# Web Service and Cloud-Based Systems 2023: Assignment 2

Yihong Xi(14720035), Xingyou Li(14741628), Zhiheng Yang(14483262)

## 1 Code Implementation

This section describes and motivates the implementation code of the assignment. According to the description of the assignment, we think it is better to containerize our service thus we can build a proxy component as an entrypoint to forward requests to different services running on different ports. As a result, the whole structure of this project contains four parts: auth_app, shortener_app, proxy and database.

### 1.1 Basic Functions

For authorization service, we build an app called auth_app. At the first part of the code file, we define a user model which includes id, user name and password. Next, we set three routes: "/users", "/users/login" and "/users/validate". The route "/users" hanldes two request methods(GET, PUT). For a POST request, the request body contains "username" and "password". If the "username" is not in our database, the password will be encrypted with md5 and stored in database, then the response with http code 200 will be returned. Otherwise, code 409 will be returned. For a PUT request, the request body contains "username", "old-password" and "new-password". If the "username" and "old-password" are correct, then the password will be updated. Another route "/users/login" is utilized to help users to log in. If user's name and password are correct, a JWT token generated by our algorithm(we will mention it later) will be returned. Finally, for the route "/users/validate", it handles GET method. It will read the JWT token from the attribute "Authorization" in the request header, and decode it. If the token is valid, http code 200 and username will be returned. Otherwise, http code 403 will be returned.

For URL shortener service, we add token validation when routes handle http request methods(exclude GET method in route "/:id"). In the validation progress, firstly, a request with JWT token from the user is sent to our auth_app, then check the response returned. If the response with http code 200, which means the JWT token is valid, corresponding result will be returned to the user. Otherwise, it will return http code 403.

### 1.2 JWT

The implementation of the JWT(JSON Web Token) includes two main methods: "encode" and "decode".

The "encode" method takes in a secret key, the payload and expiration time. Firstly, it creates a header JSON object containing the type of token(JWT) and the algorithm for signing(HS256). Then it serialize the header JSON object and encode it with base64URL encoding. Secondly, it serialize the payload JSON object and encode it with base64URL encoding similarly. Thirdly, it signs the encoded header and pyload with the secret and algorithm "SHA256". Finally, it concatenates the encoded header, payload and the signature to generate the JWT token.

The "decode" method takes in a JWT token and a secret key. Firstly, it split the encoded token into header, payload and signature. Secondly, it verifies the signature by re-calculating it with the secret, encoded header and payload. If the new signature dose not match the one provided, the function will return None. Otherwise, it will check if the token expired or not. If not, the payload will be returned.

To optimize the performance, we only put the expiration time and username into the payload. The expiration time is used for check if token is expired and the username is for authorization validation, which we think are both necessary.

### 1.3 Split Services

To split our services, firstly, we make Dockerfile for each service. Each service is working as a container running on different ports. For example, auth_app is running on port 3000, shortener_app on port 3001, proxy running on port 80 and database on port 5432(Postgres's default port). Secondly, we manage the deployment of all services with docker-compose yaml file. Thus, different services have their own container process which is separated with other services.

### 1.4 Multi-user

As mentioned in the assignment 1, the application has a feature that each data (URL) is assigned an owner, and only the owner can perform PUT or DELETE operations on that URL. We get a JWT token from the request headers attribution "Authorization", then this token is sent to our authorization service. If this token is valid, URL shortener service will get a response with status code 200, which contains the user name. As a result, we can use the user name to query if the user and the corresponding URL are in same row in our database. If someone, not the owner, tries to perform these operations, the application will return a "Authorization Forbidden" response with HTTP code 403.

## 2 Questions

### 2.1 Create an Entry Point for all Services

To create an entry point for all services, we add a proxy component in our containerization services. The proxy application is built with Nginx and contains a Nginx configuration file which can forward requests to different services ports according to the URL prefix. For example, elow is the proxy file content:

```
server {
    listen       80;
    listen  [::]:80;
    server_name localhost;
    location /users {
        proxy_pass http://auth:3000;
    }
    location / {
        proxy_pass http://shortener:3001;
    }
}
```

In our configuration, the proxy service listens to the port 80, so the entry point for the whole services is "http://127.0.0.1:80". Requests with prefix "/users" will be forwarded to "http://auth:3000" while requests with prefix "/" will be forwarded to "http://shortener:3001".

### 2.2 Scale up Services

To scale up services efficiently and independently, we can use Kubernetes to deploy our services. In Kubernetes, there is a feature called Horizontal Pod Autoscaling(HPA) to scale our services based on their respective resource utilization such as CPU or memory utilization.

To implement HPA, we can create two files: auth-hpa.yaml and shortener-hpa.yaml. In both files, we can set minimum and maximum replicas and define the resource utilization threshold which can trigger auto-scaling. Kubernetes will monitor the situation of resource utilization and scale up services automatically by creating replicas.

### 2.3 Microservice Architecture Management

A competent Microservice architecture management should include the following components:

Monitoring: An effective approach can be using containerisation technology like docker to manage different web services over several servers, allowing services to run isolated. Users can use platforms like Kubernetes to monitor the containers for various aspects, such as namespace, location(ip), version, status, age and restart time of the microservices. Also users can use monitoring container such as cAdvisor [1] or simple terminal tool such as lazyDocker [2] to collect the health information such as resource utilization of production environment or containers' running status.

Logging: We can use ELK[3] for collecting logging information. ELK stack consists of three main componnets: Elastic Search, Logstash and Kibana. Elastic Search is a database used for storing all the logs; Logstash is utilized to process the raw logs data before they are sent to Elastic Search; Kibana is a visualization platform which can help us to visualize the logs in Elastic Search. The ability of ELK to visualize data in real time reduces time-to-insights and supports a variety of use cases, which can help us to make a more effective Micro-service architecture management.

Service Mesh: Like the paper[4] introduced before, it includes service discovery, traffic routing, failure handling, and visibility to microservices.

Load Balancing: The 'kubectl autoscale' command can create a Horizontal Pod Autoscaler (HPA), while the 'kubectl describe hpa' command offers insights into the HPA's status and metrics.

Continuous Deployment and Integration: Developers practising continuous integration merge their changes back to the main branch as often as possible, emphasising testing automation to check that the application is not broken whenever new commits are integrated into the main branch. Continuous development allows every change that passes all stages of the production pipeline is released to your customers. There is no human intervention, and only a failed test will prevent a new change from being deployed to production[5].

## 3 Bonus Parts

### 3.1 Containerize Service

To manage multiple services more efficiently and ensure a consistent environment across different platforms, we have containerize all services we created with Docker. Docker is a powerful and reliable tool which can help us to deploy services regardless of the situations in the real production environment.

### 3.2 Nginx Proxy

We utilize Nginx to performance as our entry point to handle incoming requests and forward them to appropriate services. It can improve the performance of services with load balancing which is implemented by adding configuration to proxy.conf file [6].

### 3.3 Postgres

We have chosen Postgres as our database due to its high-performance, high-availability and being thought as one of the most popular database in recent years. It not only compliant ACID(Atomicity, Consistency, Isolation and Durability), but also support advanced data type such as JSON, arrays, etc.

### 3.4 Deployment Productively

We apply Docker Compose to deploy our services. Docker Compose allow users to define and manage multi-container(applications) in one single "docker-compose.yml". With that powerful tool, users can finish the deployment work by using just two simple commands: "docker-compose build" and "docker-compose up -d".

### 3.5 Test Code

The authentication service which is built newly contains several parts. In the software test of this part, a comprehensive test module with the library unittest ensures the main functions, e.g., user creation, updating password and login service, and also their intrinsic features and inner detailed logic (like validity verification, etc.) functioning properly.

## 4 Conclusion

In conclusion, we have successfully developed a micro-services architecture including auth_app, shortener-app_app, database and proxy component with Flask, Postgres and Docker. We have implemented all the basic requests required in the assignment description. Moreover, we created an entry point drived by Nginx to forward requests to different services. As a result, our implementation is believed to be efficient and reliable.

## 5 Contribution Table

| Members | work |
|---|---|
| Yihong Xi | 1. database design<br>2. building basic project<br>3. implement all basic requests and containerization |
| Xingyou Li | Test Unit |
| Zhiheng yang | Test Unit |

Table 1: Contribution Table

## References

[1] G. . Github - google/cadvisor: Analyzes resource usage and performance characteristics of running containers. `https://github.com/google/cadvisor`, mar 29 2023.

[2] J. . Github - jesseduffield/lazydocker: The lazier way to manage everything docker. `https://github.com/jesseduffield/lazydocker`, mar 29 2023.

[3] Knoldus. How to deploy elk stack on kubernetes, 2021. Accessed: 2023-04-19.

[4] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), pages 122–1225, 2019.

[5] Atlassian. Continuous integration vs. continuous delivery vs. continuous deployment, 2022. Accessed: 2023-04-19.

[6] Using nginx as HTTP load balancer. `http://nginx.org/en/docs/http/load_balancing.html`.