University of Amsterdam

Assignment 2 Report - Web Service and Cloud-Based Software

Group 13

February 2026

1 Implementation

1.1 Authentication

The authentication service handles user registration and login. When a user registers (`POST /users`), their username and hashed password get stored in the database or memory. A user can change their password by sending a `PUT` request to `/users` with their username and old and new password. When a user logs in (`POST /users/login`), their credentials are checked and if the credentials are correct, the service returns a *JSON Web Token* to the user. A token can be verified by sending it to the `POST /auth/verify` endpoint.

1.1.1 JWT

A JWT is a token that consists of a base64 encoded header, payload, and signature. The base64 encoding and decoding are handled by functions from the Python standard library. The header communicates the signature algorithm, which is `HS256` in our case.

The payload contains the fields `username`, `iat`, and `exp`. `username` is the username of the logged in user. `iat` stands for issued at and stores the time at which the token is created in seconds since 1970. `exp` contains the expiry time for the token. Tokens are valid for one hour to limit the damage that an attacker can do when a token leaks. `iat` and `exp` are *registered claims* that are predefined and recommended by the IETF. [1]

Signature verification is implemented using the `hmac` and `hashlib` modules from the Python standard library. Before signing, the header and payload are converted to base64 and concatenated with a dot character. A signature is generated by calculating the HMAC of the concatenation and a secret value. When verifying a JWT, a new signature is generated and compared with the signature in the JWT.

1.2 URL shortener

The URL shortener service exposes a REST-style API for creating, resolving, updating, and deleting shortened URLs. For this assignment, to improve the security, almost all endpoints require JWT authentication from the authenticator. For every protected request, the shortener service forwards the provided token to the authentication service for verification. If verification fails or the token is missing, the request is rejected with HTTP 403. The endpoint `POST /` generates a new short identifier for the provided URL and associates it with the authenticated user as the owner. The update (`PUT /<id>`) and delete (`DELETE /<id>`) endpoints enforce ownership, returning 403 if the requester is not the creator of the mapping and 404 if the identifier does not exist. The listing endpoint (`GET /`) returns only the identifiers owned by the authenticated user. The public resolution endpoint (`GET /<id>`) remains unauthenticated and issues an HTTP 301 redirect to the original URL when the identifier exists, or a 404 error otherwise. This integration ensures that URL management operations are restricted to authenticated users while preserving public access to shortened links.

[1] `https://datatracker.ietf.org/doc/html/rfc7519#section-4.1`

2 Design

Sequence diagram for Create Short ID

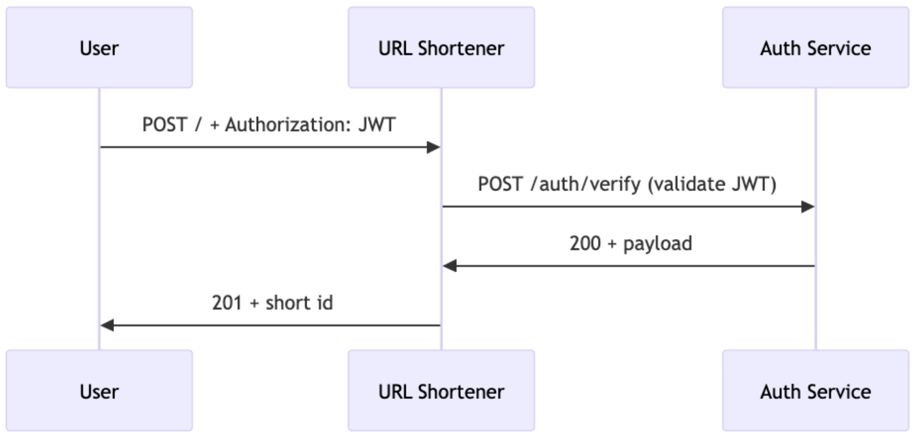


Figure 1: Sequence diagram for Create Short ID

3 Questions

Q1: Single entry point for all services

A single entry point can be provided by placing a reverse proxy in front of the microservices as an API gateway. Instead of exposing each service on a different port (e.g., 8000 and 8001), the gateway listens on a single port (e.g., 80) and routes requests to the appropriate backend based on the URL path. In this project, Nginx acts as the gateway. It exposes a single host and port and uses path-based routing: requests to `/auth/*` are forwarded to the authentication service, and requests to `/shortener/*` are forwarded to the URL shortener. Clients only need to know the gateway address and path; backend ports and hosts remain hidden.

Q2: Independent Scaling

Services can be scaled independently by running multiple instances of each microservice and distributing incoming traffic across them via a load balancer. In this project, Nginx is configured with upstream blocks that define multiple backend servers per service, using round-robin load balancing (default). The auth service has three instances (auth-service-1, auth-service-2, auth-service-3), and the URL shortener has four (url-shortener-1 through url-shortener-4). Each new request is sent to the next server in the rotation. Scaling is done by changing the number of instances in the configuration; for example, the shortener can run four instances while the auth service runs three. Docker Compose is used to run and manage these instances. All instances share the same MongoDB database, so data stays consistent. To scale even further, more instances can be added to the configuration and the gateway reloaded; the load balancer then distributes traffic across the updated set of backends. If autoscaling is required, a container orchestration platform such as Kubernetes can be used. Kubernetes supports horizontal pod autoscaling (HPA), which periodically checks metrics (e.g. CPU utilisation, healthchecks) and increases or decreases the number of pod replicas for each service independently when load changes

Q3: Managing microservice architecture

Managing a microservice architecture across multiple backend servers requires visibility into which services and instances are running, where they are located, and whether they are healthy. Relevant metrics include:

- *health* - whether each instance is responding, e.g. via a `/health` endpoint, so unhealthy instances can be removed from load balancing

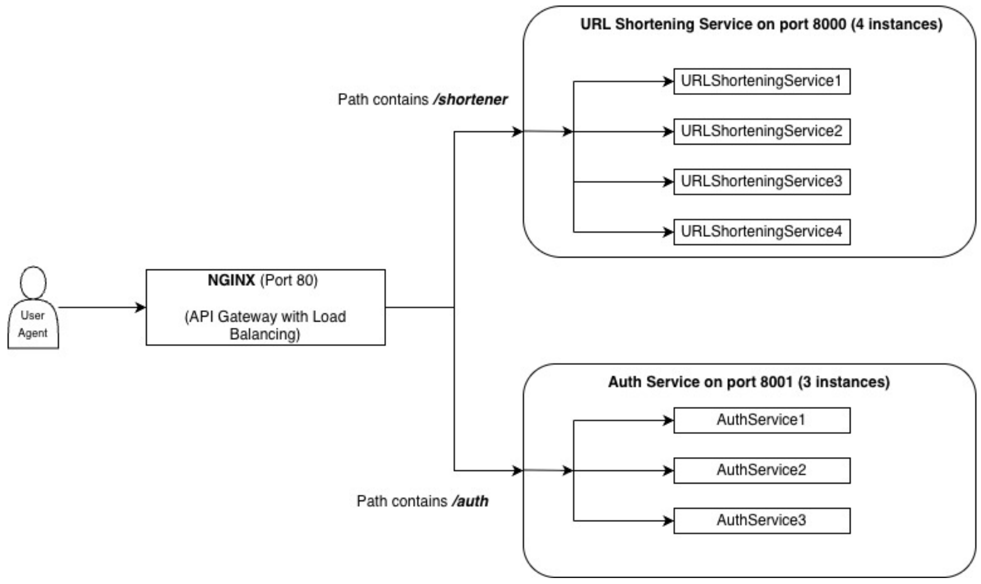


Figure 2: NGINX as API Gateway with Load Balancing

- *location/identity* - which instance handled each request, so that traffic and failures can be traced to specific replicas
- *request counts and latency* - to detect overload or slowness and inform scaling decisions.

4 Bonus

4.1 Nginx

In this project, Nginx is used as both the API gateway and the load balancer for the microservice architecture. As an API gateway, it exposes a single entry point on port 80 and routes incoming requests via path-based routing:

- `/auth/*` is forwarded to the authentication service;
- `/shortener/*` is forwarded to the URL shortener service.
- `/frontend/*` is forwarded to the frontend service.

As a load balancer, it distributes traffic across multiple instances of each service using a round-robin algorithm: three instances of the authentication service and four of the URL shortener. This improves availability and allows each service to be scaled independently. The `X-Instance-ID` response header identifies which backend instance processed each request and can be used to verify load distribution.

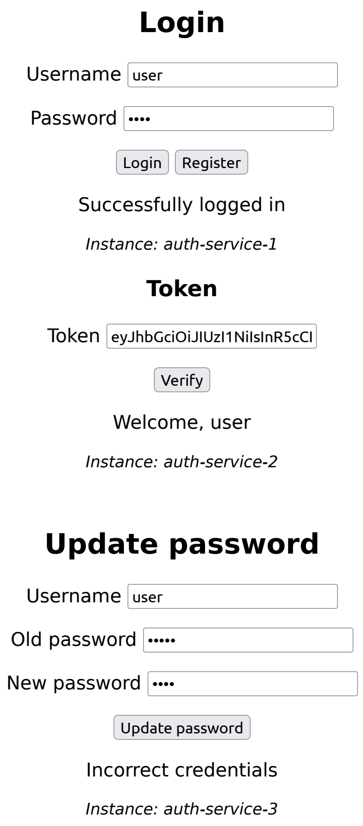
4.2 Frontend UI

Since the URL shortener API now requires a token for most endpoints, we had to include a menu for logging in to the frontend. Figure 3 shows a screenshot of the login menu. After the user logs in, the token gets automatically filled in the *Token* input field. From there, the token is used for all API calls that require the token.

We also added an indicator that tells the user which instance handled the request. This is possible because each service instance returns the `X-Instance-ID` header.

Another change since the previous version is that the frontend is now hosted by its own service. Previously, the frontend was part of the URL shortener service. We changed this because it was easier to integrate with Nginx and to decouple it from the URL shortener.

5 Division of work



Login

Username user

Password ••••

Login Register

Successfully logged in

Instance: auth-service-1

Token

Token eyJhbGciOiJIUzI1NiIsInR5cCI

Verify

Welcome, user

Instance: auth-service-2

Update password

Username user

Old password •••••

New password ••••

Update password

Incorrect credentials

Instance: auth-service-3

Figure 3: The login-related section of the frontend

Members	Contribution
Quincy Koper	New/updated API endpoints based on requirements and tests, database support for the auth service, JWT payload design, frontend UI
Albert Bao	JWT authentication and verification implementation, password hashing and verification, update API endpoints for url shortener to enable authentication
Harikrishna	Added API Gateway for url shortening and auth service, and Implemented Load Balancing using NGINX

Table 1: Members and their contributions