Assignment 2: RESTful microservices architectures

Lab Group 32

Ivy Rui Wang (iwa211), Delong Yuan (dyu208)

I. Design

In this assignment, we expanded the URL shortening service from Assignment 1. We created a new microservice—the authentication service—which includes four functions: login, logout, create user, and update password. After successful login, the JWT is passed to the client and verified in subsequent requests. This way, users can only manage their own mapping relationships. Users must provide the JWT to the shortening service for verification before they can use the service to manage their own mappings. Table I summarizes the specifications of each endpoint in the URL shortening service and table II presents the specifications for the identity authentication service.

II. Implementation details

A. Two Microservices

We designed two microservices: a URL shortening service and an authentication service. The URL shortening service cannot be accessed without logging in. Once logged in, users can manage their own URL mappings. After logging out, users need to log in again to access the URL shortening service. The authentication service offers features for login, registration, password change, and logout.

To put these two microservices into a single entry point, we utilized Flask's blueprint mechanism. We defined each microservice as blueprint objects. By registering these blueprints with the Flask application and specifying URL prefixes for each, we achieved a structured and organized URL space for our services.

Endpoint	Path	Method	Body of Request	Return Value
create_short_url	/	POST	{'value': $long\_url$}	{'error': 'Forbidden'}, 403 {'id': id}, 201 {'error': 'Long URL is required in the request body'}, 400 {'error': 'Invalid URL format'}, 400
list_urls	/	GET	-	{'error': 'Forbidden'}, 403 id/url pairs, 200
get_long_url	/⟨id⟩	GET	-	{'error': 'Forbidden'}, 403 {value: long_url}, 301 {'error': 'Short URL not found'}, 404
update_long_url	/⟨id⟩	PUT	{'value': new_long_url}	{'error': 'Forbidden'}, 403 {'value': new_long_url}, 200 {'error': 'New URL is required in the request body'}, 400 {'error': 'Invalid URL format'}, 400 {'error': 'Given ID is not existed'}, 404
delete_url	/⟨id⟩	DELETE	-	{'error': 'Forbidden'}, 403 {'message': id has been deleted'}, 204 {'error': 'id not found'}, 404
delete_all_url	/	DELETE	-	{'error': 'Forbidden'}, 403 {'message': 'All id/url pairs have been deleted'}, 404

TABLE I: Specifications of each endpoint in the **URL shortening service**.

Endpoint	Path	Method	Body of Request	Return Value
logout	/users/logout	GET	-	{'error': '401 Unauthorized', 'msg':'You are not logged in'}, 401 {'msg': 'Logout successfully'}, 200
login	/users/login	POST	{'username': $username$, 'password':$password$ }	{'error': 'Forbidden'}, 403 {'msg': 'Login successfully', 'token':JWT}, 200
update_password	/users	PUT	{'username': $username$, 'password':$password$, 'new_password':$new\_password$ }	{'error': 'Forbidden'}, 403 {'msg': 'Password updated successfully'}, 200
create_user	/users	POST	{'username': $username$, 'password':$password$ }	{'detail': 'duplicate', 'msg': 'Username already exists'}, 409 {'msg': 'User created successfully'}, 201

TABLE II: Specifications of each endpoint in the **authentication service**.

B. Concurrent Users

We utilized the threading package to manage concurrent access to in-memory dictionary variables storing user information for the authentication service and URL data for the URL shortening service. Through the use of locks, we ensured that multiple users could perform CRUD operations on the data without encountering conflicts or disorder. Additionally, all API endpoints are serverless, and user information is stored within JWT tokens, thereby ensuring seamless handling of concurrent user requests without any issues.

C. How to Generate JWT

The process of creating a JWT involves encoding three main parts—Header, Payload, and Signature—using Base64, and then concatenating them to form the JWT.

The Header typically consists of two parts: the type of token, which is JWT, and the hashing algorithm being used, such as HS256 for HMAC SHA256. In this assignment, the original header is like this:

```
{"alg": "HS256", "typ": "JWT"}
```

The Payload part is also a JSON object, containing the actual data to be transmitted. In our assignment, the original payload looks like this:

```
{"username": username}
```

In brief, we create a JWT by encoding the header and payload as Base64 strings, generating a signature based on these encoded strings and a secret key, and then assembling the encoded header, payload, and signature into the final JWT format. Specifically, the header and payload are JSON objects that are first converted to strings using json.dumps(). These strings are then encoded to bytes using .encode('utf-8'), Base64 encoded with base64.b64encode(), and finally decoded back to strings with .decode('utf-8') for easy concatenation. The Signature is created by applying a signing algorithm specified (HS256) in the header to a string consisting of the encoded header, a period (.), and the encoded payload, using a secret key. The signature is then encoded in Base64. The encoded header, payload, and signature are concatenated together with periods (.) separating each part, forming the complete JWT.

D. Two Ways of Storing JWT: in Request Headers or Cookies

In this assignment, we have tested two ways to store JWT on the client side either in the request headers and cookies. By implementing in either ways, we understood that both methods—storing JWT in request headers and cookies—can effectively store JWT for authenticating and maintaining sessions.

When stored in the request headers, JWT is typically added to the Authorization header with the Bearer schema. In test_app.py, we can see that the client includes the JWT in the header of each HTTP request. Our flask server then extracts the JWT from the Authorization header, validates it, and processes the request accordingly.

```
jwt = request.headers['Authorization']
```

Alternatively, JWT can be stored in cookies. When a JWT is stored in a cookie, the browser automatically sends the cookie with each request to the domain that sets the cookie. Our flask server then extracts the JWT from the cookies, validates it and processes the request accordingly.

```
jwt = request.cookies.get('jwt')
```

The choice between storing JWT in headers or cookies depends on the specific requirements of the application and the security considerations. Storing JWT in headers is more common in API-driven applications, while cookies are often used in traditional web applications. Cookies can offer some additional security features, such as the ability to set the Secure and HttpOnly flags, but they can also be vulnerable to cross-site request forgery (CSRF) attacks if not properly protected.

E. How to Validate a JWT's Validity

Before each access to the URL shortening service, user identity is verified through JWT. To prevent the use of tampered JWTs, it is necessary to validate the authenticity of the JWT. Our approach is to split the JWT into its three constituent parts: the encoded header, payload, and signature. We then reconstruct the signature using the original header and payload combined with a secret key, aiming to match this with the received signature.

The purpose of this process is to confirm the JWT's validity by ensuring that the signature can only be correctly generated with our secret key, thus preventing unauthorized modifications and enhancing the security of our system.

F. How to Logout

In our logout implementation, we first check if the JWT exists in the cookies of the incoming request. If not found, indicating the user is not logged in, we return a 401 Unauthorized error. If the JWT is present, we proceed to log the user out by deleting the 'jwt' cookie, effectively removing their session. This is done using the

```
response.delete_cookie('jwt')
```

If the cookie deletion is successful, we return a message indicating a successful logout. Logging out can also be implemented by making the JWT cookie expire immediately. This is achieved by using

```
response.set_cookie('jwt', jwt, expires=0)
```

By setting the expiration time of the JWT cookie to 0, the cookie is invalidated instantly, effectively logging the user out.

III. Answers to the Questions

1) To create one entry point for all of the microservices, Flask's blueprint mechanism could do the job. The blueprint mechanism is for modular and route organization, which allows us to split the application into smaller, reusable modules. In this case, we have an application that's combined with two microservices, which could be considered as two modules of the application.
   First, define all the microservices (shorten-URL and authentication services) to a blueprint object:

```
authentication_service =
↪ Blueprint("authentication_service",
↪ __name__)
```

Then register them into the Flask application while specifying their URL prefix (route registration):

```
app.register_blueprint(authentication_
↪ _service,url_prefix='/users')
```

Then these two microservices are organized in the same logical unit and share the same port as well.

2) After deploying microservices on Kubernetes, they could be scaled up independently by using Horizontal Pod Autoscaler(HPA).
   The HPA is a horizontal scaler for the target Deployment or ReplicaSet to scale. There are some metrics for the Autoscaler to decide when to scale, for example, scale when CPU utilization exceeds a certain threshold and decreases when it falls below another threshold.
   By using HPA, each deployed microservice could automatically scale up according to the rules set to them. For example, after deploying the application as 'url-login-app' with deployment 'url-login-app-deployment', we can simply use a kubectl command to add an HPA for this deployment:

```
kubectl autoscale deployment
↪ url-login-app-deployment
↪ --cpu-percent=80 --min=3 --max=5
```

   Here we specify automatic scaling (by increasing the number of pods) when CPU usage exceeds 80%. The number of pods would not be more than 5 or less than 3.
3) To distinguish different microservices, the IP address, hostname, and port number of the service should be collected. For each service's health status, the response time, throughput, workload, and service response/queue time should be collected. Also, the logs and network traffic could be collected for analysis.
   For providing such metrics, each service should expose an endpoint (for example /metrics) and provide metrics there. Then the manager could know the metrics such as health status or location from each service by analyzing the text they get from that endpoint. The manager can proactively pull information by sending requests periodically to microservices' endpoints for the latest metrics data.
   In industry, there are some popular monitoring tools like Prometheus. Prometheus could collect metrics like CPU utilization, memory usage, disk space utilization, network traffic, and request throughput. Prometheus also supports customized service metrics like queue length or tasks processed. This requires generating metric data in the service application that conforms to the Prometheus format and exposing it to the Prometheus through the HTTP endpoint.

IV. Responsibilities

Task	Ivy Wang	Delong Yuan
Specification Design	Collaborator	Collaborator
Code Implementation	Collaborator	Collaborator
Writing Report	Collaborator	Collaborator

TABLE III: Division of responsibilities between team members