Assignment 3

Group 18: Anusha Ali, Priyanka Roy, Jeewon Heo

May 11, 2023

The code for user management and URL shortener services is based on the barebone project provided by the instructors of the course Web Services and Cloud-Based Systems.

1 Docker

Everything related to Docker (assignment 3.1) can be found under the folder `docker/`. We have three services, user management, URL shortener, and database. We created new Docker images for user management and URL shortener based on the Python image. For the database, we used publicly available PostgreSQL image [2]. The user management and URL shortener services each have `app.py` and `utils.py`. `app.py` contains the code for creating a Flask application, the routes for the requests, and their behaviors. `utils.py` contains the help functions for communicating with the PostgreSQL database. The Dockerfiles for both services are also set up very similarly. First, we define a working directory within the container and Flask environment variables. Then, we install packages in `requirements.txt` file. We expose the port (5000 or 5001) and copy the files in the local directory to the container's working directory. Finally, we run the Flask app with host "0.0.0.0" and the port (5000 or 50001).

1.1 Compactness of container

To ensure the compactness of the container, we have used the slim version of the Buster image instead of the full version, which results in a smaller image size. The use of `COPY . .` after installing dependencies ensures that only necessary files are added to the image. One key design decision made in this Dockerfile is to copy the requirements.txt file separately and install dependencies before copying the rest of the project files. This helps to optimize Docker caching, as only the requirements.txt file needs to be reinstalled when dependencies are updated, rather than reinstalling all dependencies every time the code changes.

1.2 Data persistence

Data persistence is achieved by using volumes to store data outside the container. The PostgreSQL service (psql) uses a volume to store the database data, while the URL and user management services use volumes to map the service's directory to the container's file system. For psql container, there are two volume mapping. First, we map the local directory (db) which contains the database initialization SQL queries to the directory `docker-entrypoint-initdb.d` of the PostgreSQL image, which is the directory PostgreSQL looks up to initialize the database. It is very important that the tables are created correctly during the initial run of `docker compose up` because the initialization will not be triggered when the volumes are already created. Another volume mapping is `postgres_data` volume to **/var/lib/postgresql/data** directory in the PostgreSQL image and is used by the psql service to store the user and URL data. The volume will retain even if all containers are destroyed and will persist data for different sessions. When the containers are started using `docker compose up`, the volumes are created (if they do not exist) and mapped into the containers. When the containers are stopped using `docker compose down`, the volumes are not deleted and any data stored in them is preserved.

1.3 NGNIX proxy

The user management, URL shortener, and database services are available on ports 5000, 5001, and 5432. While the database is not directly accessed by users, the other two services are. In order to serve these two services under one port, we employed NGINX's reverse proxy. We have a fourth service called `nginx`. This service allows us to access the services via localhost with port 80. The configuration can be found in

`docker/nginx/nginx.conf`, which specifies that the user management service is reachable via `localhost/users` and URL shortener via `localhost/`.

2 Kubernetes

2.1 Experience working with Kubernetes

Our experience with Kubernetes comprises two phases. The first is based on configuring the cluster on virtual machines, and the second is based on container deployment using Kubernetes.

2.1.1 Cluster setup

This was done following the documentation provided to us in canvas "k8s-setup-commands.pdf". While the initial steps to install docker and Kubernetes was easy to follow, but we faced a few issues when setting up the cluster in three different virtual machines. The biggest problem we faced was a conflict between the CNI plug-in from Kubernetes and the docker containerd. Removing conatinerd and restarting the kubeadm worked at times. The second issue we encountered while setting up the control plane and worker nodes in the provided VMs. Errors like node status "Not Ready", "connection to the server refused" and many more was encountered. To summarize, we encountered different types of errors at different stages and tried to trouble-shoot them either by "tearing down the old cluster set-up and starting again fresh", or "by troubleshooting why the service was failing" and finally by downgrading the Kubernetes version from 1.27v 1.23v. After resolving these issues, everything worked fine.

2.1.2 Deployment

This was done following the official website of Kubernetes and also based on prior knowledge. For our application, we created three separate deployments and three separate services of the micro-services. Our application is comprised of three parts a) database b) user-API and c) URL-API. For each microservice, we created Kubernetes yaml files that took care of the services and the deployments. In this part, Kubernetes was heavily used to deploy our applications which were already dockerized. The Docker images of our user-API and URL-API services were pushed to our public Docker hub repository (`jwnheo/user-api` and `jwnheo/url-api`) to be pulled by Kubernetes deployments. Also, we deploy three replicas for URL shortener service as specified in under `replicas` tag in the deployment file. For database deployment, we have k8s config and secrets network additionally implemented.

2.2 Kubernetes' algorithm for load balancing

There are two types of load balancing in Kubernetes. One is exposing Kubernetes services to the external world and another is used by engineers to balance network traffic loads to those services. When we have multiple replicas of service in Kubernetes, the system needs to determine which replica should handle each request. This is done through a process called load balancing. For exposing to the outside world it uses Ingress service and within the cluster, it uses ClusterIP and NodePort services [1, 3]. Based on observations and documentation, Kubernetes' load balancing algorithm seems to be a combination of round-robin and IP hash.

L4 and L7 Round Robin: The Round Robin approach, equally distributes incoming traffic among all accessible backend Pods. It means that until the allotted number of connections is achieved, each Pod will get an equal amount of requests. Once the limit is met, the traffic is cyclically transferred to the following Pod that becomes available. This behavior can be observed by examining the endpoints of a service using the `kubectl describe svc <service-name>` command. The endpoints list shows the IP addresses of the pods that are currently serving the service, and the order in which they are listed appears to follow a round-robin pattern.

IP-Hash: In addition to Round Robin, Kubernetes additionally supports the client's source IP address-based IP Hash load balancing technique. This technique transfers the originating IP address to one of the accessible backend Pods by hashing it. For stateful applications that demand "session affinity", this makes sure that all requests from the same client are sent to the same Pod which is useful for stateful applications that require session affinity.

2.3 Note on Implementation

1. **(Bonus)** We have set up persistent volume and persistent volume claim for our database. The Kubernetes setup can be found under `kube/db/postgres-storage.yaml`.

2. The user management and URL shortener services are deployed as NodePort services. This allows the services to be accessed via any of the three VM's IP addresses and the node port — 30003 (user) and 30002 (URL). On the other hand, the database service is deployed as a ClusterIP since its only traffic comes from within the cluster. There is no need for NodePort.

3. The services are deployed under namespace shortener-app, so every `kubectl` command must be postfixed with -n shortener-app.

3 Work Division

	Anusha	Priyanka	Jeewon
Code	X	X	X
Report	X	X	X

Table 1: Division of workload.

References

[1] Kubernetes Load Balancer: Expert Guide With Examples - CAST AI - Kubernetes Automation Platform. https://cast.ai/blog/kubernetes-load-balancer-expert-guide-with-examples/.

[2] PostgreSQL. https://hub.docker.com/_/*postgres*.

[3] Services, Load Balancing, and Networking — Kubernetes. https://kubernetes.io/docs/concepts/services-networking/.