INFO8011: Microservices Infrastructure in Docker

Emilien Wansart
Emilien.Wansart@uliege.be

Maxime Goffart
Maxime.Goffart@uliege.be

Academic Year 2025-2026

> **Group size**: 1
> **Weight**: 20%
> **Deadline**: 14 November 2025

1. Introduction

In this assignment, you will create a microservices infrastructure in Docker. The topology aims to replicate a simple social media platform, composed of multiple services.

The main goals of this assignment are to gain an insight into modern microservices architectures by implementing some of their components in Go or Python, as well as acquiring a deeper understanding of Docker.

2. Architecture Diagram

The architecture diagram is shown in Figure 1. Each box (excluding the client) corresponds to a Docker container. Note that **the Prometheus container is not shown**. Microservices are in green, databases in red, an arrow symbolizes an HTTP request, a red arrow means that the request uses HTTPS.

The user-service and post-service along with their databases are given to you in the project sources.

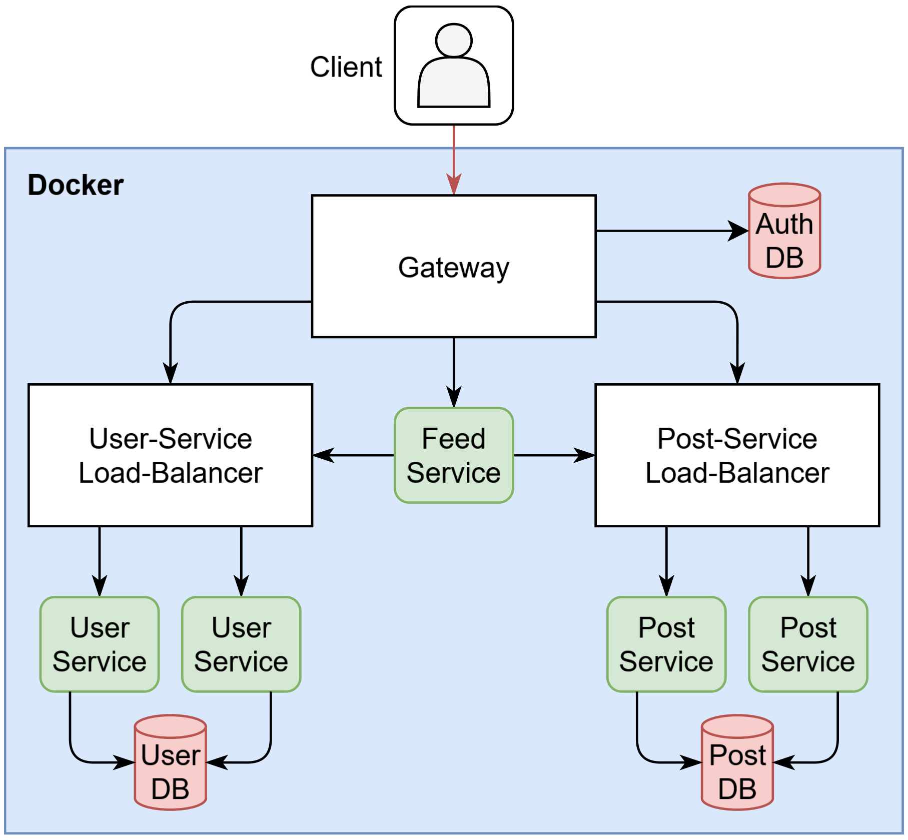


Figure 1: Microservice infrastructure diagram.

3. Architecture Components

3.1. Gateway

The gateway is the single entry-point of the architecture. It has the following features:

- Act as an authentication service for clients to register and login to the social media platform.
- Act as reverse proxy between the clients and the microservices.
- Include a Prometheus integration for monitoring.

Communications between the clients and the gateway are encrypted using TLS. Communications inside the architecture (past the gateway) remain in plain text.

3.1.1. Authentication Service Role

The gateway allows clients to register and login to the platform. The gateway is backed by a database that stores the registered clients. The data include their emails, passwords, and a universally unique identifier (UUID)[1] for each user.

- The `POST` method on `/api/auth/register` allows the client to register to the platform. Clients can only register once.

  ```
  POST /api/auth/register
  Content-Type: application/json
  ------------------------------------------------
  {
    "email": "<email>",
    "password": "<password>"
  }
  ```

- The `POST` method on `/api/auth/login` allows the client to log in to the platform. If the credentials correspond to ones used at the registration stage, an access token is returned. The number of access token that can be provided is **limited to 10 per hour.**

  ```
  POST /api/auth/login
  Content-Type: application/json
  ------------------------------------------------
  {
    "email": "<email>",
    "password": "<password>"
  }
  ```

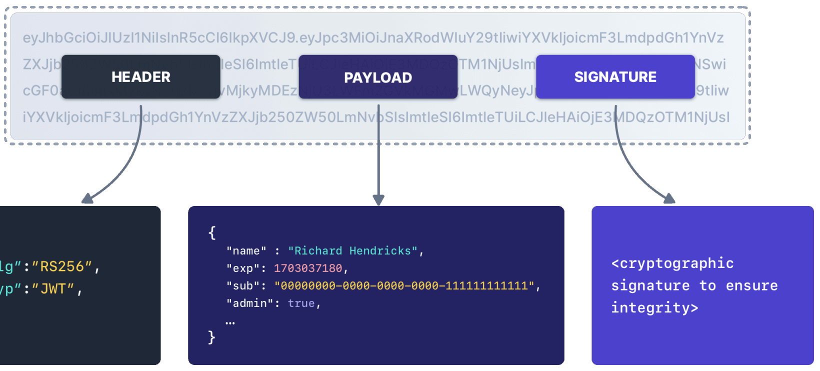
  Example of JSON response:

  ```
  {
    "access_token": "<token>"
  }
  ```

The access token returned by the gateway at login is a JSON Web Token (JWT)[2]. The gateway only implements the most basic form of JWT, which works as follows:

1. Client authentication. When a client submits valid credentials (email and password) to the `/api/auth/login` endpoint, the gateway generates a JWT.
2. Token structure. The JWT consists of three parts:
   - a header specifying the signing algorithm (HS256 in our case);
   - a payload containing claims, which, in our case, only includes the subject (`sub`, the user UUID) and the expiration time (`exp`);
   - a signature created by hashing the header and payload with a secret key.

[1]See https://en.wikipedia.org/wiki/Universally_unique_identifier.
[2]See https://www.jwt.io/introduction#what-is-json-web-token.



```
{
  "alg":"RS256",
  "typ":"JWT",
  …
}
```

```
{
  "name" : "Richard Hendricks",
  "exp": 1703037180,
  "sub": "00000000-0000-0000-0000-111111111111",
  "admin": true,
  …
}
```

```
<cryptographic
signature to ensure
integrity>
```

Figure 2: JWT structure (Source: https://fusionauth.io/articles/tokens/jwt-components-explained).

3. Stateless authorization. The token is returned to the client and must be included in the `Authorization: Bearer <token>` header of subsequent requests.
4. Validation at the gateway. Upon receiving a request, the gateway verifies the token's signature using the secret key. If the token is valid and not expired, the user UUID is extracted from the claims and injected into the request headers (`X-User-ID`) before forwarding the request to the corresponding microservice.

3.1.2. Reverse Proxy Role

The gateway also fulfills the role of a reverse proxy, forwarding incoming user requests to the correct microservice.

The proxy logic is based on the URL path:

- Authentication endpoints (**unauthenticated endpoint**)
  - `/api/auth/register` ⟶ handled directly by the gateway (local implementation).
  - `/api/auth/login` ⟶ handled directly by the gateway (local implementation).
- User-service endpoints (**authenticated endpoint**)
  - `/api/profile/me` ⟶ forwarded to a user-service via the user-service load-balancer.
  - `/api/profile/{userId}` ⟶ forwarded to a user-service via the user-service load-balancer.
  - `/api/friends` ⟶ forwarded to a user-service via the user-service load-balancer.
- Post-service endpoints (**authenticated endpoint**)
  - `/api/posts` ⟶ forwarded to a post-service via the post-service load-balancer.
  - `/api/posts/{userId}` ⟶ forwarded to a post-service via the post-service load-balancer.
- Feed-service endpoints (**authenticated endpoint**)
  - `/api/feed` ⟶ forwarded to feed-service directly.

Example of a request from client to gateway (on the left), then from the gateway to the corresponding service (on the right):

```
POST (<gateway_URL>)/api/profile/me
Content-Type: application/json
Authorization: Bearer <token>
________________________________________

{
  "username": "<username>",
  "bio": "<biography>"
}
```

Gateway forwards
⟶

```
POST (<user_L-B_URL>)/profile/me
Content-Type: application/json
X-User-ID: <UUID>
________________________________________

{
  "username": "<username>",
  "bio": "<biography>"
}
```

3.1.3. Prometheus Integration

Prometheus[3] is an open-source monitoring system that collects and stores time-series metrics via HTTP scraping.

[3]See https://prometheus.io/.

To observe and analyze the requests made to the gateway, you will define 2 metrics to expose via the `/metrics` endpoint of the gateway:

- `gateway_requests_total`: Record the number of requests sent to each endpoint.
- `gateway_request_latency_seconds`: Measure the latency of requests per endpoint.

You are free to choose the metric types and labels. See Section 3.4.2 for more information.

3.2. Microservices

All HTTP requests made to the services must include an HTTP header field named `X-User-ID` holding a UUID, which uniquely identifies a user. This UUID will be provided by the gateway.

3.2.1. User-service

Do not implement! **This microservice is provided to you in the project sources.**

This microservice stores the information related to the users, which includes their usernames, biographies, and friends lists. There are 2 identical instances of this service, backed by a shared database.

- The `GET` method on `/profile/me` allows to fetch the profile of the user identified by `X-User-ID`.

```
GET /profile/me
Content-Type: application/json
X-User-ID: <UUID>
```

- The `GET` method on `/profile/{userId}` allows to fetch the profile of the user identified by `userId`. `X-User-ID` is unused but required to ensure authentication.

```
GET /profile/{userId}
Content-Type: application/json
X-User-ID: <UUID>
```

Example of JSON response:

```
{
  "uuid": "3b1be25a-d7eb-4e49-9a87-adf1364f4dca",
  "username": "CortezTheKiller",
  "bio":"Famous spanish conquistador"
}
```

- The `POST` method on `/profile/me` allows to create *or* update the profile of the user identified by `X-User-ID`. This is typically the first request made after the first login.

```
POST /profile/me
Content-Type: application/json
X-User-ID: <UUID>
────────────────────────────────
{
  "username": "<username>",
  "bio": "<biography>"
}
```

- The `GET` method on `/friends` allows to fetch all friends of the user identified by `X-User-ID`.

```
GET /friends
Content-Type: application/json
X-User-ID: <UUID>
```

Example of JSON response:

```
[
  "5d815531-cc3a-44e2-9a49-c82ba2097c92",
  "aba1272b-c35a-4c4f-a869-2227a0867c7f",
  "ce0bfcb5-9085-4218-bbd3-93d3eb1b8db8"
]
```

- The `POST` method on `/friends` allows to add a friend to the user identified by `X-User-ID`. A friend can only be added if it exists in the database.

```
POST /friends
Content-Type: application/json
X-User-ID: <UUID>
────────────────────────────────
{
  "friend_uuid": "<UUID>"
}
```

- The `DELETE` method on `/friends` allows to remove a friend from the user identified by `X-User-ID`.

```
DELETE /friends
Content-Type: application/json
X-User-ID: <UUID>
────────────────────────────────
{
  "friend_uuid": "<UUID>"
}
```

3.2.2. Post-service

Do not implement! **This microservice is provided to you in the project sources.**

This microservice stores the posts that are posted by the users. A post is simply a piece of text. There are 2 identical instances of this service, backed by a shared database.

- The `GET` method on `/posts/me` allows to fetch all posts posted by the user identified by `X-User-ID`.

```
GET /posts/me
Content-Type: application/json
X-User-ID: <UUID>
```

- The `GET` method on `/posts/{userId}` allows to fetch all posts posted by the user identified by `userId`. Note that `X-User-ID` is unused but required to ensure authentication.

```
GET /posts/{userId}
Content-Type: application/json
X-User-ID: <UUID>
```

  Example of JSON response:

```
[
  {
    "username": "CortezTheKiller",
    "content": "Looking for the new world...",
    "timestamp": "1519-08-20T13:08:41.099474Z"
  },
  {
    "username": "CortezTheKiller",
    "content": "I came dancing across the water",
    "timestamp": "1519-08-20T13:02:46.264428Z"
  }
]
```

- The `POST` method on `/posts/me` allows to create a new post for the user identified by `X-User-ID`.

```
POST /posts/me
Content-Type: application/json
X-User-ID: <UUID>
________________________________________
{
  "content": "<text>",
}
```

3.2.3. Feed-service

This microservice creates the feed that would be displayed on the users homepages. A user's feed is made up of the 10 latest posts (at most) of his/her friends. The posts are ordered from newest to oldest.

The feed-service is allowed to query the other services (**via the load-balancers**), it is not backed by any database. There is only one instance of the feed-service.

- The `GET` method on `/feed` allows to fetch the feed of the user identified by `X-User-ID`. The feed-service will send a request to the user-service load-balancer to fetch the friends of the user, then, for each of his/her friend, it will send a request to the post-service load-balancer to fetch the respective posts.

```
GET /feed
Content-Type: application/json
X-User-ID: <UUID>
```

  Example of JSON response:

```
[
  {
    "username": "The Bernal Díaz del Castillo",
    "content": "Everything's going well!",
    "timestamp": "1520-05-20T13:03:56.846895Z"
  },
  {
    "username": "CortezTheKiller",
    "content": "Looking for the new world...",
    "timestamp": "1519-08-20T13:08:41.099474Z"
  },
  {
    "username": "CortezTheKiller",
    "content": "I came dancing across the water",
    "timestamp": "1519-08-20T13:02:46.264428Z"
  }
]
```

3.3. Load-Balancers

An important aspect of microservices architecture is high availability and scalability for each microservice. To achieve this, a load-balancing layer is placed between the gateway and the service instances (called backends). In the architecture, there is a *user-service load-balancer* and a *post-service load-balancer*, each having 2 backends. The **feed-service does not use a load-balancer** because it only has one instance.

The load-balancers **must operate at layer 4** of the OSI model. They only need to forward TCP connections, **UDP support does not need to be implemented**.

The features of the load-balancers are the following:

- Distributes TCP connections transparently.
- Uses a simple `config.yaml` file for configuration. The following parameters must be supported:
  - `port`: The port on which the load-balancer listens.
  - `algorithm`: The algorithm used for distributing connections.
  - `backends`: A list of backend instance URLs (IP address + port).
  - `rate`: The maximum transfer rate (in MB/s) allowed per connection.
- Supports the `roundrobin`, `leastconn` and `hashing` algorithms.
  - The `roundrobin` algorithm selects each backend one after the other.
  - The `leastconn` algorithm always selects the backend that has the lowest number of connections.
  - The `hashing` algorithm uses the IP address of the client to distribute the load.
- Includes a **healthcheck** functionality.
  - The load-balancer periodically checks if its backends are alive. If a backend does not respond, it will be considered down and removed from the backend pool. This can be achieved by using a parallel job (or subprocess) that pings each backend at fixed intervals.
- Includes a **rate limiting** functionality.
  - Each connection has an upper bound on the data transfer rate. This helps prevent service abuse by misbehaving clients. A good value for the `rate` parameter is 100 MB/s.

3.4. Locust and Prometheus

3.4.1. Locust

Locust[4] is a load testing tool based on Python. We ask you to implement a complete locustfile (named `locustfile.py`) testing **all endpoints** of the platform's API.

You should not add a Docker container for Locust. It must be used directly on your host.

3.4.2. Prometheus

We ask you to add a Prometheus container in the architecture. You will configure Prometheus to scrape the `/metrics` endpoint of the gateway and include the 3 following recording rules[5]:

- Record the total number of requests per method (`GET`, `POST`, `DELETE`).
- Record the request rate per second, averaged over the last minute on `/api/posts` endpoint with `POST` method.
- Record the average latency per endpoint over the last minute.

4. Deliverables

This project must be implemented in either Go and/or Python, but we strongly recommend Go, as it is well suited for network programming. You can, if you want, use a mix of the two languages for different components.

For the gateway database, we recommend you to use a PostgreSQL container, but you can choose a different option if you prefer.

[4]See https://docs.locust.io/en/stable/writing-a-locustfile.html

[5]See https://prometheus.io/docs/prometheus/latest/configuration/recording_rules/

4.1. Source Code

- Complete Go and/or Python implementations of the gateway, load-balancer, and feed-service.
  - Including Dockerfiles for each component to support deployment with `docker compose`.
  - Do not upload your own images on Docker Hub.

4.2. Docker Compose configuration

- A `docker-compose.yaml` file orchestrating all components, including a Prometheus container.

4.3. Monitoring Setup

- A `locustfile.py`, able to load test the platform's API.
- Prometheus configuration files, including the 3 recording rules.

4.4. Documentation

- A `README.md` file **briefly** documenting your project, it must include:
  - An explanation about the folders and files structure.
  - How to build, start, and stop the system (should be based on `docker compose`).
  - How to test the platform using Locust and Prometheus.
- Also, remember to add inline documentation within your code *when necessary* to ensure easy understanding.

4.5. Optional Deliverables

- SSL certificates (self-signed) for HTTPS support.
- Anything needed to make your project work.

5. Project Guidelines

- **Installing Docker:** If you run Linux, you should be able to install Docker using your package manager (`apt`, `dnf`, etc.). If you *unfortunately* run MacOS or Windows, the easiest approach is to install Docker Dekstop. For Windows, we also recommend using WSL.
- Become familiar with both `docker` and `docker compose` commands along with the `docker-compose.yaml` syntax.
- **Stay organized**: Do not duplicate the same codebase. Instead, make use of the `build` and `environment` elements of `docker-compose.yaml` to specialize the application.
  - You should make use of the Docker interpolation mechanism to ensure a flexible configuration.
  - To simplify addressing between containers in Docker, you should use a **bridge network** driver. This will allow automatic DNS resolution between all containers. You will then be able to use a container's name as a valid IP address.
- Try to use as many Go/Python libraries as you need. Do not implement everything from scratch! For example, we expect you to use an external library for handling the JWT logic.
- Begin with a small and functional version of each component, then gradually expand.
- You have some degree of freedom, so try to make the best choices possible.
  - For example, you are free to select the port number for each component (but some are better than others).
- Good luck, and try to have fun!

6. Submission and Grading

The submission of your assignment is subject to the following rules:

1. You must submit an archive named **`FIRSTNAME_LASTNAME.zip`** on the submission platform.
2. The archive must contain every file needed to run the topology in Docker.
3. The deadline is November 14 2025, 23:59 PM (hard deadline).

This assignment will count for **20% of the final grade**.