1 Authentication service

Since we already architected the URL shortener service to use an abstract key value data store interface and implemented an in-memory version as well as a persistent version using redis[1], we implemented the authentication service in Flask[2] as a thin wrapper around the persistent key value store, where we map a username to its hashed password. In a production deployment, one would probably map another unique identifier to username and password pairs, since usernames could possibly change in the future. However, since there is no REST endpoint to change the username and this is just for demonstration purposes, we just assume the username to be a stable key.

To generate user JWT authentication token after verifying a user's credentials, we use the PyJWT[3] library to generate JWT auth tokens and also encode the username of the authenticated user in the JWT payload, so that it can be subsequently used for identification.

2 URL shortener authentication

As suggested, we expect to receive the JWT user authentication tokens in the `x-access-token` HTTP header field and proceed to validate the token by checking its signature and expiration time using PyJWT[3] as well.

To encapsulate the auth logic to be modular and reusable, we used a Python decorator function `require_auth(f)` that wraps every REST endpoint handler function `f` except for the public `GET` call. The `require_auth` decorator function either

1. calls the wrapped REST endpoint handler function `f` with the username provided in the JWT payload as an injected keyword parameter in case of successful authentication.

2. or immediately returns an appropriate error code in case the JWT token is missing or invalid without proceeding to call the wrapped handler.

Subsequently, each handler function then uses the username from the token payload as the `user_id` we had already implemented before to differentiate between different users and enforce protection of different users shortened URL's.

3 Single Entry Point

We implemented two approaches to consolidate the different microservices and expose them via a single entry point.

1. The fist approach is the usage of a *reverse-proxy* that is put in front of the backend services and routes requests to the appropriate backend microservices based on the host header or the URL for example. As part of the simple docker-compose based deployment model, we used Traefik[4] as a reverse proxy that routes all requests whose URL is prefixed with `/users` to the `authentication` microservice, while all other requests are routed to the `shortener` microservice. To give a concrete example, `http://10.64.140.43/users/login` is routed to the `authentication` service, while e.g. `http://10.64.140.43/2` is routed to the `shortener` service. The configuration of the Traefik reverse proxy is done via docker labels, which removes the need for any configuration files.

2. The second approach uses the Istio[5] service mesh. An Istio `Gateway` and Istio `VirtualService` can be used in conjunction with the Istio service mesh in a Kubernetes[6] (k8s) cluster to monitor and route traffic entering the cluster. This traffic can then be directed by the `Gateway` and `VirtualService` to specific k8s services running in namespaces where Istio's sidecar proxy injection is enabled, which listen on specific ports and route traffic to specific pods that can handle actual user requests. Another slightly simpler solution, that requires less configuration to get started, is to use k8s' standard `Ingress`, which manages external access to the services in a cluster.

4 Scaling

Kubernetes (k8s) offers a *Horizontal Pod Autoscaler* (HPA), which can automatically scale the number of pods in a specific deployment based on e.g. the memory or CPU resource usage of the container, which we used to automatically scale the `shortener` and `authentication` deployments. A HPA allows you to use built-in metrics like average CPU-utilization or custom metrics to determine when a service is (close to being) overloaded and then to automatically deploy new pods, potentially within a minimum and maximum number of pods. New requests can then be directed to these new pods to spread the load on a service, where k8s takes care of the load balancing using e.g. round-robin scheduling. When there is less traffic, the number of pods can be scaled down again.

5 Tracking Microservices

A service mesh, which is a dedicated infrastructure layer for handling service-to-service communication[1], can be used to keep track of all the services. As mentioned already, we used Istio as a service mesh for our deployment. Istio allows one to inject a sidecar proxy for each pod, which intercepts the pods network communication and provides the middleware that enables all the benefits such as traceablility, service discovery, monitoring, amongst others. Intercepting communication using sidecar proxies enables insights as provided by tools such as Kiali[7] that can help to visually keep track of the mesh and its microservice topology, request routing, request rates, and latency, which indicate the location and health of the web services.

[1]https://www.cncf.io/blog/2017/04/26/service-mesh-critical-component-cloud-native-stack/

References

[1] Redis key value database. https://redis.io/. Accessed: 2021-04-11.

[2] Flask microframework. https://flask.palletsprojects.com/en/1.1.x/. Accessed: 2021-04-11.

[3] PyJWT: A python implementation of rfc 7519. https://pyjwt.readthedocs.io/en/stable/. Accessed: 2021-04-28.

[4] Traefik: A modern http reverse proxy and load balancer. https://traefik.io/. Accessed: 2021-04-28.

[5] Istio: Connect, secure, control, and observe services. https://istio.io/. Accessed: 2021-04-28.

[6] Kubernetes: Production-grade container orchestration. https://kubernetes.io/. Accessed: 2021-04-28.

[7] Kiali: Service mesh management for istio. https://kiali.io/documentation/latest/features/. Accessed: 2021-04-28.