Trustworthy graceful degradation

Failure tolerance across service boundaries

Daniel Rodgers-Pryor
CTO @ Stile Education

Stile Education

Used by ⅓ of Australian Science Students (Years 7-10)

13 engineers and growing! Join us:
stileeducation.com/who-we-are/engineering-at-stile/

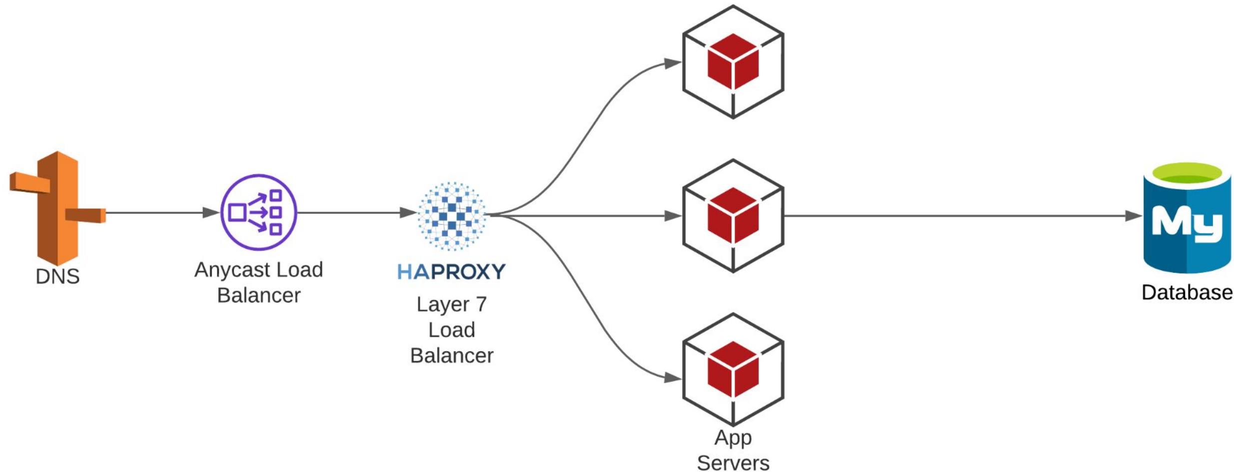
A world-class science education for every student

Stile helps teachers bring their science classes to life with beautiful lessons based on real-world science and global issues.

Set up a trial

A Highly Available Web Service

Simple, elegant, effective



Who works with a service like this?

...or really does it look more like this?

Asset cache

Slack

Cache

Blob Storage

Video Encoding

Email

MongoDB

Event Queue

OLAP Database

DNS

Anycast Load Balancer

HAPROXY

Layer 7 Load Balancer

Database

Replica1

Replica2

Message Queue

CRM

App Servers

File Encoding

Search Database (&Logs)

Business Intelligence

Metrics

Tracing

Support

Wiki

TTS

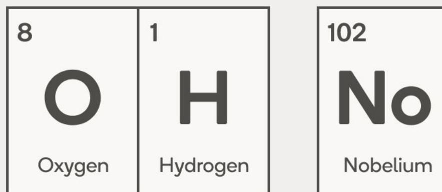
Image Encoding

Does your service work when all of those other things are down?

Story: How problems can happen

- MySQL is our primary database for core services
- Mongo was used a secondary database for less critical services
- Core services were supposed to work without it

Oops.



Stile is offline

We're incredibly sorry for this interruption.
Our engineering team is aware of the issue and is working as fast as possible to get Stile back up and running.

Check server status

The good news is that Squiz is still online and working.
Every topic from Stile has a Squiz activity, so your students can use it to continue learning.

Root cause

- Mongo went down
- Some of our core services depended on non core services
- Those non-core services depended on Mongo

Postmortems and Action Items to the rescue!

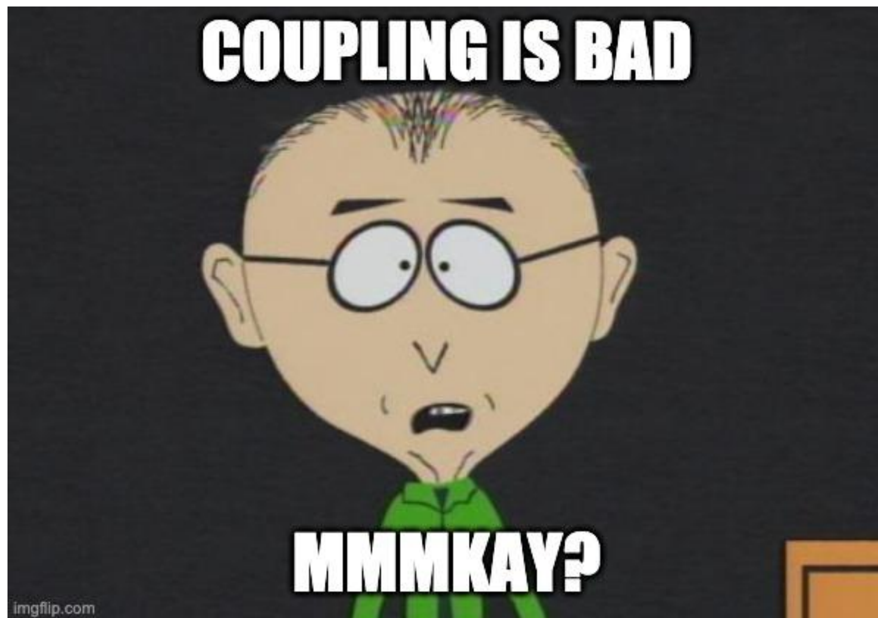
Establish the root cause.

Fix the immediate issues.

Remind all of our engineers that they're not supposed to depend on Mongo in core services.

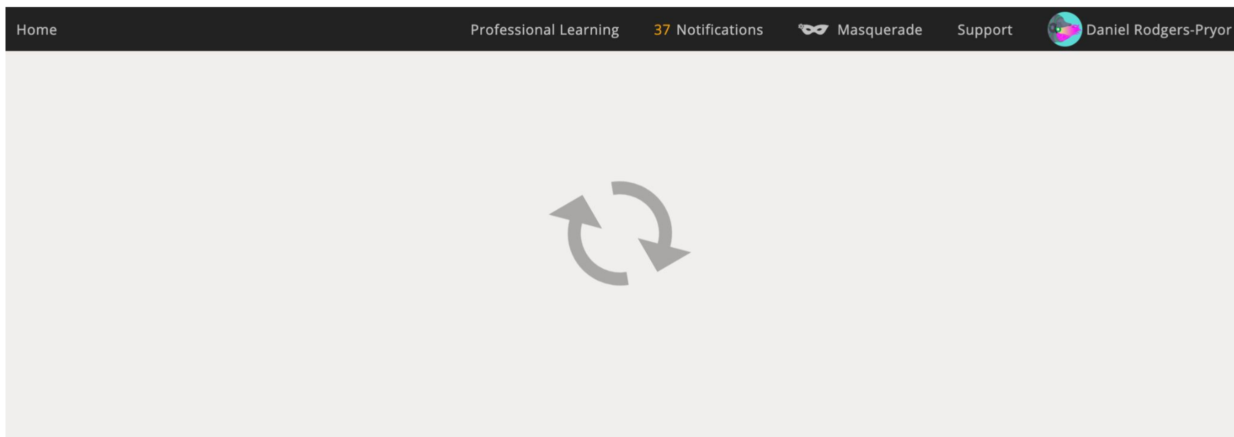
And we're done!

That'll fix it.



But then it happens again

Mongo is down, and this time, Stile seems… really slow?



But we *fixed* the coupling, and we reminded everyone not to do it again!
How could it break? What's going on.

Root cause

...it's the frontend.

```
loginActionQueue.whenLoaded({
    endpoint,
    principal,
}).then(
    () => {
        // Once the queue is loaded, start moving through it
        navigate("/postLoginActions/next");
    },
    () => {
        // Abort if fetching the queue fails
        finishedLoginActions();
    },
);
```

And it's not just error handling: timeouts matter too.

It’s easy to fail ungracefully

- Accidental dependencies between services
- Coupling through workflows in the frontend
- Buggy/broken error handling code
- Conditional coupling
- Rare (unhandled) error states:
    - Bad DNS caching
    - Database node failover
    - Very slow responses

1. Failing gracefully: Understand all of the possible error modes

- Application specific errors (eg. database too-many-connections error, authorization error etc.)
- TCP connection errors
- TCP timeout ← **this one is easy to forget!**

2. Failing gracefully: Retry what you can

- HTTP 503 (service unavailable) — probably a good idea to retry
  - Can be service specific: Eg. S3 misuses HTTP 503 to mean SLOW DOWN
- One node down in a cluster? Mark it down, pick another and retry
  - Don’t retry connecting to down nodes on every request
    - You might storm them with connections and make it hard to recover
    - You’ll definitely slow down your ability to serve requests
  - Set a timeout for retrying connections to the node

3. Failing gracefully: ...and fail helpfully where you can't

- HTTP 4xx (client errors) — retrying probably won't help (and might hurt if it's HTTP 429)
- Timeout? Probably not a good idea to retry:
    - The service might slow because it's overloaded, so retrying will make it worse
    - Resource exhaustion: if a downstream service is taking too long to respond, then lots of your request-handler threads/processes will be tied up waiting for a response: don't make the problem worse!
    - If you've already hit a timeout, then your downstream client might be about to time out waiting for *your* response

Log a detailed error and return 503 (Service Unavailable) to the client. Make it easy to see where the error occurred and what it was rather than leaving a trail of silent timeouts.

Expect your clients to retry if they need to.

3. Failing gracefully: It's not you, it's me

Returning 503 and letting clients retry is a great way to work around host-specific problems.
But you don't want to keep accepting traffic and returning errors.

If a process can't service any requests — especially critical ones — then maybe it's time to give up.

This can help if the problem is isolated to the process or host. Eg.

- Bad NIC
- Bad DNS cache
- Bad persistent connection state
- Corrupted memory

3. Failing gracefully: It’s not you, it’s me

If the errors are isolated to unimportant downstream services and aren’t affecting high-priority requests, then obviously this doesn’t apply.

If critical errors persist across restarts, then your startup health checks should stop you from accepting more traffic (and ideally cause the possibly-broken instance to be replaced)

4. Failing gracefully: First do no harm

Don’t make it worse by retrying, reconnecting and restarting!

Retrying and reconnecting can amplify traffic to overloaded services.

Restarting can cause a storm of reconnection attempts, and can starve your load balancer of healthy backend targets.

THE MONGO LOAD AVERAGE
IS OVER 9000!!
imgflip.com

4. Failing gracefully: First do no harm

If there's obviously poisoned connection state or memory corruption (eg. SEGFAULT), then crash and crash fast.

For everything else: coordinate restarts. Report as unhealthy and have your orchestrator restart a limited number of services at a time.

Consider two kinds of healthcheck:

- For the load balancer: Can I accept requests?
- For the orchestrator: Should I be restarted?

And even that isn’t enough

That was a lot of competing requirements to deal with! But even if we handle every error perfectly, that’s not enough.

These errors are each so rare that there will be months or years between each occurrence.

How can you be confident that it still works when you need it?

When fixing the problem isn't enough

Many of you are thinking: Chaos engineering can help here!

But there's an even more powerful tool, one that let's developers find, fix and learn from their mistakes without the SRE team ever hearing about it.

You already have the solution: CI

You can test (pretty much) all of this in your CI pipeline, run it for every change, and avoid nasty coupling from reaching prod.

Stile / Big Friendly Pipeline / master
Trigger other pipelines and collate results.

All Builds | Edit Steps | Pipeline Settings | New Build

Merge pull request #19887 from StileEducation/dependabot/bundler/random-tools/analyze-content/sorbet-runtime-0.5.9115
Build #603537 | master | 7ef2a4d6a5
Passed in 1h 18m and blocked

Configure pipeline | SLB untested manifest | Tracing untested ma... | Build Backend AMI | Dev workstation bootstr... | Elasticsearch unteste...
Cockroach untested ... | Deployment Agent A... | Backup3 untested m... | Armchair General ... | Deploy data-warehouse?
Docker Registry unte... | tested public-api2 (i... | Prober untested mani... | Prometheus untested... | Stile app untested m...
AWS Primary Carer | Stile app tested ... | Build Async AMI | Buildkite Annotations | Certbot untested ma... | Logstash untested m...
Stile isolated unteste... | Buildkite AMI manifest | Mongo untested man... | Consul untested man... | Stile Data Warehouse...
Data Warehouse... | Stile untested produc... | Production AMI manif... | Prod Data Wareh...

Peter Gates
Created today at 7:41 PM | Triggered from Webhook

Rebuild | Cancel

1. Unit test your service clients

```ruby
context 'when handling a too many connections error' do
  # Overload the provided db. Returns a block that can be called to wait on the results.
  def overload(db, swallow_errors: true)
    # Keep going until we get to a too many connections error
    # The exact number of connections will vary base on other database activity.
    # DO NOT configure this test against the shared CI RDS database. It is intended
    # to be run against a local database, otherwise it will cause other test runs to fail.
    # We'll need a shitty thread pool on our end too to manage these connections.
    queries = []
    (1..(MAX_CONNECTIONS + 10)).each do
      # Try to make a connection, expecting it may fail with 'too many connections'.
      queries << Thread.new do
        begin
          db.run 'SELECT SLEEP(2)'
        rescue => err
          puts "Time-waster query exited with an error (this is probably normal): #{err.inspect}"
          raise unless swallow_errors
        end
      end
    end
    return -> { queries.each { |thread| thread.join } }
  end

  # Explicitly do retry this test. It may well flake.
  it 'does not disconnect healthy connections', retry: 5 do
    # Set up a DB pool with the purpose of overloading it.
    ::Sequel.connect(*opts) do |too_many_connections_db|
      # I had grand plans to run an API query here 'for realism', but it just
      # seems to make this test much more flaky due to the increased complexity.
      # api = factory.get(TestApi)

      waiter = overload(too_many_connections_db)

      # Before we sleep for a bit, there'll probably be 50 to 100 connections that've managed to start.
      _debug_get_current_connection_count(too_many_connections_db)

      # Give the threads a chance to start and connect to MySQL.
      sleep 2

      # After we've slept for a bit, there should be 151 or more. The attempt to measure them will likely fail with a 'too many connections' error - this is good and normal.
      _debug_get_current_connection_count(too_many_connections_db)

      # Future connections should cause a 'too many connections' error to be handled.

      # Check we do log about max connections, and DON'T log the 'resetting all connections' message or call that function.
      expect(Stile.log).to receive(:error)
        .with(
          'MySQL connection error due to having reached the max connections limit!',
          anything,
        )
        .at_least(:once)
      expect(Stile.log).not_to receive(:error).with(
        'MySQL connection error. Requesting graceful restart and raising Stile::ServiceUnavailable instead.',
        anything,
      )
```

```ruby
context 'when there is a connection error' do
    let(:error_count) { 1 }
    let(:setup) do
        super()

        callcount = 0
        allow_any_instance_of(::Redis).to receive(:watch)
            .and_wrap_original do |m, *args, &block|
            callcount += 1

            if callcount <= error_count
                raise Redis::ConnectionError.new
            end

            m.call(*args, &block)
        end
    end

    it 'marks the host as down and returns false' do
        expect(result).to be false
        expect { shard_connection.get(key2) }.to raise_error {
            ::Redis::DisconnectedError
        }

        # Wait for a reconnection to be attempted
        retry_for(
            timeout_seconds:
                Stile::RedisCache::RECONNECT_THROTTLE_PERIOD + 1,
        ) do
            # The operation shouldn't have been applied
            expect(shard_connection.get(key2)).to eql('1')
        end
    end

    context 'which takes a long time to happen (eg. a timeout)' do
```

2. Integration test your minimum viable system

You need to know what your key workflows are and have a test suite for them, ideally an integration test suite which includes the frontend.

We use a suite of smoke tests that we built to run on each deploy.

But take whatever kind of service integration tests you've got running on CI, and run them with no unnecessary dependencies.

Now instead of conversations like this

YOU BROKE PRODUCTION!   HAHA CI GOES BRRRRRRR

You get to have conversations like this

Tia 5:10 PM

I've got errors I can't figure out in non-prod routes tests, dirt api tests and prober tests without non essential services and lsr_gem is failing my clean_build. Is anyone free to help me at this late time of the day?

12 replies Last reply 9 days ago

And quickly find the source of the problem

```
F, [2021-09-06T23:57:18.497703 #6] FATAL -- public_api2: Received an error while configuring service
---
error:
  :message: '<Stile::ServiceUnavailable "Service unavailable: Mongo connection could
    not be established. Please try again later.">'
  :stack:
  - "/app/localgems/lib-stile-ruby/lib/stile/dependency_factory.rb:607:in `block (2
    levels) in get'"
  - "/app/localgems/lib-stile-ruby/lib/stile/dependency_factory.rb:351:in `synchronize'"
  - "/app/localgems/lib-stile-ruby/lib/stile/dependency_factory.rb:351:in `synchronize'"
  - "/app/localgems/lib-stile-ruby/lib/stile/dependency_factory.rb:578:in `block in
    get'"
  - "/app/localgems/lib-stile-ruby/lib/stile/dependency_factory.rb:965:in `detect_cycles'"
  - "/app/localgems/lib-stile-ruby/lib/stile/dependency_factory.rb:576:in `get'"
  - "/app/localgems/lib-stile-ruby/lib/stile/dependency_factory.rb:169:in `block (2
    levels) in declare_dependencies'"
  - "/app/lib/orc_service/migrations/migrate_latest_seen_to_mysql.rb:26:in `run'"
  - "/app/localgems/lib-stile-ruby/lib/stile/amqp/service.rb:318:in `block in configure'"
  - "/app/localgems/lib-stile-ruby/lib/stile/amqp/service.rb:318:in `each'"
  - "/app/localgems/lib-stile-ruby/lib/stile/amqp/service.rb:318:in `configure'"
  - "/app/lib/public_api.rb:210:in `block in create_service'"
  - "/app/localgems/lib-stile-ruby/lib/stile/threaded_service.rb:40:in `block (2 levels)
    in initialize'"
  - "/app/localgems/lib-stile-ruby/lib/stile/process.rb:633:in `block in configure_process'"
  - "/app/localgems/lib-stile-ruby/lib/stile/synchromesh.rb:37:in `block (2 levels)
    in run_outside_reactor'"
  - "/app/localgems/lib-stile-ruby/lib/stile/correlated_thread.rb:17:in `block (2
    levels) in initialize'"
  - "/app/localgems/lib-stile-ruby/lib/stile/exception_handling.rb:139:in `block in
    with_rescue_handlers_outside_em'"
  - "/app/localgems/lib-stile-ruby/lib/stile/exception_handling.rb:107:in `with_rescue_handlers_inside_em'"
  - "/app/localgems/lib-stile-ruby/lib/stile/exception_handling.rb:139:in `with_rescue_handlers_outside_em'"
  - "/app/localgems/lib-stile-ruby/lib/stile/exception_handling.rb:10:in `with_rescue_handlers'"
  - "/app/localgems/lib-stile-ruby/lib/stile/correlated_thread.rb:17:in `block in
    initialize'"
```

3. Chaos Testing

If you think you don’t need a service, try shutting it down. In prod. Right now.

Make rare failures common enough that you’ll have confidence dealing with them

3. Chaos Testing: Limitations

Chaos testing is great for auto-healing systems where you expect to handle the failure without disruption:

- Kill a process
- Kill a box
- Kill an AZ
- Kill a whole region

Expect it to be replaced, and expect your load balancers to route around the problem in the meantime.

If your strategy for handling a failure is **graceful-degradation**, then the decision is tricker.

You expect — even in the best case — to degrade the user experience. This can be a good way to spend your spare SLO headroom, but you don't want to **stop** testing your infrastructure when you hit your SLO limits: you want to invest more!

4. Advanced Technique: Request 'Cursing'

How to get the benefits of chaos testing optional dependencies without actually degrading the experience for your users.

Build a system for flagging individual requests with Eg. 'pretend that mongo is down'

4. Request Cursing: Implementation

Then have your driver/client code pretend that they are down for those request!

```
module Mongo
    class Server
        class Connection
            # This is the most convenient place that I could find in the mongo driver to wrap all connections.
            alias_method :_old_stile_violence_dispatch, :dispatch
            def dispatch(*args)
                if Stile::AMQP::Correlation.curse_mongo_requests_until &&
                        Time.now <
                            Stile::AMQP::Correlation.curse_mongo_requests_until
                    # All attempts to communicate with Mongo should
                    # be "cursed" for this request, to simulate some
                    # kind of spooky failure with Mongo itself.
                    #
                    # This is an "in vivo" reliability testing tool.
                    #
                    # Note that we specifically _don't_ request a graceful restart
                    # for Mongo connectivity errors encountered during coursed requests.
                    # We're not trying to test the service's reconnection behaviour here,
                    # but rather the behaviour _within_ a given request.
                    #
                    # Restarting services in production in response to cursed requests
                    # would also make us more timid about cursing requests regularly
                    # (e.g. in a special kind of "cursed probers") which is the opposite
                    # of what we want!
                    raise ::Mongo::Error::SocketTimeoutError.new
                end
```

4. Request Cursing: The Result

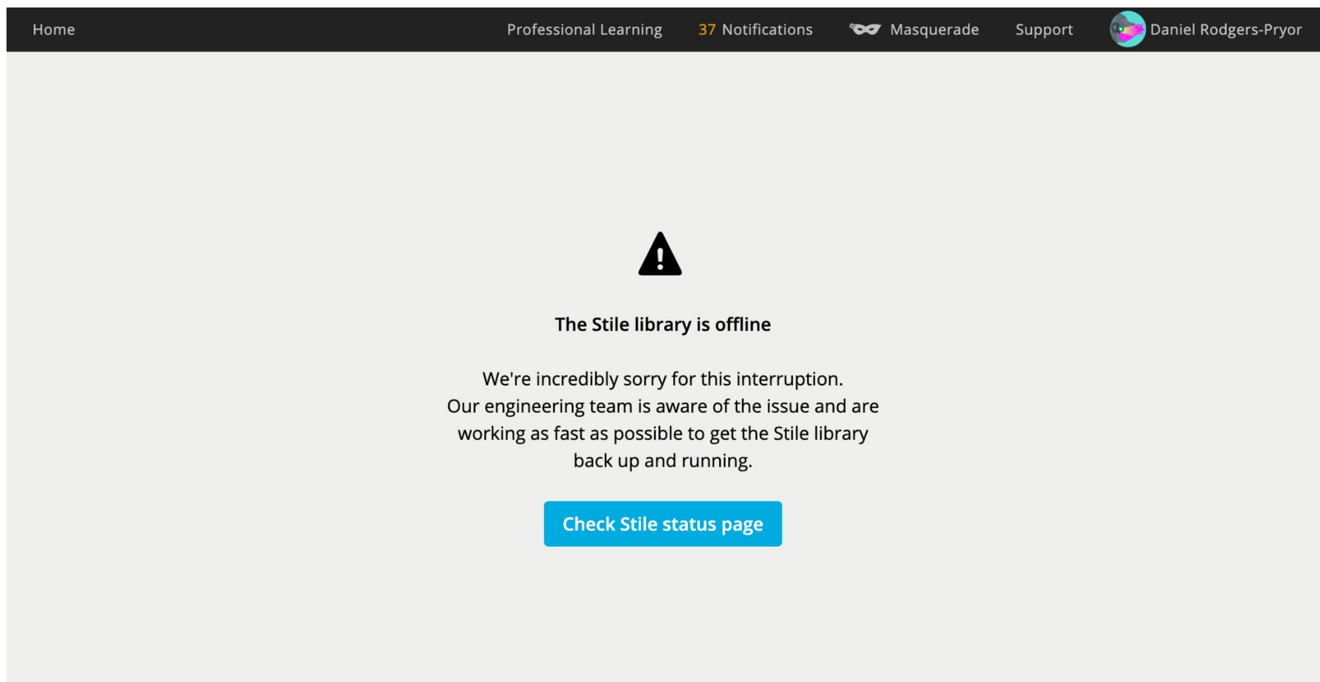
```
✓ ~/dev-environment [master {origin/master}|⚑ 27] $ curl -I -X GET 'https://stileapp.com/api/au/v3/compilations/5b3439f3-ba63-4365-8221-8fc9c892c002'
HTTP/2 200
content-type: application/json
vary: Origin,Accept-Encoding
x-content-type-options: nosniff
content-security-policy: sandbox
x-stile: 1
x-stile-echo:
strict-transport-security: max-age=31536000; includeSubDomains; preload
server-timing: slb_region;desc=slb-prod-apse2
x-stile-req-window: 60
x-stile-req-allowed: 350
x-stile-req-actual:
```

4. Request Cursing: The Result

```
✓ ~/dev-environment [master {origin/master}|⚑ 27] $ curl -I -X GET 'https://stileapp.com/api/au/v3/compilations/5b3439f3-ba63-4365-8221-8fc9c892c002' -H 'X-Stile-Curse-Mongo-Requests: true'
HTTP/2 503
content-type: application/json
vary: Origin,Accept-Encoding
content-length: 62
x-content-type-options: nosniff
content-security-policy: sandbox
x-stile: 1
x-stile-echo:
strict-transport-security: max-age=31536000; includeSubDomains; preload
server-timing: slb_region;desc=slb-prod-apse2
x-stile-req-window: 60
x-stile-req-allowed: 350
x-stile-req-actual: 1
```

Request Cursing: Use Cases - Manual Testing

Teach your client to curse all requests it sends, then browse around and see what the actual user experience is like when your downstream service is down.



Request Cursing: Use Cases - CI

If it’s a pain to actually break or run without some dependencies in CI (eg. you use shared, persistent testing infrastructure), then you can instead configure your test client to curse it’s requests and get a similar effect.

Don’t forget to also test that your services can start without optional dependencies through!

Can also be used to simulate rare failure modes (eg. TCP timeout, obscure application-level errors) which can’t be reliably generated by the real service during tests.

Request Cursing:
Use Cases - Test More Abstract 'Services'

You don't just need to curse an isolated service like a single database. Example:

- We annotate all internal and external API methods with SLO-level metadata
  - We use this to shed load by dropping non-critical requests when needed
  - Also for granular monitoring of API uptime and responsiveness

```
service WorksheetItemService {
    rpc Get(GetRequest) returns (GetResponse) {
        option (stile.http).slo = L1;
        option (stile.http).route = "/worksheetItems/:id";
        option (stile.http).method = GET;
    }
    rpc GetVersion(GetVersionRequest) returns (GetResponse) {
        option (stile.http).slo = L2;
        option (stile.http).route = "/worksheetItems/:id/versions/:serial_number";
        option (stile.http).method = GET;
    }
    rpc Undelete(UndeleteRequest) returns (GetResponse) {
        option (stile.http).slo = L3;
        option (stile.http).route = "/worksheetItems/:id/actions/undelete";
        option (stile.http).method = POST;
        option (stile.http).internal_admin_api_only = true;
    }
```

Request Cursing:
Use Cases - Test More Abstract 'Services'

- **But how do we know that those SLO labels are accurate? Cursing!**

  Eg. X-Stile-Curse-Slo-L3-Requests: true

- **Just like with testing single services, we can:**
  - Manually test how the whole system will behave with selective load-shedding enabled
  - Block merges in CI if critical (SLO L1) workflows don't succeed when all L2 and L3 APIs are shedding requests

Request Cursing: Implementation Considerations

- Propagate your request local data to downstream services to fully see the impacts
- Set an expiry on the curse (so that async jobs will eventually succeed and not leave the system in an inconsistent state forever)
- Consider restricting access to avoid exposing more surface area to an attacker

```
if env.has_key? 'HTTP_X_STILE_CURSE_MONGO_REQUESTS'
    # We don't want to allow just *anyone* to curse requests,
    # because curses can cause real (but relatively innocuous)
    # errors in a running prod stack and might trigger odd error
    # handling pathways and otherwise allow an attacker more
    # access than they'd usually have.
    unless @auth.is_group_member?(
               principal: principal,
               group: 'global_dark_wizards',
           )
      raise Stile::PermissionDenied.new
    end
```

Summary

- Consider **all** of the dependencies of your system, not just the core ones
- Consider all of the ways that they can fail, and handle them
- Unit test your error handling
- Integration test your system without optional dependencies
- Integration test the user experience of complex failure modes
- Use chaos engineering and request cursing to check that it keeps working in production

Stile: We’re Hiring!

stileeducation.com/who-we-are/engineering-at-stile

We're a small, diverse, tight-knit team with a mission to radically improve mainstream science education at schools. By creating world-class science lessons, coupled with intuitive tools that allow teachers to take advantage of the latest pedagogies, we’re already helping hundreds of thousands students in Australia get excited about science every week.

Stile is already used in 1 out of 3 Australian schools, and over the next few years, we're striving to perfect our product and bring our lessons to the rest of the world. This is an opportunity to have a big influence on education from within a small, high impact team.