Reconstruyendo mi infraestructura personal con Alpine Linux y Docker

Publicado en Jue, 28 de febrero de 2019

Durante más de una década, he ejecutado uno o más servidores para alojar una serie de sitios web personales y aplicaciones web. Recientemente decidí que era hora de reconstruir los servidores para abordar algunos problemas y hacer mejoras. La última vez que hice esto fue en 2016 cuando cambié los servidores de Ubuntu a FreeBSD . Los servidores salientes se administraron con Ansible . Después de ser un escéptico de Docker durante mucho tiempo, finalmente lo descubrí recientemente y decidí reconstruir en Docker . Esta publicación tiene como objetivo describir algunas de las elecciones realizadas y por qué las hice.

Antes de comenzar, me gustaría tomar un momento para reconocer que esta infraestructura está construida según mis valores de una manera que me funciona. Puedes tomar diferentes decisiones y eso está bien. Espero que esta publicación sea interesante pero no prescriptiva.

Antes de la reconstrucción, así era mi infraestructura:

- Servidor FreeBSD 11 en hosting DigitalOcean (Nueva York):
    - PostgreSQL 9
    - nginx
    - Barniz
    - 2 aplicaciones de rieles
    - Sitios estáticos
- Servidor Debian 9 en hosting DigitalOcean (Nueva York):
    - Wizards Mattermost instancia
- Servidor FreeBSD 12 en el alojamiento Vultr (Sydney):
    - instancia de rust.melbourne Mattermost
    - PostgreSQL 11

Notará 3 servidores en 2 países y 2 proveedores de alojamiento. Además, el servidor de Rust Melbourne no fue administrado por Ansible como lo fueron los otros dos.

Tenía varios objetivos en mente con la reconstrucción:

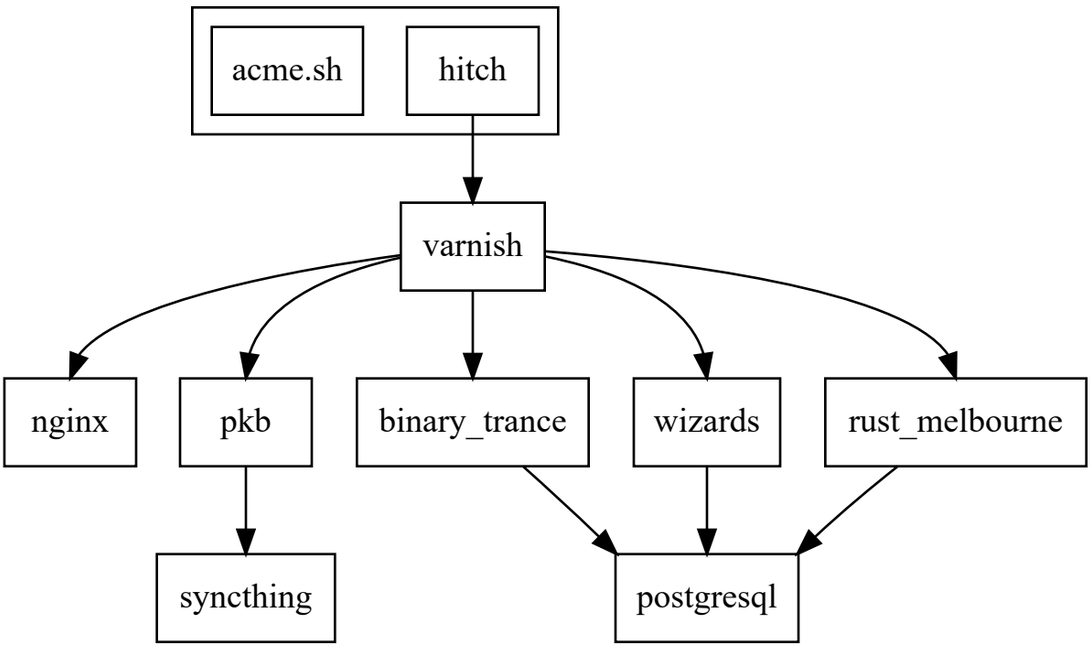
- Mover todo a Australia (donde vivo)
- Consolidar en un servidor
- https habilita todos los sitios web

Configuré mi infraestructura original en los EE. UU. Porque era más barata en ese momento y la mayoría del tráfico a los sitios web que alojo proviene de los EE. UU. La instancia de Wizards Mattermost se agregó más tarde. Es para un grupo de amigos que están en Australia. Estar en los EE. UU. Lo hizo bastante lento a veces, especialmente al compartir y ver imágenes.

Otro inconveniente de administrar servidores en los EE. UU. Desde AU es que hace que el tiempo de ciclo de Ansible sea "hacer un cambio, ejecutarlo, arreglarlo, repetir", terriblemente lento. Había estado en mi lista de tareas durante mucho tiempo trasladar a Wizards a Australia, pero seguí posponiéndolo porque no quería tratar con Ansible.

Si bien tener un solo servidor que haga todo no sería la arquitectura recomendada para los sistemas empresariales, para el alojamiento personal donde la pequeña posibilidad de tiempo de inactividad no dará como resultado la pérdida de ingresos, la simplicidad ganó, al menos por ahora.

Esto es lo que terminé construyendo. Cada caja es un contenedor Docker que se ejecuta en la máquina host:



- `pkb`es https://linkedlist.org
- `binary_trance`es https://binarytrance.com
- `wizards`y `rust_melbourne`son Mattermost casos
- El resto son software del mismo nombre.

No siempre he estado a favor de Docker, pero creo que ha pasado suficiente tiempo para demostrar que probablemente haya llegado para quedarse. También hay algunos beneficios realmente buenos para los servicios administrados de Docker. Por ejemplo, compilar localmente y luego enviar la imagen a producción, y aislarla del sistema host (en el sentido de que puede destruir el contenedor y reconstruirlo si es necesario).

Elegir un sistema operativo host

Mudarse a Docker desafortunadamente descartó FreeBSD como el sistema host. Hay un puerto Docker muy antiguo para FreeBSD, pero mis intentos anteriores de usarlo mostraron que no estaba en un estado lo suficientemente bueno como para usarlo como host. Eso significaba que necesitaba encontrar una distribución de Linux adecuada para actuar como host Docker.

Viniendo de FreeBSD, soy fanático del modelo de base estable + paquetes actualizados. Para mí, esto descartó los sistemas basados en Debian (estables), que a menudo encuentro que tienen paquetes desactualizados o faltantes, especialmente en las últimas etapas del ciclo de lanzamiento. Investigué un poco para ver si había alguna distribución que usara un modelo de estilo BSD. La mayoría que encontré fueron operaciones abandonadas o de una sola persona.

Entonces recordé que, como parte de su trabajo en Sourcehut , Drew DeVault estaba migrando cosas a Alpine Linux . Había jugado con Alpine en el pasado (antes de que se hiciera famoso en el mundo de Docker), y considero que Drew utilizó alguna evidencia a su favor.

Alpine se describe a sí mismo de la siguiente manera:

> Alpine Linux es una distribución de Linux independiente, no comercial y de propósito general diseñada para usuarios avanzados que aprecian la seguridad, la simplicidad y la eficiencia de los recursos.

¡Ahora esa es una declaración de valor que puedo respaldar! Otras cosas que me gustan de Alpine Linux:

- Es pequeño, solo incluye lo esencial:
    - Evita la hinchazón mediante el uso de musl-libc (que tiene licencia MIT) y busybox userland .
    - Tiene una instalación ISO de 37Mb destinada a instalaciones de servidores virtualizados.
- Es probable que sea (y terminó siendo) la base de mis imágenes de Docker.
- Habilita una serie de características de seguridad de forma predeterminada.
- Las versiones se realizan cada ~ 6 meses y son compatibles por 2 años.

Cada versión también tiene paquetes binarios disponibles en un canal estable que recibe correcciones de errores y actualizaciones de seguridad durante la vida útil de la versión, así como un canal innovador que siempre está actualizado.

Tenga en cuenta que Alpine Linux no usa systemd , usa OpenRC . Esto no influyó en mi decisión en absoluto. `systemd`me ha funcionado bien en mis sistemas Arch Linux. Puede que no sea perfecto, pero hace muchas cosas bien. Benno Rice hizo una gran charla en linux.conf.au 2019, titulada, The Tragedy of systemd , que lo hace interesante para ver este tema.

Imágenes de construcción

Entonces, con el sistema operativo host seleccionado, me puse a crear imágenes de Docker para cada uno de los servicios que necesitaba ejecutar. Hay muchas imágenes Docker preconstruidas para software como nginx y PostgreSQL disponibles en Docker Hub . A menudo también tienen una `alpine`variante que construye la imagen a partir de una imagen base alpina. Decidí desde el principio que estos no eran realmente para mí:

- Muchos de ellos compilan el paquete desde el origen en lugar de simplemente instalar el paquete Alpine.
- La construcción de Docker fue más complicada de lo que necesitaba, ya que intentaba ser una imagen genérica que cualquiera pudiera extraer y usar.
- No era un gran fanático de extraer imágenes aleatorias de Docker de Internet, incluso si eran imágenes oficiales.

~~Al final solo necesito confiar en una imagen de Docker Hub : la imagen alpina de 5Mb . Todas mis imágenes están construidas sobre esta imagen.~~

Actualización 2 de marzo de 2019: ya no dependo de ninguna imagen de Docker Hub. Después de la versión Alpine Linux 3.9.1 , noté que las imágenes oficiales de Docker no se habían actualizado, así que construí la mía. Resulta que es bastante simple. Descargue el tarball miniroot del sitio web de Alpine y luego agréguelo a una imagen de Docker:

```
FROM scratch

ENV ALPINE_ARCH x86_64
ENV ALPINE_VERSION 3.9.1

ADD alpine-minirootfs-${ALPINE_VERSION}-${ALPINE_ARCH}.tar.gz /
CMD ["/bin/sh"]
```

Un aspecto de Docker que realmente no me gusta es que dentro del contenedor eres root por defecto. Cuando construí mis imágenes, hice un punto de hacer que los procesos de punto de entrada se ejecutaran como un usuario sin privilegios o configure el servicio desplegable para un usuario normal después de comenzar.

La mayoría de los servicios eran bastante fáciles de Dockerise. Por ejemplo, aquí está mi nginx `Dockerfile`:

```
FROM alpine:3.9

RUN apk update && apk add --no-cache nginx

COPY nginx.conf /etc/nginx/nginx.conf

RUN mkdir -p /usr/share/www/ /run/nginx/ && \
  rm /etc/nginx/conf.d/default.conf

EXPOSE 80

STOPSIGNAL SIGTERM

ENTRYPOINT ["/usr/sbin/nginx", "-g", "daemon off;"]
```

No me esforcé por hacer que las imágenes fueran especialmente genéricas. Solo necesitan trabajar para mí. Sin embargo, hice un punto para no hornear ninguna credencial en las imágenes y en su lugar utilicé variables de entorno para cosas como esas.

Encriptemos

I've been avoiding Let's Encrypt up until now. Partly because the short expiry of the certificates seems easy to mishandle. Partly because of certbot, the recommended client. By default `certbot` is interactive, prompting for answers when you run it the first time, it wants to be installed alongside the webserver so it can manipulate the configuration, it's over 30,000 lines of Python (excluding tests, and dependencies), the documentation suggests running magical `certbot-auto` scripts to install it… Too big and too magical for my liking.

Despite my reservations I wanted to enable https on all my sites and I wanted to avoid paying for certificates. This meant I had to make Let's Encrypt work for me. I did some research and finally settled on acme.sh. It's written in POSIX shell and uses `curl` and `openssl` to do its bidding.

To avoid the need for `acme.sh` to manipulate the webserver config I opted to use the DNS validation method (`certbot` can do this too). This requires a DNS provider that has an API so the client can dynamically manipulate the records. I looked through the large list of supported providers and settled on LuaDNS.

LuaDNS has a nice git based workflow where you define the DNS zones with small Lua scripts and the records are published when you push to the repo. They also have the requisite API for `acme.sh`. You can see my DNS repo at: https://github.com/wezm/dns

Getting the acme.sh + hitch combo to play nice proved to be bit of a challenge. `acme.sh` needs to periodically renew certificates from Let's Encrypt, these then need to be formatted for `hitch` and `hitch` told about them. In the end I built the `hitch` image off my `acme.sh` image. This goes against the Docker ethos of one service per container but `acme.sh` doesn't run a daemon, it's periodically invoked by cron so this seemed reasonable.

Docker and cron is also a challenge. I ended up solving that with a simple solution: use the host cron to `docker exec acme.sh` in the `hitch` container. Perhaps not "pure" Docker but a lot simpler than some of the options I saw.

Hosting

I've been a happy DigitalOcean customer for 5 years but they don't have a data centre in Australia. Vultr, which have a similar offering – low cost, high performance servers and a well-designed admin interface – do have a Sydney data centre. Other obvious options include AWS and GCP. I wanted to avoid these where possible as their server offerings are more expensive, and their platforms have a tendency to lock you in with platform specific features. Also in the case of Google, they are a massive surveillance capitalist that I don't trust at all. So Vultr were my host of choice for the new server.

Having said that, the thing with building your own images is that you need to make them available to the Docker host somehow. For this I used an Amazon Elastic Container Registry. It's much cheaper than Docker Hub for private images and is just a standard container registry so I'm not locked in.

Orchestration

Once all the services were Dockerised, there needed to be a way to run the containers, and make them aware of each other. A popular option for this is Kubernetes and for a larger, multi-server deployment it might be the right choice. For my single server operation I opted for Docker Compose, which is, "a tool for defining and running multi-container Docker applications". With Compose you specify all the services in a YAML file and it takes care of running them all together.

My Docker Compose file looks like this:

```
version: '3'
services:
  hitch:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/hitch
    command: ["--config", "/etc/hitch/hitch.conf", "-b", "[varnish]:6086"]
    volumes:
      - ./hitch/hitch.conf:/etc/hitch/hitch.conf:ro
      - ./private/hitch/dhparams.pem:/etc/hitch/dhparams.pem:ro
      - certs:/etc/hitch/cert.d:rw
      - acme:/etc/acme.sh:rw
    ports:
      - "443:443"
    env_file:
      - private/hitch/development.env
    depends_on:
      - varnish
    restart: unless-stopped
  varnish:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/varnish
    command: ["-F", "-a", ":80", "-a", ":6086,PROXY", "-p", "feature=+http2", "-f", "/et
    volumes:
      - ./varnish/default.vcl:/etc/varnish/default.vcl:ro
    ports:
      - "80:80"
    depends_on:
      - nginx
      - pkb
      - binary_trance
      - wizards
      - rust_melbourne
    restart: unless-stopped
  nginx:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/nginx
    volumes:
      - ./nginx/conf.d:/etc/nginx/conf.d:ro
      - ./volumes/www:/usr/share/www:ro
    restart: unless-stopped
  pkb:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/pkb
    volumes:
      - pages:/home/pkb/pages:ro
    env_file:
      - private/pkb/development.env
    depends_on:
      - syncthing
    restart: unless-stopped
  binary_trance:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/binary_trance
    env_file:
      - private/binary_trance/development.env
    depends_on:
      - db
    restart: unless-stopped
  wizards:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/mattermost
    volumes:
      - ./private/wizards/config:/mattermost/config:rw
      - ./volumes/wizards/data:/mattermost/data:rw
      - ./volumes/wizards/logs:/mattermost/logs:rw
      - ./volumes/wizards/plugins:/mattermost/plugins:rw
      - ./volumes/wizards/client-plugins:/mattermost/client/plugins:rw
      - /etc/localtime:/etc/localtime:ro
    depends_on:
```

```
      - db
    restart: unless-stopped
  rust_melbourne:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/mattermost
    volumes:
      - ./private/rust_melbourne/config:/mattermost/config:rw
      - ./volumes/rust_melbourne/data:/mattermost/data:rw
      - ./volumes/rust_melbourne/logs:/mattermost/logs:rw
      - ./volumes/rust_melbourne/plugins:/mattermost/plugins:rw
      - ./volumes/rust_melbourne/client-plugins:/mattermost/client/plugins:rw
      - /etc/localtime:/etc/localtime:ro
    depends_on:
      - db
    restart: unless-stopped
  db:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/postgresql
    volumes:
      - postgresql:/var/lib/postgresql/data
    ports:
      - "127.0.0.1:5432:5432"
    env_file:
      - private/postgresql/development.env
    restart: unless-stopped
  syncthing:
    image: 791569612186.dkr.ecr.ap-southeast-2.amazonaws.com/syncthing
    volumes:
      - syncthing:/var/lib/syncthing:rw
      - pages:/var/lib/syncthing/Sync:rw
    ports:
      - "127.0.0.1:8384:8384"
      - "22000:22000"
      - "21027:21027/udp"
    restart: unless-stopped
volumes:
  postgresql:
  certs:
  acme:
  pages:
  syncthing:
```

Bringing all the services up is one command:

```
docker-compose -f docker-compose.yml -f production.yml up -d
```

The best bit is I can develop and test it all in isolation locally. Then when it's working, push to ECR and then run `docker-compose` on the server to bring in the changes. This is a huge improvement over my previous Ansible workflow and should make adding or removing new services in the future fairly painless.

Closing Thoughts

The new server has been running issue free so far. All sites are now redirecting to their https variants with `Strict-Transport-Security` headers set and get an A grade on the SSL Labs test. The Wizards Mattermost is *much* faster now that it's in Australia too.

There is one drawback to this move though: my sites are now slower for a lot of visitors. https adds some initial negotiation overhead and if you're reading this from outside Australia there's probably a bunch more latency than before.

I did some testing with WebPageTest to get a feel for the impact of this. My sites are already quite compact. Firefox tells me this page and all resources is 171KB / 54KB transferred. So there's not a lot of slimming to be done there. One thing I did notice was the TLS negotiation was happening for each of the parallel connections the browser opened to load the site.

Some research suggested HTTP/2 might help as it multiplexes requests on a single connection and only performs the TLS negotiation once. So I decided to live on the edge a little and enable Varnish's experimental HTTP/2 support. Retrieving the site over HTTP/2 did in fact reduce the TLS negotiations to one.

Thanks for reading, I hope the bits didn't take too long to get from Australia to wherever you are. Happy computing!

✦

Previous Post: My Rust Powered linux.conf.au e-Paper Badge
Next Post: A Coding Retreat and Getting Embedded Rust Running on a SensorTag

Stay in touch!

Follow me on Twitter or Mastodon, subscribe to the feed, or send me an email.