Implementation Description

Overall Design

We implemented **two microservices**:

1. **Authentication Service**
   - **Port:** 5001 (default)
   - **Files:** `auth.py` (main blueprint), `user_storage.py` (SQLite user database), and `jwt.py` (custom JWT logic).
   - **Responsibilities:**
     - User registration (`POST /auth/users`)
     - User login + JWT issuance (`POST /auth/users/login`)
     - Password changes (`PUT /auth/users`)
     - Token validation (`POST /auth/validate_token`)
2. **URL Shortener Service**
   - **Port:** 5000 (default)
   - **Files:** `app.py` (Flask main), `url_storage.py` (SQLite storage).
   - **Responsibilities:**
     - Creating and managing short URLs (`POST /`, `PUT /<id>`, `DELETE /<id>`, etc.).
     - Enforcing ownership (only the URL's creator can modify or delete).
     - Requiring valid JWTs for any action that manipulates a user's URLs.

JWT Implementation

- **How:**
  - Implemented in `jwt.py` using HMAC-SHA256. We build a **header** (`alg: HS256`, `typ: JWT`) and a **payload** with:
    - `sub`: the user's unique ID from the `users` table
    - `exp`: the expiration timestamp (by default, set to 1 hour from issuance)
  - We Base64URL-encode both header and payload as strings, then sign them with `secret_key` via `hmac.new(..., sha256)`.
  - Final format: `header_b64.payload_b64.signature_b64`.
- **Token Contents:**
  - We keep it minimal: `user_id` (as `sub`) plus expiry. This ensures short tokens and efficient signature generation.
- **Validation in Shortener Service:**
  - The `@jwt_required` decorator extracts the token from `Authorization: Bearer <token>` headers.

    - Calls `validate_jwt(token, JWT_SECRET)`; if signature or expiry is invalid, returns **403**. Otherwise, the decorator injects `user_id` into the route function.

Multi-User Extension

- **Ownership Enforcement:**
    - Each row in the `urls` table has a `user_id` foreign key (matching the user's ID in the `users` table).
    - When a user attempts to `PUT` or `DELETE` a given URL, the service checks if `url_data['user_id'] == user_id_from_token`. If not, respond with **403**.
- **Routes:**
    - `POST /`: Creates a short URL for the logged-in user.
    - `PUT /<id>` / `DELETE /<id>`: Modify or delete only if the current user is the owner.
    - `GET /my-urls` or `GET /`: Returns all short URLs for the authenticated user.
    - `GET /<int:url_id>`: Publicly fetches/redirects to the original URL (returns 301 or 404).

Questions

i. Single Entry Point for All Microservices

In real-world deployments, we usually place a **reverse proxy** (e.g., NGINX, Traefik, or HAProxy) or an **API gateway** in front of multiple services. It listens on a single public port (80/443), then routes requests internally:

- `example.com/auth/...` → Authentication service (port 5001)
- `example.com/short/...` → Shortener service (port 5000)

This setup provides a single external endpoint for clients while maintaining separate internal services.

ii. Scaling Services Independently

If we see high load on one microservice (e.g., the authentication service), we can horizontally scale just that service by launching additional container instances behind a load balancer. Meanwhile, the shortener might remain at a smaller scale if it sees less traffic. Tools like **Kubernetes** or **Docker Swarm** handle auto-scaling based on CPU/RAM usage or custom metrics, letting each microservice scale independently.

iii. Managing a Distributed Microservice Architecture

To manage many services on multiple servers:

- **Service Discovery**: So each service can find where the others are (especially if containers move around).
- **Monitoring/Metrics**: Tools such as Prometheus + Grafana collect metrics (e.g., response times, request counts, memory usage), displayed on dashboards.
- **Logging**: Aggregating logs (e.g., ELK stack: Elasticsearch, Logstash, Kibana) makes debugging easier.
- **Health Checks**: Each service can offer a simple `/health` route so orchestration systems (like Kubernetes) or load balancers can verify it's alive.

iv. Technologies in Mind

- **NGINX** or **Traefik** for reverse proxy / single entry point.
- **Docker Compose / Kubernetes** for container orchestration and auto-scaling.
- **Prometheus & Grafana** for real-time monitoring, plus alert rules on latencies or error rates.
- **ELK Stack** for centralized log management.

Bonus Part

Our system implements JWT token invalidation via the `/users/logout` endpoint. Key features:

1. **Blacklist Mechanism**: When users choose to logout, the tokens would be added into the blacklist to stop their utilization within 2 hours.
2. **Automatic Cleanup**: If there is a user logout, the expired tokens would be deleted.
3. **Security**: Double validation (signature + blacklist check) and atomic database operations.
   API returns standardized JSON responses (200/500) for success/failure notifications.

Yueming Sun	Code development for jwt authentication
Hongyu Chen	Report writing and some code development
CY Yau Chun Yuen	Report writing