. There can be a series of levers to pull for the responder, such as:

1. Modify this rate-limit.
2. Turn these queues off, in this order.
3. Here's what to do while turning things back on. If you had to pause a queue long enough that the SLA was violated, here's what to do.

Cost: 1, Benefit: 3 Giving people the tools they need to respond to this kind of "soft incident" can go even further than automated tools.

Recommendation: Do not open incidents automatically, require humans to open incidents with a human-decided SEV, set expectations around who shows up to which SEV

Something I noticed around your Salesforce limits today is that you're automatically opening incidents around these limit warnings from Salesforce. Most

of these incidents go without any response: the channel just gets closed after inactivity or the alert self-resolves.

This kind of thing can lead to the normalization of deviance.

Instead, it works a lot better if automated alerts cannot trigger incidents, but always page a human who, then, can open an incident. Incidents were only opened by humans at Gusto, and it meant that whenever there was incident, you knew that a human response was required, because a human being had made that determination. Incidents and alerts are not the same thing. An alert without a need for human intervention is not an incident.

Cost: 1, Benefit: 2 Maybe I'm nitpicking here. But I think a lot about signal versus noise in alerting, and opening incidents automatically feels like a guaranteed way for incidents to become ignored.

Recommendation: Increase observability of limit consumption by attaching it to a transaction.

In my experience, if you want to solve a problem in your codebase, you just have to make the problem visible to the developers that work on it. If you did even a half-decent job of hiring, your dev team is going to be full of people who wake up every day and want to work on a application they can take pride in. Devs are naturally internally motivated. Usually, if a problem isn't being solved, it's because that problem isn't being felt by the developers who work on it. Make them feel it, and they're naturally going to be inclined to solve it.

This is why observability initiatives can work so well.

What I suggest is that using any of the ratelimits discussed in previous recommendations: incrementing the "slow query" ratelimit, incrementing any of the ratelimits at all, should be attached as an attribute of the transaction in your observability tool. This then makes it trivial to sort transactions by how much of the ratelimit they consume. I'll use Datadog as an example because I know you're headed this way:

```
# This transaction had a slow query. After incrementing the query_signature dictionary, we:
current_span = Datadog::Tracing.active_span
current_span.set_tag('slow_queries', 'true') unless current_span.nil?
```

… or whatever you feel would be useful. My experience is that devs tend to think at the "transaction" level: the Sidekiq job or the controller action. Attaching relevant attributes to transactions is therefore the best way to get their attention to a particular issue.

Cost: 1, Benefit: 2 Easy to implement.

Recommendation: Limit Sidekiq job concurrency on long-SLA queues during day, crank them up at night

I'll use `ProcessDocSendJob` here as an example.

This job is on a 24 hour queue. And yet, if you look at the "Salesforce calls within the last 24 hours" chart on NewRelic, this job is routinely one of the highest-throughput callers of Salesforce for all of Glue! It can run at ~150 jobs per second, each calling Salesforce ~14 times. That doesn't make sense. Here's an operation that's incredibly latency-insensitive, and yet it's consuming resources at an extremely high rate.

These jobs are running on the hour (another anti-pattern: enqueueing lots of background job work *exactly on the hour*, which then results in outages *on the the hour* as concurrency suddenly blows up), during primetime during the day (in my view: 6am PT until roughly 7pm PT). It certainly looks like these jobs are being executed with an actual queue latency in the seconds, not hours or days, range.

High-SLA queues are meant to be slow most of the time. Running high-SLA queues at a queue latency much faster than their SLA means:

1. You'll never smoke out jobs which are misclassified in the wrong category until something REALLY bad happens. That's a bad time to find out that this job should have been in the 5 minute queue, not the 24 hour queue.
2. You're running at high concurrency than you need to.

You can take a couple of actions here:

1. You can scale to zero 12 or 24 hour queues during US primetime, either *all of the time* or only during "QOS events".
2. Reduce concurrency all of the time. Generally, long-SLA queues of an hour or more should be averaging about 50% of their SLA most of the time.

Cost: 1, Benefit: 3 This is probably the easiest "ratelimit" you'll get to implement, with the biggest impact.

Recommendation: Create queueable/backgroundable write functions for Salesforce (CQRS), with possible automatic batching

If our strategy is going to be to introduce "quality of service" levels to the Salesforce API, that means that a useful proportion of our Salesforce traffic can't have an "ASAP" or "realtime all the time!" quality of service requirement. Basically every web call will have this level of qos attached. So, moving calls out of web and into Sidekiq will allow us to bring more Salesforce traffic into the "throttleable" realm.

For read operations, it will probably be difficult to move any of these into the Sidekiq realm. If you're reading it during a web request, you probably are reading it because it could change the output of the response you're putting together.

However, for writes, this isn't necessarily the case. There's plenty of write load that doesn't actually affect the response.

This is one of the principles behind command query separation. If write operations do not have return values, they do not necessarily have to be blocking.

You could create a "background write" facility using Sidekiq that basically says "here's a POST I would like to execute, and I don't need the result, so do this at some point in the future". It can be quite simple. The caller could also attach SLA expectations to this write to show how "eventual" they can tolerate their "eventual consistency".

Once you have a queue of Sidekiq POSTs, you may even be able to automatically batch them. That's going to be extremely dependent on the workload and the API involved, far beyond what I could get in to in just this month of looking at the app.

Cost: 2, Benefit: 3 It's unclear to me the *degree* of load this could take off of web. It has the added benefit of decreasing latency on web though, because you're no longer blocking on the response from Salesforce for that write.

Recommendation: Use Sidekiq+Iterable for long running jobs, report LR jobs to owner

A critical precondition of the "Limit Sidekiq job concurrency on long-SLA queues" recommendation is that jobs do not, themselves, execute lots of Salesforce calls in a short amount of time.

Some jobs definitely do this, however. An example is `Negotiators::LegalAllocationsJob`, which takes ~30 minutes to execute and runs 12.5k Salesforce calls, or about 7 per second for 30 minutes. That's an incredibly high load relative to most of your other work.

It's also extremely dangerous for a job like this to be interrupted. SIGTERM can occur unexpectedly, at any time.

Whether or not this kind of job is idempotent is usually unpredictable. In my experience, they're usually not.

Using the new Iterable facility in Sidekiq is usually the best way to "create" idempotency in a long-running job.

Let's say we do hit a QoS limit halfway through this job, raising an exception. Now what? When we re-enqueue the job and run it again, are we just going to run another 12.5k Salesforce calls? That's not a very useful delay! Iterable would allow us to not repeat any calls we've already made.

In addition, an inventory of long-running jobs (>30 seconds, the sidekiq shutdown timeout) can and should be turned into a backlog of work.

Cost: 2, Benefit: 3 There aren't that many of these jobs running around, maybe 10 or so.

Recommendation: Surface dangerous conditions to the relevant team through code ownership

You've got about 50 people actively working on the `glue` repo each month. In the app and lib directories, there's about 80,000 lines of Ruby (and there's about 200,000 lines in spec, by the way, nice ratio). This is just about the point where "tragedy of the commons" scenarios start to occur. It's just beyond the size where everyone feels comfortable with keeping the whole codebase in their human "context" window.

When you start to lack context, making changes can have an unintended/unknown affect on either the system as a whole or you start to assume that "someone else will take care of that". Hm, this job has lots of slow Salesforce queries? Bummer, someone else will take care of that though. This web request calls Salesforce 50 times? Bummer, but I only touched that once and it was years ago, someone else will take care of that.

I saw code ownership work really well at Gusto for reducing the size of the commons. By the end of the effort, there really was very little "commons" code at all, and almost everything had an explicit team owner. CODEOWNERS is a good place to start for what ownership already exists.

Ideally, you use this team and ownership data to also route alerts. If a controller starts consuming too much of a rate limit, that alert goes straight to the team that owns the controller, or the background job, or whatever.

As I said before, engineers are motivated when they "feel the pain" of an app that's not performing. Notifications and alerts based on team greatly increase signal to noise ratio.

Cost: 3, Benefit: 3 Maybe you're not at the size yet where you feel this is relevant, I don't have the full context. I think you're probably *just* getting there.

Recommendation: Automated load shedding

This is the automated side of the "QOS playbook" I mentioned earlier. You have the 12 and 24 hour queues. You can automatically scale these down when bad conditions are detected and really no one would ever be the wiser, and humans don't have to be involved. Even 12 hour pauses on the 24 hour queue generally can go by without any notice whatsoever.

This can be triggered on an ongoing basis by anything checking `/limits`. If the daily API count is getting consumed too quickly, just shut down queues completely for a period of time and consume that SLA. View unused SLA as a resource to be manipulated, not avoided. You don't get gold stars for unused SLA.

Cost: 2, Benefit 3: I think the amount of load on these queues isn't that high,

but automating scale-to-low-or-zero for a couple of queues isn't that hard to do.

Outcome: Improve Response Times

Ok, on to the fun stuff!

Really, most of these are also going to have the effect of reducing the Salesforce call count, which means that ~20% of the effect of each of these recommendations is also going to be towards improving the previous "robustness" outcome.

There's really nothing the app does from a latency perspective *other than* to wait on Salesforce. There is no meaningful source of latency which is not Salesforce network wait. So, almost all of my recommendations focus on this.

Recommendation: Create and enforce a latency standard for web transactions

It's difficult to optimize in the absence of a requirement. With a requirement, we know which transactions need work done, and which can be left alone. "It should be faster than it is now" is a bad requirement because every transaction is always permanently in a "work needing to be done" state. This is a good recipe for burnout or indefinite procrastination. A requirement like "the p95 must be below 1 second" is measurable and actionable, and controllers which meet it do not need work done.

Almost every Rails controller action, in a normal world, can achieve a 250ms average and 1 second p95. However, `glue` is an atypical Rails application. Currently, it has a 665ms average and a 1.75 second p95. If you squint, that means its roughly twice as slow as what I think almost any Rails application can achieve.

If we instituted 250ms/1second as a standard today, roughly half of your popular controller actions would fail. This is probably too aggressive, given your priorities. Usually, I also like to limit latency standards based on traffic as well.

For example, my standard recommendation is "for our top 50 endpoints by throughput, p95 response time must be less than or equal to 500ms". Usually if ~10 or fewer endpoints violate that standard, it's a useful way to think about it. If you take this list and sort by p95, you get a rank-order list of "highly trafficked endpoints that have bad performance". In Datadog this is trivial, in NewRelic it's a little harder to produce with NRQL but not that bad.

I would suggest something like a 2 second p95 limit for the top 50 throughput endpoints. Eyeballing your transaction list now, transactions like Offers#show, Opportunities#create, and Pull_credit#create would fail. These all look like great candidates for optimization work.

Cost: 1, Benefit 3. Easy to do and gets people pointed in the right direction.

Recommendation: Create a longpolling utility for endpoints like ea/opportunities/send_ea

You have some endpoints which wait on very long external service queries, and not necessary Salesforce. `ea/opportunities/send_ea` is one such transaction.

It spends almost 20 seconds waiting on Docusign.

Keeping long transactions open in Puma isn't great for a number of reasons:

1. Hitting back/causing a retry is really expensive.
2. Easy to hit network timeouts if something goes wrong
3. You rarely have the same retry facilities available
4. Keeping Puma threads locked up waiting is not ideal, and increases queue times for web

This kind of pattern probably doesn't only occur here. You can provide a "facility" for "make this request and come back later" which looks like:

1. The API call returns a URL, which means "long-poll this URL with GET, eventually your resource will be there.
2. You can usually also provide some frontend niceties for making this sort of thing easy to do.
3. You can provide a Sidekiq/controller mixin that makes this pattern easy to implement.

Cost: 2, Benefit: 2

Recommendation: Emit the stack for each Salesforce query.

I've really beat the drum on tooling and observability in this report, and I'll do it again for the next few recommendations.

If you want to use Salesforce as your database, as your source of truth, than you need the same tooling that we get with ActiveRecord, but for your specific Restforce-powered context.

For removing SQL, one of my favorite techniques is to just use `verbose_query_logs` (which you have on already, nice) to look at where in the stack the SQL is coming from. You could do this yourselves in a Faraday middleware and use `caller_locations` with a backtrace filter to get the same information. It would be immensely valuable for figuring out where Salesforce calls are coming from, to see if they could be removed or combined.

Cost: 1, Benefit: 3 Pretty easy to do!

Recommendation: Add salesforce to rack-mini-profiler-like tool

One of my biggest weapons over the years has been `rack-mini-profiler`. It's traditionally not been geared towards JSON APIs, but that's something I'd like

to address. Whether this recommendation is fulfilled by RMP or some other tool isn't that important (although I am a maintainer on RMP now so I have the power to merge things).

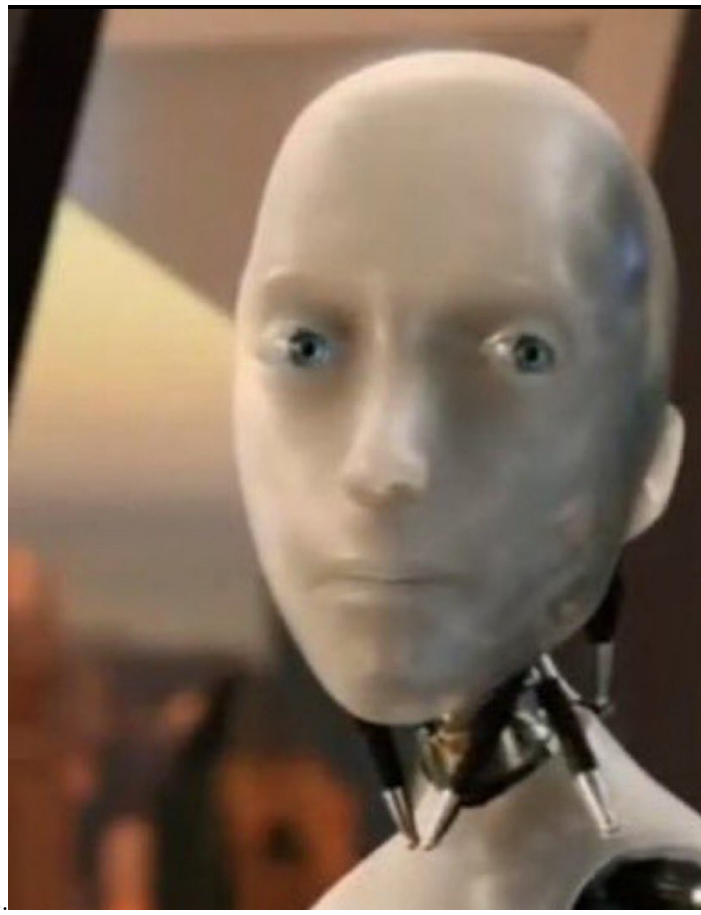
What you really need is a per-request trace of every Salesforce query, with the complete (and a filtered) stack for each query. This addresses two things I've wanted in RMP for a long time:

1. We should have "the speed badge" view of RMP accessible by a query parameter so you can use it with JSON APIs like yours.
2. We should report HTTP calls the same way we report SQL queries.

Fix both of those problems and RMP would be a massively useful tool for you. Some devs will just never read the logs, so my previous recommendation won't apply to them, but this one might be their preferred workflow. Sometimes you have to provide multiple ways to do the same thing.

Cost: 2, Benefit: 3

Recommendation: Create a parallel query collector



If you make parallel querying easy, people will do it. How your CPU looks at you while it's idling, waiting for your extremely long I/O call to finish.

I find it hard to believe that every single Restforce call is necessarily blocking.

If we were doing this at the HTTP level, it would look like:

```
require 'parallel'
require 'httparty'

class ParallelHttpCollector
  def initialize()
    @pending_requests = []
  end
```

```ruby
  def add(url, method: :get, headers: {}, body: nil)
    @pending_requests << {
      url: url, method: method,
      headers: headers, body: body
    }
  end

  def execute
    Parallel.map(@pending_requests, in_threads: 10) do |req|
      HTTParty.send(req[:method], req[:url],
        headers: req[:headers],
        body: req[:body],
        timeout: 30
      ).parsed_response
    end
  end
end
```

The key with these kind of collectors is to make them as narrowly focused as you can. That keeps them safe and reusable. If you make it too broad, you risk things like people calling this recursively, or causing SQL queries inside the block which blows up your AR db pool, etc.

Cost: 2, Benefit: 2. Probably difficult to decide how often this can be used.

Recommendation: Report request queue time (also in Datadog)

You currently are not reporting request queue time to New Relic. This can also be reported to Datadog.

Every single request to your backend from a browser client experiences latency along the following steps:

1. Time spent routing from the client to your server (network RTT, not really under your control)
2. Time spent routing inside your infra to a container (internal RTT, under your control but quite minimal time)
3. Time spent queueing, waiting for an empty Puma process (0 if any Puma process is available, but can be substantial if all are busy)
4. Time spent actually running the request (what Sentry reports).

Step 3 is a common snafu. You currently don't have this number recorded or reported anywhere. It can be a massive component in user response times if the deployment is under heavy load or is misconfigured. Imagine running a grocery store, but no one knew how long the lines for checkout were. That's what this is like.

Ideally, you scale up or down based on this number. When request queue time goes up, scale up. However, your load is so low and predictable, I don't yet recommend autoscaling. But you do need this number still so you can make accurate manual adjustments.

1. **Insert a time-since-epoch-in-milliseconds header** to each request at the first available point you can in your infrastructure. Usually this is the load balancer, sometimes the nginx/traefik/reverse-proxy in front of Puma. New Relic has instructions for this.
2. **In a rack middleware, subtract Time.now - request_start_time**. Report to elastic. You now have request queue time. New Relic already has a middleware here, you don't need to do anything.

Cost: 1, Benefit: 1. Traffic isn't a big deal for you, but I think this would be a short project.

Recommendation: Implement the Speedshop Standard Dashboard

Over the years now, we've arrived at a "standard dashboard". It is a set of 25 charts.

This is the dashboard:

- Experience
  - Page load time (all loads) (You will skip this, as an API only app.)
    - Page load time (initial/cold load)
    - Page load time (hot SPA route changes)
  - Time for interactions (i.e., time spent waiting on DOM/network for clicks that don't change the URL) (You will skip this, as an API only app.)
  - Time to execute customer-blocking background jobs. For any background job where a customer is actively waiting on the result and is blocked until that job completes (password reset email), tracks total time from enqueued_at until completion.
  - % of responses which took longer than 500ms, organized by controller action.
- Scalability
  - Web utilization
    - Total Puma process count
    - Concurrent request load (average req/sec * sec/req)
    - Process count / load
  - HPA/scaler status (web and workers)
    - current, min, max
  - Web request queue timing (p75,p95,pmax)
  - Worker latency
    - For each queue, show queue latency (and SLA for that particular queue)

- Reliability
  - Database, cache DBs, and Redis DBs
    - CPU (load and utilization)
    - IOPs (if limited)
    - Read/write latency
    - Error rates
    - Hit-rate (if cache)
  - Error rates
    - Web, worker
  - www.*.com uptime

Basically, most of these should also have an associated alert/monitor. Many of them should also have SLOs (like the last uptime number, or worker latency)

We have some standard terraform stuff for Terraform for implementing this. We can make this a retainer project when you move to Datadog next year.

Cost: 2, Benefit: 4: A good picture of what's happening is the foundation of future work. The remaining work to "fill out" the dashboard is nearly done.

Recommendation: Terraform your perf monitors and SLOs

I'm not asking your to IaC your entire infrastructure with this recommendation, but I do think you'd find it useful to IaC your Datadog/NewRelic setup.

Almost every team I work with eventually goes this route. It just becomes *far* easier to manage X number of SLOs for your queues when adding a queue to the setup is as simple as adding one line to a terraform file and hitting "merge", with everything auto-applied by a Github Action. It also adds a layer of accountability, history and "why did we change that?" support.

Cost: 2, Benefit: 2. Mostly the benefit is for the future when any of this needs to be changed, but it's a low-effort project.

Recommendation: Alert the dev if a Restforce table is accessed more than once

One of my principles for SQL access is "you should only access each table once". It's a kind of impossible goal, you can rarely ever actually do that and sometimes it's not even optimal. But it's a good north-star. Phrased another way, "you should always access a table one less time than you are right now!".

It would be useful to surface (in the log or somewhere else) any time a Salesforce table is accessed more than once in a transaction. It would probably just be another Restforce middleware that adds query counts to a thread variable, which can then be loggged and cleared out at the end of each transaction.

Cost: 2, Benefit: 3.

Recommendation: Create more realistic seeding in dev

To bang on the "devs will solve problems if they feel the problem" drum one last time, one big reason I see devs not fixing N+1s is because they don't have a realistic seed to work with. Glue is no exception, the seeds are pretty sparse. Working with realistic data locally is basically a prerequisite for finding and fixing database access.

There are a lot of ways to try and do this. A download from production of any non-PII table might be a good place to start. For tables with PII (Account, etc), the app is *perhaps* small enough that you could simply maintain a realistic seed here. The thing is not the *quantity* of data but the *complexity* of it. You want seed data which exercises every feature, every edge case, every nook and cranny of the code.

Cost: 3, Benefit: 4

Outcome: A Few Misc Things

Recommendation: Replace sidekiq-unique-jobs with Mike's code

File this one under "potential footgun". You pay for Sidekiq Enterprise, but you're using the OSS `sidekiq-unique-jobs` gem instead of Mike's "official" unique jobs implementation. Mike's uses fewer Redis operations, which means less load and fewer headaches for you. Plus, you're paying for it, which means you're paying for Mike Perham on support, and why not take advantage of that?

Cost: 3, Benefit: 3 Your overall volume is low enough that "Redis load" probably has never been a problem for you.

Recommendation: Sidekiq Datadog Integration

Another thing you're paying for but not taking advantage of YET is the very good Sidekiq/statsd integration. This is just a note to install it when you make the Datadog transition.

Cost: 1, Benefit: 3 It's really very good, and observability on queue latency is huge.

Recommendation: Migrate to Datadog

I love Datadog. It's by far my preferred vendor to New Relic. I'm glad you're making the switch. I don't need to say much more, because you're already going down this path, other than to keep going!

Cost: N/A, Benefit: N/A

Recommendation: Index all _id columns, enforce this via Github Action

We have a standard Github Action for enforcing that all columns which end in _id must be indexed. This action does require a `schema.rb` file, which you don't have.

The alternative is to use Active Record Doctor, which we've also done in the past, it's just a bit harder to install. Having the ability to check this on every pull request is just such an easy win that guarantees an entire class of mistake will no longer occur, so we're telling almost everyone to implement this.

Cost: 1, Benefit: 2

Summary

- Outcome: Reduce Outage Incidents Caused By Salesforce Integration
  - Recommendation: Limit Sidekiq job concurrency on long-SLA queues during day, crank them up at night *Cost: 1 Benefit: 3*
  - Recommendation: Create QoS degradation playbooks *Cost: 1 Benefit: 3*
  - Recommendation: Use Sidekiq+Iterable for long running jobs, report LR jobs to owner *Cost: 2 Benefit: 3*
  - Recommendation: Create queueable/backgroundable write functions for Salesforce (CQRS), with possible automatic batching *Cost: 2 Benefit: 3*
  - Recommendation: Increase observability of limit consumption by attaching it to a transaction. *Cost: 1 Benefit: 2*
  - Recommendation: Do not open incidents automatically, require humans to open incidents with a human-decided SEV, set expectations around who shows up to which SEV *Cost: 1 Benefit: 2*
  - Recommendation: Attach QoS expectations to every SF request, let the ratelimiter decide *Cost: 4 Benefit: 5*
  - Recommendation: Pass all Salesforce API requests through a single, rate-limited service *Cost: 4 Benefit: 5*
  - Recommendation: Create a long-query lock *Cost: 3 Benefit: 4*
  - Recommendation: Surface dangerous conditions to the relevant team through code ownership *Cost: 3 Benefit: 3*
  - Recommendation: Automated load shedding *Cost: N/A Benefit: N/A*
  - Recommendation: Maintain internal ratelimits for all relevant SF limits *Cost: N/A Benefit: N/A*
- Outcome: Improve Response Times
  - Recommendation: Implement the Speedshop Standard Dashboard *Cost: 2 Benefit: 4*
  - Recommendation: Emit the stack for each Salesforce query. *Cost: 1*

Benefit: 3
  - Recommendation: Create more realistic seeding in dev *Cost: 3 Benefit: 4*
  - Recommendation: Alert the dev if a Restforce table is accessed more than once *Cost: 2 Benefit: 3*
  - Recommendation: Add salesforce to rack-mini-profiler-like tool *Cost: 2 Benefit: 3*
  - Recommendation: Terraform your perf monitors and SLOs *Cost: 2 Benefit: 2*
  - Recommendation: Report request queue time (also in Datadog) *Cost: 1 Benefit: 1*
  - Recommendation: Create a parallel query collector *Cost: 2 Benefit: 2*
  - Recommendation: Create a longpolling utility for endpoints like ea/opportunities/send_ea *Cost: 2 Benefit: 2*
  - Recommendation: Create and enforce a latency standard for web transactions *Cost: N/A Benefit: N/A*
- Outcome: A Few Misc Things
  - Recommendation: Sidekiq Datadog Integration *Cost: 1 Benefit: 3*
  - Recommendation: Index all _id columns, enforce this via Github Action *Cost: 1 Benefit: 2*
  - Recommendation: Replace sidekiq-unique-jobs with Mike's code *Cost: 3 Benefit: 3*
  - Recommendation: Migrate to Datadog *Cost: N/A Benefit: N/A*