Melton API

Introduction

The Melton Foundation wanted to find a way to keep their fellows more engaged, a group of dashing young fellows comes by and decides to save the day with a new app! Said app needs a frontend-backend component, like most apps do. But hosting a backend costs money, time, patience, love and care - So I (Pari) come to help out.

The Melton API is largely written in python and uses GitHub as a versioning system. They need a way to host it such that it is publically available through a public DNS address. Ideally the hosting also has some alert management, logging,etc.

The hosting destination was quite clear for me: Indenwolken - My VPS running some of my other tools and services. But getting the code in a usable form onto the server wasn't straightforward.

Solutions that were needed:

- CI system + Building DockerFiles + Push image to Docker repo - To run tests, build the image - Potentially could use Jenkins or GithubActions

- CD/Orchestration system - To connect to to the server and start the docker image - Potentially could use Rancher, Nomad or Ansible

- Logging system - To collect and store logs - Could use the default docker logging driver or Loki

- Alerting - To figure out when the service is down and alert me - Notification via Telegram/PagerDuty

I will go through each of the topics page by page, and summarise it in the end.

CI + Build + Docker repo

The world of CI is large, unfortunately I was just too used to Gitlab CI (The best IMHO). Since the code was hosted on GitHub I was out of luck with using Gitlab CI. I took it as a chance to explore other options- Jenkins and Github Actions in particular

Jenkins needs to be run somewhere, and running it on the same server as the deployment might mean both go down at the same time. Also, its just more work and I didn't feel like doing it/ dealing with certs, bla bla.

Github Actions wins by default, the best kind of victory. Github Actions supports building docker images through their `build-push` action. But before that, we need to talk about secret management.

Secrets are a tough nut to crack when it comes to CI/CD - How do you reliably get an API key into the build system without just committing it in code. The solution I went for is still not the most ideal, I can talk about the negative of it in the end. But I went with using the env-method approach. Environment variables can easily be coded into the program, the code simply expects this environment variable to exist. These variables are however injected into the Docker image during the build stage. And they are injected by Github Actions - which reads the value from the Github Secrets feature.

TL;DR:

image-1597600383422.png

Caveats:

In this whole step, the API key is never directly exposed, it is simply referenced. The Github Secret is setup such that a secret can only be aletered and never read once it is entered. Implying one cannot steal an API key, but one can destroy it. This makes the entire CI process relatively safe. Code-scanners wouldn't find any API key in the code to exploit, any attack to find the key needs to be extremely targeted.

The keen eyed amongst you might have noticed a flaw, `Environment Variable`! This implies any person that has access to the docker image, need only run it and then they have complete access to the API key. The mitigration strategy for this is to set the Docker image access as private, enabling a need-only access to it. Narrowing the attack surface. Since the Melton API isn't a critical piece of software, such a narrow attack surface is deemed acceptable.

Since we have the Github Action that can build and push our image, we only need to find an endpoint to push it to. Gitlab private container repositories would be amazing (Ahh Gitlab <3). But since we don't have that, we will defer to just using a private repository on DockerHub.

To summarize,

- API keys are "inserted" as Github Secrets, made accessible to the program as Environment Variables
- Github Actions build the Dockerfile using the default action
- The built docker image is pushed to a private DockerHub repository

Continuous Delivery/Orchestration Sytem

Once we have the image built and ready to be run, we need to find a way to get it to the server. There were three great options: Rancher, Nomad/Kubernetes and Ansible.

I will divide all 3 into their own segments and talk about the pro-cons of each option. Spoiler alert, I went with Ansible.

Rancher:

Rancher (v1.6) is a simple orchestration system. You can define services you want to run with a YAML file (similar to Docker-compose) and submit it to your rancher endpoint, rancher will then start said service in one of its registered hosts and automatically forward traffic, load balance, etc for you.

> **Pro:** Automatic load balancing, easy to define, GUI based management
>
> **Cons:** Hard to run on a 1 host system, We would need to define and setup an ingress, High overhead for a small payoff

Nomad/Kubernetes:

Before anyone gets offended, I know the two aren't the same. But they are both "multi-node" setup systems.

Nomad: Closer to Rancher than it is to kubernetes, You can define a single-host and mention what services you want it to run, the host then schedules itself and restarts connections if needed.

Kubernetes: Typically made for multi-node systems, super efficient and highly scalable system. Its totally the bees-knees as the young people say.

> **Pro:** Very easily scalable, high availability is pretty much guaranteed, I'm used to using kubernetes so quick turn-around time

> **Cons:** Suuuuper overkill for what we need, single node clusters tend to be "Dev-mode" and not production ready since the systems are built expecting components to fail

Ansible:

Ansible is an automation tool that can be scripted to manage anything. It can be hence setup to perform Continuous Delivery but cannot perform orchestration itself. Its lightweight. Its run once and forget. We let docker inherently handle orchestration, since all the orchestration we need is a restart when the service dies.

Ansible can be run via a Github Action using the shell-run mode.

> **Pro:** Run and forget, performs CD without too much overhead

> **Cons:** Since its run once and forget, it cannot retry when things don't go as expected. No orchestration possible with it, just delivery

Since its the best fit for what we want, I decided to go ahead with just using Ansible and figuring out the cons through other methods.

> Ansible runs on a remote server via SSH

The ansible part of the Github Action is divided into two parts: Add login method for the server, Run ansible playbook

> **Adding login method for server:** Since I do understand basic OPSec, I have disabled SSH login for the server through a password/username. This reduces chances that someone bruteforces the credentials. Login is only possible through whitelisted public keys. I generated a keypair to be used by Ansible and added the key as a Github Secret.
>
> Now we need to import and configure ansible to use this key, this was easiest with just using a Github Action job that runs on an Ubuntu image, giving me a bash shell to just perform my commands.
>
> The job first creates a `.ssh` folder and adds the previously mentioned key there. Access is modified to allow the action agent to access it, the `eval "$(ssh-agent -s)"` ensures the agent is running. Then `ssh-add` will add the key to the keychain for SSH access.
>
> **Run ansible playbook:** Once the dependency has been installed on the ubuntu image (In Github Action Job). We need to disable host key checking (it was just more work to implement :P). Since the server might actually be down or the config borked, We run a ping test first.
>
> Once the ping test is successful, `ansible-playbook` can be run. The hosts that this is run on is defined in the `hosts`, if we decide to have multiple servers in the future, simply define them in the `hosts` file and the deployment will be performed on all hosts.

Additionally, the ansible-playbook needs access to the private docker repository. The access credentials are defined in the Github Actions and are passed on as an env variable while running the playbook. The docker-tag defines the image to be deployed, this is fetched and passed by using the `${GITHUB_REF#refs/*/}`.

The Ansible-Playbook:

> The variables are defined in the beginning of the file, if the port of the image changes, or the name of the image changes, please change it in the ansible-playbook variable definition.

The ansible playbook is divided into two parts: Install dependencies and then Login/Create the Docker container.

Install dependencies: Since the host can change at any time, its best practice to install explicit dependencies needed by the image. This is defined in the ansible install tasks.

Login/Create Docker container: Since the docker image is in a private repository, the ansible-playbook `docker_login` is first used to login to the docker hub repo. Once the login is successful, we should have access to the image. The `docker_container` can now be used to define the image to be run on the remote server and the state the image needs to be at at the end of the task. Since there are quite a few things happening here, I will post the code block and describe it below.

```
- name: Create default containers
    docker_container:
      name: "{{ default_container_name }}{{ item }}"
      image: "{{ default_container_image }}:{{ container_tag }}"
      network_mode: host
      ports:
        - "{{ src_port }}:{{ dest_port }}"
      volumes:
        - /root/melton-app/staticfiles:/code/api/static/
      state: started
      log_driver: loki
      log_options:
        loki-url: http://localhost:3100/api/prom/push
        loki-retries: 5
        loki-batch-size: 400
        loki-external-labels: tag=melton
    with_sequence: count={{ create_containers }}
```

The image name and container tag is defined in the variable at the start of the playbook. The `network_mode` is set to `host` to avoid having to expose ports on top, I was just lazy to lock this

down. It also enables easy access to the log-driver. The ports are defined in the variables, and are mapped here - The host port is mapped to the port on the docker container. Since the image contains static images stored on the server, we map the volume to enable persistence over multiple restarts of the docker image. The state of the image is set to started, to ensure its run before the task completion. The log driver is defined in the next step chapter, but we send all logs to loki to manage them. The number of containers to be run on the host is defined by the count, this can be increased if we need loadbalancing.

At the end of this playbook, the image will be run on the remote server and will have the state defined. It would be ready to receive traffic on the `src_port`, we will later configure NGINX to deliver traffic to this port.

Logging System

Logs are somehow irritating to comb through, especially when the logs are scattered around. A good logging system would defintely help alleviate some of this hassle. The standard recommendation seemed to be the ELK stack, Loki was the new kid of the block. So the two logging tools I tried out were: ELK stack, Loki

ELK Stack:

> The ELK stack stands for Elastic Search, Logstash and Kibana - One for metric collection, metric ingestion and then distribution/visualization. I setup the system with docker containers and it took a lot of work in order to even get the system to start properly. I think the root cause is that the ELK stack is not meant to be run on a single host. It took up significantly more resources than the Melton API it was monitoring, doesn't really make sense. It was also quite hard to configure for basic log collection. Since it took more resources than it was worth, I decided to keep looking.

Loki:

> As with other stuff the Grafana guys create, Loki is pure gold. `Loki is a horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus. It is designed to be very cost effective and easy to operate.` Sounds like someone was reading my requirement list :D
>
> So I decided to set up Loki+Promtail(metric collector) in a docker container, and I added `/var/log` as a volume mount. This volume mount means Loki would already get access to system level logs. The config(defined in `loki-promtail-config.yaml`) then scrapes the defined logs and ingests it in a format that is query-able.
>
> Loki is also set up to be able to be used as a log driver directly by docker, this means all docker logs (when specified) are directly labelled and sent over to Loki. Loki is also horizontally scalable, so multiple hosts can forward their logs to the Loki push endpoint.
>
> In our setup, Loki is limited to localhost, listening on port 3100. One can add Loki as the logging driver by using the below code in their docker-compose file:

```
logging:
    driver: loki
    options:
      loki-url: "http://localhost:3100/api/prom/push"
      loki-retries: "5"
      loki-batch-size: "400"
```

I should eventually set up HTTPS support too :P

Now that we have logs ingested by Loki (its quite efficient in resource usage too), then we just have to add it as a datasource on Grafana and we instantly have access to the logs. We can search logs by tags and labels, and even create a custom dashboard that constantly displays the log rate and log themselves.

The dashboard looks like this:

The log frequency is plotted in the graph on top, the App logs are in the second box and the last one defines the requests received by NGINX. The request received may help figure out the source of a crash.

Alerting System

Uptime is important for a backend, we need to be notified when shit hits the fan. The first step of figuring out the service isn't working is to check its status regularly.

However, placing the alerting system on the same machine as the service is a bit unwise. If the server crashes, it takes you alerting system with it. The only way to efficiently achieve this is to implement distributed alerting. We're cheap, so we don't want to pay for someone like PagerDuty/Datadog/etc, enter DIY solutions with my Raspberry Pi at home!

The problems we need to solve:

- Service to send HTTP(S) requests to an endpoint and interpret the result

- Serivice to store that information in a time series database - makes it easier to query

- Service to read time series database and fire an alert when rules are met

- Service to detect the alert and send it to a phone/email/human shock collar

I will provide a TL;DR since its quite a lengthy page and most people will loose interest.

Blackbox_Exporter sends GET requests to the Melton API and interprets the result. It stores the result in Prometheus time series database. Alertmanager reads the data and detects when an alert needs to be fired. Telegram takes that alert and sends it to a Telegram group.

image-1597596457463.png

Proceed furthur if you want a more detailed explanation, i promise i tried to make it funny.

Service to send HTTP(S) requests to an endpoint and interpret the result:

> Blackbox_exporter from prometheus is a great service that does exactly what we need, and its part of the prometheus ecosystem that I explain later on. The config simply specifies the module to use, method, the timeout and if the exporter needs to perform HTTP/HTTPS.

```
modules:
  http_2xx:
    prober: http
```

```
    timeout: 10s
    http:
      valid_status_codes: []
      method: GET
```

A perceptive individual might notice a lack of specification with regard to which URLs to test, I say this individual should chill a bit. I'm getting to it. Snitches get stitches.

Serivice to store that information in a time series database:

Prometheus is supposed to have given humanity the gift of fire, well, the software gives us the gift of monitoring - Clearly the better gift. Prometheus takes metrics and organizes the info into a time series database. It then exposes a port that is ready to be scraped by the monitoring system.

The prometheus service is the glue that bring all the other services together, its the central hub that organizes everything. Prometheus performs its tasks based on the `scrape_configs`. All the jobs described here are the metrics that Prometheus scrapes and organizes into a time series database.

Prometheus also supports alerting, One needs to just specify the endpoint to the alertmanger. We will look into that in the next step.

One can specify the scrape interval - this defines the minimum resolution of the metrics being scraped.

If one looks at the `job_name: 'blackbox'` then you see the URLs. Essentially prometheus specifies that it uses the blackbox_exporter module and run against those URLs and saves the information in prometheus with the labels.

prometheus.yml:

```
global:
  scrape_interval:     30s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 30s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
      monitor: 'example'

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
```

```
  # - "first_rules.yml"
  # - "second_rules.yml"
  - alert.rules.yml

alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - localhost:9093

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 30s
    scrape_timeout: 10s

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ['localhost:9090']

  - job_name: node
    # If prometheus-node-exporter is installed, grab stats about the local
    # machine by default.
    static_configs:
      - targets: ['localhost:9100']

  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module: [http_2xx]
    static_configs:
      - targets:
        - http://httpbin.org/get
        - https://melton.indenwolken.tech/api/
```

```
      - https://hathibox.indenwolken.tech/status.php
      - https://grafana.indenwolken.tech/api/health
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: localhost:9115
```

Service to read time series database and fire an alert when rules are met:

Alertmanager is also part of the prometheus ecosystem, and as defined in the name, it manages alerts. The alerts themselves are defined in the `alert_rules.yml` file. The only alert we care about at the moment is when the endpoint is unreachable for a defined duration of time. That alert looks like this:

```
groups:
- name: alert.rules
  rules:
  - alert: EndpointDown
    expr: probe_success == 0
    for: 300s
    labels:
      severity: "critical"
    annotations:
      summary: "Endpoint {{ $labels.instance }} down"
```

The alertmanager simply fires the alert and nothing else. When the issue is resolved, then the alert no longer fires. Its the job of other services or integrations to perform some action on these alerts.

Service to detect the alert and send it to a phone/email/human shock collar:

Since whatsapp doesn't have a pluggable API, I had to find an integration that works with telegram or signal. I decided to go with telegram since I could find an easily pluggable telegram bot that does this job.

The telegram bot takes a token, and uses that token to post the alert from alertmanager in a specified telegram group. I found it on github, no code audit or anything, but hey - it works.

`https://github.com/inCaller/prometheus_bot`

Traffic Ingress/NGINX

NEEDS UPDATE: Traffic now flows to HAProxy first and then to NGINX.

The Melton Backend would communicate with the front-ends with HTTP/HTTPS protocols. These protocols are reserved to use port 80 and 443 respectively. Since the server might run multiple services that need to be accessible via those ports, we would need an ingress to manage that port and pass on the traffic based on the URL.

The current choice I went with was NGINX, NGINX is a high performance load balancer, SSL termination and Ingress service. It receives a request at its defined ports (80/443) and looks up its own config and decides where to forward the traffic to.

In our scenario, The Melton API listens on port `8080` and has no built in SSL termination - i.e it uses unencrypted communication. Since this is generally a horrible idea, we will terminate SSL with NGINX and pass traffic from NGINX to the Melton API in an unencrypted manner - This isn't too big of a security risk since the unencrypted part happens only inside the confines of the server.

The Melton API also has some static files stored that need to be accessed, these are placed under the `/static/` endpoint and mirror the filestructure on the server. A request with this suffix should respond with the corresponding file on the server.

The config block of the NGINX service and the description are as below:

```
server {
        listen 443; □□□□□□□□□□# Listens on port 443 - HTTPS
        server_name meltonapp.com;□□□□□□□ □# The URL to check for
        ssl on;□□□□□□□□□□□□# Defines if NGINX should use SSL
        ssl_certificate /------/fullchain.pem;□□□□□# Defines the location of the SSL certificate
        ssl_certificate_key /------/meltonapp.com/privkey.pem;□# Defines the location of the certificate key
        ssl_session_cache shared:SSL:10m;□□□□□□# Cache location for SSL

        location /static/ {□□□□□□□□□# If the /static/ endpoint is called, the data from a specific location is served
            root /usr/local/share/staticfiles;□□□□□# Location of data to server
        }

        location / {                                    # In all other scenario's it passes on the traffic to the below location
                proxy_pass http://localhost:8000;□□□    # The port and location the Melton API is accessible from
```

```
            proxy_set_header Host $host;□□□□    # Passing on the headers
            proxy_redirect http:// https://; □□□    # Defines if a redirect is needed
        }
}

server {□□□□□□□□□□□□□□ □# A new server block to redirect HTTP traffic
    listen 80;□□□□□□□□□□□□□# Listens on port 80 - HTTP
    return 301 https://$host$request_uri;□□□□□□# Respond with a 301 (Redirect) to the HTTPS endpoint
}
```

To harden the server a bit, ufw is used. UFW is a built in firewall that can be used to block external port access to the server. This can be used to force all traffic to pass through the ingress and use HTTPS.

TODO: Add info on /static/ file mapping and when a request URL is resolved by NGINX and when its resolved by the melton docker image

TL;DR

Since pictures are worth a 1000 words:

image-1597598234354.png