the web server deployment model breaks at hobby scale

lets say you want to make a web thing that you intend other people to host on their own servers. unfortunately, by thinking this very thought you immediately locked your codebase out of several efficiency tricks other software intended to be hosted privately can make use of, and you have now burdened yourself with reinventing several wheels others have already done better.

let's start out with a baseline, which we'll add to as requirements change:

- something to take care of tls
  - separate from the application so your private keys aren't exposed wider than necessary
  - lets you change how you get your certs without impacting application code
- the app

in pretty much every case, the first bit will likely also be a relatively capable web server that includes reverse proxying as one of many features. one very common feature is to serve static files, which is currently handled by your application.

#static files

it's not a bad idea to offload static file serving to the reverse proxy. it's implementation will likely be way better than whatever comes with the hip new framework you chose, and even if not, static files will no longer clog up the mediocre amount of requests your python or node web server can handle concurrently. you attempt this.

if either one of your app or the reverse proxy is containerized or otherwise compartmentalized, every admin setting up your application now has to poke a hole in one or both of these compartments to let the reverse proxy access the static files distributed with the app.

once your app 

one whose reverse proxy ut reverse proxying and i add back in

realize it exists. you weep as you realize you have contributed to the fair share of inefficiency bogging down the potential of the lower end hardware your target audience has access to.

in "professional" scale? static files are deployed completely separately, likely to an s3 bucket or somesuch and fronted by an Actual Cdn™, so this issue never comes up.

#unauthenticated caching

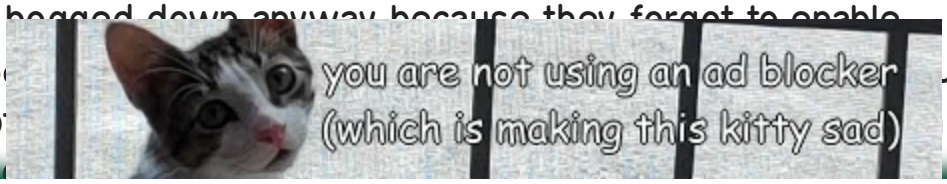
you recognize your application will get a lot of unauthenticated visits, either by humans or bots. unauthenticated visits always get the same response, so there's no real reason to re-compute these responses if nothing changed. you determine that an hour is an adequate amount of time for a "stale" page in your context.

you look fondly at the vinyl cache website, with full knowledge you can't actually require it as you're just one part of the already existing rube goldberg machine of a hobbyist's web server.

dejected, you add the necessary `cache-control` and `vary` headers to your endpoints. you find out about `no-vary-search` and realize it's chrome only. you know the exact set of query parameters that will change the response, but there's nothing to do from where you are. if you *could* depend on vinyl, you could trim out the unused query parameters, so annoying people trying to resurrect a dead joke about link previews don't try to bypass your cache by adding `?qweqweawe` after your links.

you find out your web server library has a caching middleware. you read through the features it supports and realize it doesn't support anything other than `cache-control`'s ttl and perhaps `vary` if you're lucky. you add it in, with nothing but a glimmer of hope in your heart that nobody tries anything funny. you put a configuration switch and documentation in hopes that admins who already have a good cache set up will tweak it for best efficiency. they don't

once your app is released, you get an issue from a caddy user, because they trusted caddy's unbelievably mediocre caching plugin which will gladly corrupt responses if it feels like it. someone else has enabled nginx's cache but their install keeps getting bogged down anyway because they forget to enable `proxy_cache_lo` ut enabled your cache op their own machine, you



you also realize your builtin caching middleware is caching your static files. serving those are relatively trivial and likely already cached by your OSs disk caching facilities, so you're just bloating your memory for nothing. because your middleware only looks at `cache-control`, you can't do anything without preventing browsers from also caching your best-practice immutable static files on the client's end. you weep.

in "professional" scale? they control the entire machine, so they can indeed set up vinyl or something along it's lines and tune it for the exact cache behavior they want. well, ok, they probably outsource it to a cdn which does a mediocre job at it, but they have the option to control caching this granularly.

#authenticated caching

you recognize your application may get quite a lot of authenticated visits as well. maybe you're doing something federated, where you have remote instances authenticating with you. the data they fetch still won't change that often, and certainly won't change *between* different requesters. there's no reason to re-compute those either.

you determine a shorter timeout, say, a minute, is adequate for a "stale" response. this cache is only truly meant to alleviate bursts of requests. because you know you'll have to run your code before the cache anyhow, you're confident you can invalidate the cached data when it changes.

you quickly realize if you want to support external caches, your only options are

- write plugins for each cache middleware
  - or, more realistically, just one supported cache middleware and leave the rest "to the community" (nobody will bother)
- write a reverse proxy that handles your authentication and passes the result into the cache middleware as appropriate, with a header or something you can `vary` on.

you go with neither, as both options complicate the deployment of your software. you are very attentive to the fact that people will not bother with deploying your software if the instructions are more than

- run it
- point your reve

to the point where there's an entire ecosystem of "one-click deployment" middleware and operating systems, which package up everything into a nice point and click interface for newcomers. you realize the advantages of this, letting more people control their own data is a good idea. you support the concept even if you weep at the existing implementations of it.

so you take the only remaining option. you stop supporting external cache middleware, and move the authentication middleware to run earlier than the caching middleware in your application's code. you weep, because the cache middleware in your framework sucks.

in "professional" scale? again, they control the entire machine. they can pick either solution and make it work.

#the best programming language for the job

you have enough interactivity to warrant a single-page-application as your frontend. however, you also want people without javascript enabled to be able to read the bits that don't require interaction. perhaps this interactivity only makes sense for logged-in users. you decide to use a server-side-rendered single-page-app framework.

every single competent option is written in javascript. this makes sense, you don't want to write the same code twice, both for the backend and for the frontend. your application, however, is not written in javascript.

you briefly have a really bad idea, which is to embed nodejs into your application. this technically works, but feels very fragile. the single-threaded nature of nodejs also means you'll want multiple frontend processes to load balance between. additionally, you want your frontend and application to be separate in various resource use metrics, so you can tell where an optimization will be most effective.

you briefly have a second bad idea, to make your application launch the nodejs processes itself. you realize you'll need to reinvent half of systemd, openrc, dinit, or runit, just to make deployment simpler, while annoying every sysadmin out there who would really like to use their existing service manager to manage the frontend.

your only optic ntirely, making it talk publicly

you see that the reverse proxying rules you would need are more complicated than usual. you can't Simply forward `/api` to your application and the rest to the frontend due to various reasons outside your control. an idea forms in your head: make your application act as it's own reverse proxy to the frontend. maybe you could even seed it with some initial data to make page loads faster.

you quickly realize your web framework of choice does not have a way to do that. you'll need to build that functionality in yourself. doing this, you stop to think: wouldn't these requests to the frontend also clog up my application? a request slot doing nothing but waiting for node to render some html, which itself does another request back to your application using another slot, could instead be used to do something useful.

defeated, you cave in and split the frontend entirely. you document what needs to be reverse proxied and give examples for caddy and nginx, repeatedly cursing at nginx for not having an `if` that's useful. whenever someone asks you for help with another reverse proxy, you can't do anything but close the issue with "patches welcome".

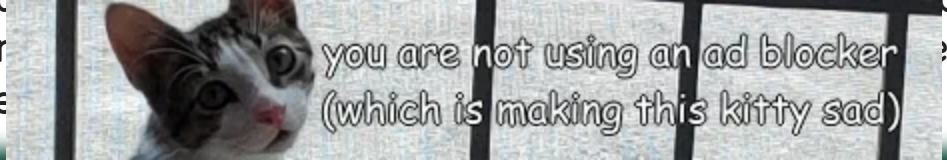
in "professional" scale? the frontend was split out from the get-go because deployment was never a concern.

#the web server is not the only dependency!

you get an issue from a japanese person. they love your app, but searching for text takes quite a long time and uses quite a lot of system resources. that's strange. you use postgres with tsvector or a gin index. it's blazing fast on your end. you inquire further: they're searching in japanese.

that's a problem. postgres's builtin full-text-search functions break outside latin script. you look around for a solution. you're not gonna use elasticsearch or meilisearch because you care about not just the resource usage of your application alone, but also when colocated with other applications. sonic looks interesting, but it's another index to synchronize and another service admins will have to maintain. this doesn't sit right with you.

you come across pgroonga. it claims to solve the problem while just being a postgres extensior es in your database inste



your application now depends on a postgres extension. most hobby setups share one postgres for every app to amortize the overhead of having a postgres instance. you are now asking all these admins to meddle with the shared postgres install Just for your application, potentially requiring compiling the extension from source. especially given postgres updates are already difficult to start with, you realize the easy deployment you set as a goal to yourself is long gone. you weep.

in "professional" scale? there is one postgres and it's only job is to serve the application. extensions are No Big Deal

#the predictable end

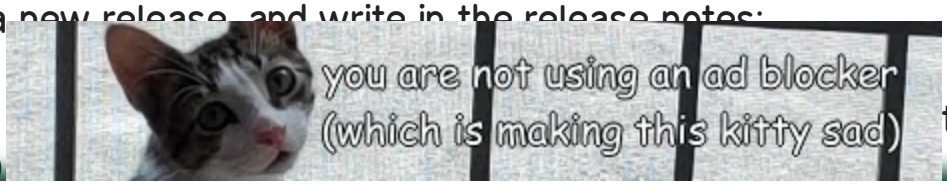
you take one more look at the mess you just built, the novel you've written just to document how to install it, and all the support requests you had to answer for every combination of external dependencies. you remember your initial goal of making something easy to deploy, so people will actually use it. you see several vibe-coded installation scripts coming out that abstract all this away and do it worse than you would if you gave up this constraint. you realize what must be done.

you move the deployment documentation into the developer docs folder, where it will be left to rot. you produce a dockerfile which bundles all of:

- runit (to start all these)
- your favorite reverse proxy
- the cache, configured perfectly for your application
  - you rewrite your authentication middleware to work as a plugin for the cache, for maximum efficiency
- postgres, with all the required extensions, and a script to tune it on first boot because you kept getting issues about slow queries from people who had no idea you had to tune postgres
- the frontend
- the backend

so all the admin has to do is to point their existing reverse proxy at your container.

you produce a new release, and write in the release notes:



> **IMPORTANT:** ted. Please follow the D

the user makes an http request. the pre-existing reverse proxy parses the request and writes a new http request to the container. the second reverse proxy in the container parses this new request, and writes a new http request the cache. the cache parses this new request, doesn't find the response in it's cache, and writes a new http request to one of the backend or the frontend. it then parses this new request. if this is the frontend, more internal requests to the backend may need to be triggered, but at least they don't need to go above the cache. the response is written. the cache parses the response, saves it if necessary, and writes a new response. the containerized reverse proxy parses this response, does the nothing it's configured to (not) do, and writes a brand new response. the pre-existing reverse proxy parses this response, adds a header or two maybe, and writes the final response to be shipped to your user.

is all this really necessary?

the user, it later turns out, was a scraper operating over a residental proxy. it fetched the very first revision of a page that was last relevant 2 years ago. it will now proceed to fetch the second revision, blindly following all the links on the history tab of the page, from a different ip address.