1 Container Virtualization

We used Docker, a more portable, lightweight and efficient alternative to virtual machines to containerize the authentication and url shortener service. Our experience deploying and running services in Docker was very positive, since there is a lot of good documentation online and many of the team already worked with docker before.

Because we implemented all the microservices in the Python Flask framework, we used the `alpine`-based `tiangolo/uwsgi-nginx-flask:python3.8-alpine` base docker image that provides the necessary software infrastructure and entrypoint to run a Python Flask web server. After copying all the required service source files, additional required packages are installed via the `pipenv` and `pip` package manager. To make the installation succeed, we also had to install the `alpine-sdk` apk package that installs all build tools that are required for some `pip` packages to build and install. All further configuration of the services was done by setting environment variables for each microservice.

In order to deploy the services and make the services available to other services, we use `docker-compose`, which creates a docker network for the deployment where each service is addressable via their name through a DNS hostname.

Finally, we provided a `nginx` as well as a `traefik` proxy that exposes both services outside the isolated docker network. The `nginx` proxy is exposed on port 80 and the `traefik` proxy is exposed on port 8081. Both are configured to proxy requests with a URL prefix `/users` to the authentication service (*http://auth*) and all other requests to the url shortener service (*http://shortener*). The only noteable difference using `nginx` compared to `traefik` is in the configuration, which is done via a static configuration file in the case of `nginx` and dynamically using docker labels in the case of `traefik`.

To conclude, we did not experience any issues with using `docker` and `docker-compose` for containerizing and deploying the url shortener application.

2 Container Orchestrations

Using Kubernetes to orchestrate our containers was fairly straightforward. Members of our group have, however, already participated in the Software Containerization course at the VU. This course also covers Kubernetes. Therefore, our experience might not reflect that of someone new to Kubernetes and we might have forgotten certain issues experienced when dealing with Kubernetes in that course. We specifically used MicroK8s[1], which offers "low-ops, minimal production Kubernetes" with "sensible defaults that 'just work'" supporting single-node and multi-node clusters. These "sensible defaults" might have made our experience more smooth compared to using a version of Kubernetes that requires more configuring. Configuration, specifically related to networking, of the cluster and its (Istio) service mesh has been the main difficulty in our whole experience. We do not think the configuration is fundamentally hard, but rather we lack practical experience in setting up these kinds of environments.

References

[1] https://microk8s.io/