Assignment 2: RESTful Microservices Architectures

Group 6

Introduction

In this assignment we create another second microservice that is responsible for user authentication. This service will extend the URL-Shortener that was prepared in the first assignment by protecting some of its actions from unauthorized use.

In principle, the goal of the assignment is to further explore RESTful architecture, this time focusing on the cooperation between various microservices and how they can be combined to work together.

Implementation

This section describes the implementation after successful completion of points 1 and 2 of the instruction. The extensions made to complete the point 6 are described in the next sections.

The app is divided into two services: the authentication service and the actual shortener service. The new addition, authentication service, is based on the JSON Web Tokens (JWT), an approach where JSONs are encrypted using a private key. The protected service compares the received token and the one generated using a defined key and this way knows if the received token was created using the same secret key.

Shortener Service Update

This part is based on the project prepared for the first assignment. In order to focus on the sole microservices and make it easier to test the application, we decided to simplify the responses made by the requests. Previously, some of the requests were resulting in HTML files generated using template mechanism (still available in the *templates/* directory and via *db_pandas.py*, however without authentication features). Therefore, there are several changes in the URL shortener, briefly described below:

- URL mappings are stored only in the memory (as required according to the instruction).
- Pandas data structures are no longer used to manage the mappings, a simple dictionary is used instead.
- Operations performed on the data (like adding, fetching or deleting) are available through methods of a *Database* class storing the dataset (in *db_urls.py*) — no actual database is used though.
- We also updated what each route is returning so that the content and HTML codes match the ones described in the instruction.

RESTful User Service for User Authentication

We created a UserDatabase class to store and manage the data. The class consists of a dictionary (that holds users' names and passwords) and two methods: *create_new_user() check_user_credentials()*, both taking user name and password as arguments. The former one creates a new user after checking if the username has not been used before (otherwise, a *KeyError* is thrown) and the latter one checks the credentials against the ones already stored in the app and returns the token.

The other addition is the main Python file of the authorization service — *auth_app.py*. This one contains the Flask app and defines the routes handled by it: create_new_user() handles POST requests to /user and *login_user()* handles login POST request to /user/login. Both accept *username* and *password* fields passed as the form-data (due to past template approach).

Creating a new user fails if either username or password is not present in the form data passed with the request or when the username is already registered. The app throws 400 HTML exception then. Successful user creation is concluded with a simple “success” JSON and HTML status 200.

Logging in is only slightly more complicated. Again, the username and password fields must be present in the form-data of the request. Second, the username and password must match the records in the dictionary. Failure, in either case, leads to throwing 403 “Forbidden” HTML exception. If the fields are correct, then JWT framework generates the token using the private key defined earlier and returns in a JSON.

The authentication in the URL-Shortener is added with a single decorator that is placed before each function that has to be protected. Decorator is shown in the listing on the right. It raises the 403 HTML exception when the token is missing or encoded using an incorrect private key. When token is correct, the decorated method may continue.

```
@wraps(f)
def decorated(*args, **kwargs):
    if 'x-access-token' in request.headers:
        token = request.headers.get('x-access-token')
    else:
        abort(403)
        return
    try:
        jwt.decode(token, private_key)
    except jwt.DecodeError:
        abort(403)
    return f(*args, **kwargs)

return decorated
```

To create this part of the assignment, we relied on the resources available with the Flask tutorial by Anthony Herbert[1], including a specific one addressing the JWT authentication[2].

Multiple services using one port (Question 3)

One of the possibilities would be to utilize the API Gateway pattern, which, for instance, means creating another microservice that may be called “edge service” and that will be accepting the requests as a gateway and direct them to appropriate actual services and then return the answer to the client. This way we introduce additional abstraction level, so we can easily replace or redirect services to our liking.

There are ready-to-use servers offering such feature, like Apache httpd or Nginx that can work as a reverse proxy (one that retrieves resources from multiple services for the client), that in a way implements the idea described above.

Services Load Balancing (Question 4)

Using the gateway service, we obtain an easy way to control the requests and the way our system handles them. When one of the services is experiencing increased load, gateway may use more than one instance of the microservice. Servers like Apache or Nginx have ready load balancers to use in such cases. For instance, the concept of edge service that handles the requests from the outside is used in companies like Netflix[3].

---

[1] Python Web Development Tutorials, Pretty Printed, https://prettyprinted.com

[2] “Creating a RESTFul API in Flask With JSON Web Token Authentication and Flask-SQLAlchemy”, Pretty Printed, https://youtu.be/WxGBoY5iNXY

[3] “How to load-balance microservices at web-scale”, Martin Goodwell, https://www.dynatrace.com/news/blog/load-balancing-microservices/

Tracking Distributed Microservices (Question 5)

The problem seems to be best addressed by a technique called “distributed tracing”[4]. In principle, each transaction performed within the system carries a unique ID, that can be used to gather information about the way particular services were used in a single transaction and then processed together by a logging service to further analyze the whole system or find a transaction that resulted in a failure.

Health of the microservice might be checked via requests sent on-demand or cyclically by the microservice, reporting on its status, the status of its components or conveying more complex information. These checks might be utilized by orchestration tools, displayed in the dashboards or trigger more critical alerts.

Additional Implementation Extension (Question 6)

We decided to implement (infrastructure-wise) the API Gateway using Nginx server (as a side effect, also load balancing), that allowed us to run gateway service in a few simple steps. After installation of the Nginx server on a local machine, we needed to add a single addition to the configuration file:

```
server {
	listen 80;
	server_name localhost;

	location /user {
		proxy_pass http://127.0.0.1:5001;
	}

	location / {
		proxy_pass http://127.0.0.1:5000;
	}
}
```

Line listen 80 defines that the reverse proxy will be available on port 80. Server_name is left at the default value, later should be changed to the actual domain. Then there are two location blocks for /user and / locations, and the requests to those locations will be passed to the services running at the addresses defined in the proxy_pass field.

These modifications are already leading to the working API gateway. Requests to localhost:80/user/, localhost:80/user/login are being passed to the authentication service and localhost:80/ to the URL-Shortener. Further modifications in the Nginx could, e.g., enable the load balancing features.

---

[4] "Implement distributed tracing in your microservices-based app”, IBM, https://www.ibm.com/garage/method/practices/code/distributed-tracing