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.

Revision #4
Created 2 August 2020 11:16:12 by pari
Updated 2 August 2020 12:16:26 by pari