npm's first rust service

the problem: add a feature to npm's simplest microservice

why was this non-trivial?

→ the microservice was old: vintage 2014

→ not git-deployable

→ used old configuration (like, really old)

→ no modcons

Chris's challenge to me:
don't just add the feature
but rewrite it in Go

requirements

→ a **public** proxy in front of license-api

→ proxy GET requests from npm-e installations to fetch license data

→ proxy POST callbacks from Stripe

requirements

→ a **public** proxy in front of license-api

→ proxy GET requests from npm-e installations to fetch license data

→ proxy POST callbacks from Stripe

→ *new!* send the Stripe callbacks to a 2nd destination!

I spent an afternoon rewriting the existing javascript service

fallback plan

familiarity with the problem

let's look at code (condensed quite a bit)

```
var server = require('restify').createServer({ name: 'public-license-api' });
var proxy = require('http-proxy').createProxyServer();

Monitor(server); // <--- add our monitoring hooks
server.get('/license/:productId/:billingId/:licenseKey', fetchLicense);
server.post('/stripe/callback', stripeCallback);

server.listen(process.env.port, process.env.host || '0.0.0.0', function() {
    logger.info(`starting public-license-api on port ${process.env.PORT}`);
});
```

and the request handlers look like this

```
function stripeCallback(req, res, next)
{
  proxy.web(req, res, { target: process.env.PRIVATE_LICENSE_API });
  next();
};
```

So. Little. Code.

task: learn Go, then reimplement

I bounced off Go hard

no per-project dep management

no package registry

verbose language with no payoffs

no new ideas since the 70s

give up? nah.

I decided to give Rust a try.

systems language

aka designed for writing systems

canonical examples: C & C++

more direct access to hardware/memory

more control / more responsibility

PITA vs performance tradeoff

systems languages are what everything else is implemented in

Mozilla invented Rust to write their next browser in "safe, concurrent, practical language"

modern FP language features

no garbage collection

a compiler that does its best to help

no exceptions, only return values

`match` on the Some<T>, None option

```rust
fn get_env_var<'a>(name: &'a str) -> std::string::String
{
    match env::var_os(name)
    {
        Some(v) => v.to_string_lossy().into_owned(),
        None => String::from(""),
    }
}
```

best feature: cargo/crates.io

modern dep management

a package manager based on semver

writing an http proxy in Rust is crushing a walnut with a piledriver

perfect for learning

because the problem itself is trivial

we have perf-critical work coming up

best to get experience *now*

So I dove in.


How hard can it be?

here's the spine of the app

rust-flavored this time

```
fn main()
{
    let mut server = Pencil::new("stripe-receiver");
    let metrics = get_env_var("METRICS");
    let port = get_env_var("PORT");
    let host = get_env_var("HOST");

    monitoring::monitor(&mut server, metrics); // Logging is now enabled.

    server.get("/license/<product_id:string>/<billing_id:string>/<licence_key:string>", "license", fetch_license);
    server.post("/stripe/callback", "stripe", handle_stripe);

    let listen_path = format!("{}:{}", host, port);
    info!(slog_scope::logger(), "listening on {}", listen_path);
    server.run(&*listen_path);
}
```

Pencil: http framework

inspired by flask

familiar to users of restify

first step: implement /ping

```
pub fn ping(_: &mut Request) -> PencilResult
{
    Ok(Response::from("pong"))
}
```

Not so bad!

next: /status

which does a deeper look...

```
blog-posts|⇒ http GET localhost:4701/_monitor/status
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 226
Content-Type: application/json
Date: Thu, 10 Nov 2016 18:42:33 GMT

{
    "averageRequestRate": 0,
    "git": "ef13da9",
    "message": "ef13da9 Merge pull request #6 from npm/ceej/modernize",
    "name": "public-license-api",
    "pid": 26206,
    "rss": {
        "heapTotal": 27381760,
        "heapUsed": 18249472,
        "rss": 47382528
    },
    "uptime": 14.357
}
```

→ uptime

→ memory use

→ PID

→ git commit hash

→ git commit message

→ request rate!

yak shave time

ceejbot/git-latest-commit

my first Cargo crate

a build dep that writes a file

logging & env vars

```
let port = get_env_var("PORT");
let host = get_env_var("HOST");
let listen_path = format!("{}:{}", host, port);
info!(slog_scope::logger(), "listening on {}", listen_path);
```

chose `slog` for logging
global scope, can do json

```
pub fn status(_: &mut Request) -> PencilResult
{
  let pid = format!("{}", psutil::getpid());
    let thisproc = Process::new(psutil::getpid());
    let rssbytes = match thisproc
    {
        Ok(v) => v.rss,
        Err(e) => { println!("{:?}", e); 0 }
    };

  let mut seconds = get_now_millis();
  unsafe { seconds = seconds - starttime; }

  let uptime = format!("{:.0}", seconds);
  let rss = format!("{}", rssbytes);

  let mut status = BTreeMap::new();
  status.insert("name", "stripe-receiver");
  status.insert("pid", &*pid);
  status.insert("version", env!("CARGO_PKG_VERSION"));
  status.insert("uptime", &*uptime);
  status.insert("rss", &*rss);
  status.insert("git", GIT_HASH);
  status.insert("message", GIT_SUMMARY);
  return jsonify(&status);
}
```

omg metrics

numbat-metrics/rust-emitter

numbat-emitter is deeply idiomatic node

re-writing it in rust has been work

→ JSON is, well, native to JS but not to Rust

→ current API doesn't hide the serde_json choice

→ learned about lazy_static and mutexes

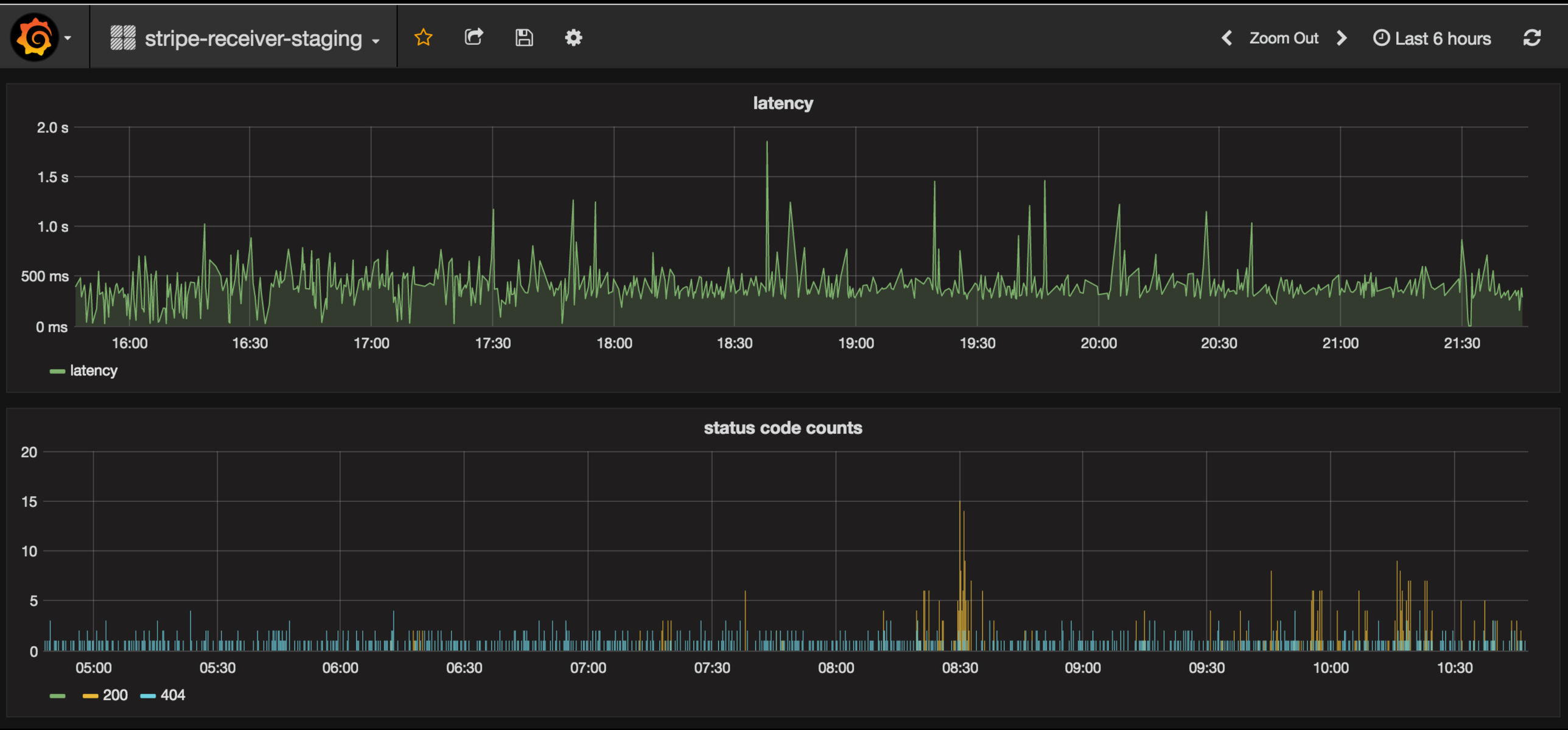
```
let mut point_defs: BTreeMap<&str, Value> = BTreeMap::new();
point_defs.insert("x", serde_json::to_value("global"));

emitter().init(point_defs, "numbat-emitter");
emitter().connect("tcp://localhost:4677");
emitter().emit_name("initialization");
emitter().emit_name_val_tag("response", 23, "status", "200");
```

error handling ha ha ha

doesn't retry, doesn't reconnect

but the happy path works!


stripe-receiver-staging
Zoom Out
Last 6 hours
latency
2.0 s
1.5 s
1.0 s
500 ms
0 ms
16:00
16:30
17:00
17:30
18:00
18:30
19:00
19:30
20:00
20:30
21:00
21:30
latency
status code counts
20
15
10
5
0
05:00
05:30
06:00
06:30
07:00
07:30
08:00
08:30
09:00
09:30
10:00
10:30
200
404

the server is operationalized
now actually proxy something!

```rust
fn proxy_request(request: &Request, request_body: Vec<u8>,
  target: &str, target_host: &str)
  -> Result<hyper::client::Response, hyper::Error>
{
    use hyper::header::Host;
    let client = hyper::Client::new();

    let mut headers = request.headers.clone();
    headers.set(Host {
        hostname: target_host.to_string(),
        port: None
    });

    let proxy_response = try!(
        client.request(request.method.clone(), target)
        .headers(headers).body(&request_body[..]).send());

    Ok(proxy_response)
}
```

current status:

in staging, handling traffic successfully

to-do list

1. finish up the metrics emitter & publish it
2. continue to learn Rust idioms & rewrite
3. fix my proxy code omg
4. build server & build artifact deployer

conclusion: we can write perf critical services in Rust