MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA
FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DISTRIBUTED SYSTEMS

Assignment 2

Load Balancing
Reverse Proxy

Prof. Tudor Cioara
As. Liana Toderean
As. Gabriel Antonesi

S.l. Marcel Antal
As. Alexandru Rancea

Conf. Cristina Pop
As. Dan Mitrea

2024-2025

1. Introduction

As web applications grow in complexity, they face challenges in ensuring reliability, performance, and security. By distributing workloads across multiple servers and routing traffic efficiently, distributed systems can boost scalability, reduce response times, and improve fault tolerance. Load balancers and reverse proxies are essential components that help manage traffic distribution and application performance.

2. What is a reverse proxy?

A **reverse proxy** is a server that sits between client devices and backend servers, intercepting requests from clients and directing them to the appropriate server. Here's how a reverse proxy may improve your setup:

- **Caching**: reverse proxies can store copies of frequently accessed content, reducing load on backend servers and speeding up response times for clients;
- **SSL Termination**: they handle encryption and decryption of SSL/TLS traffic, offloading this computational work from backend servers;
- **Enhanced Security**: by hiding the IP addresses of backend servers, a reverse proxy adds a layer of security, protecting your infrastructure from direct exposure to clients.

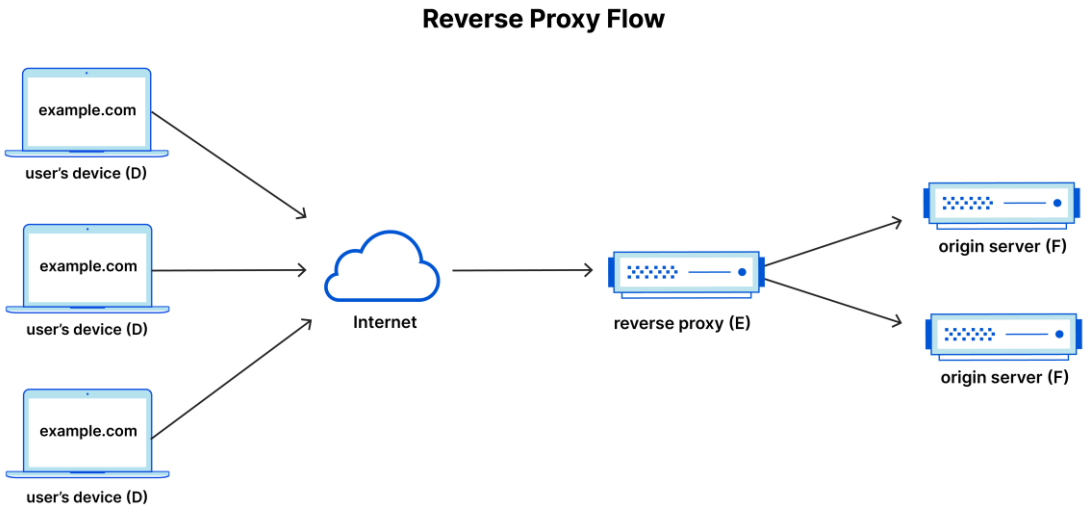


Figure 1 Reverse Proxy Flow

3. What is the role of a load balancer?

A **load balancer** distributes network traffic across multiple servers to avoid overloading any single server, ensuring that applications remain responsive even under high traffic conditions. Benefits of load balancing include:

- **Improved Performance**: traffic is routed to servers with available resources, optimizing the response times;
- **High Availability**: by distributing traffic across multiple servers, load balancers ensure redundancy, meaning that if one server fails, traffic is rerouted to other servers, minimizing downtime;
- **Scalability**: load balancers allow you to add or remove servers easily, adapting to traffic changes and to network constraints.

Load Balancing Algorithms

Load balancers use different algorithms to distribute requests:

- **Round Robin**: requests are routed to each server in turn (most common approach);
- **Least Connections**: traffic is directed to the server with the fewest active connections;
- **Least Response Time**: traffic will be directed to the fastest and least busy server;
- **IP Hashing**: requests are distributed based on client IP, ensuring that a specific client consistently reaches the same server;

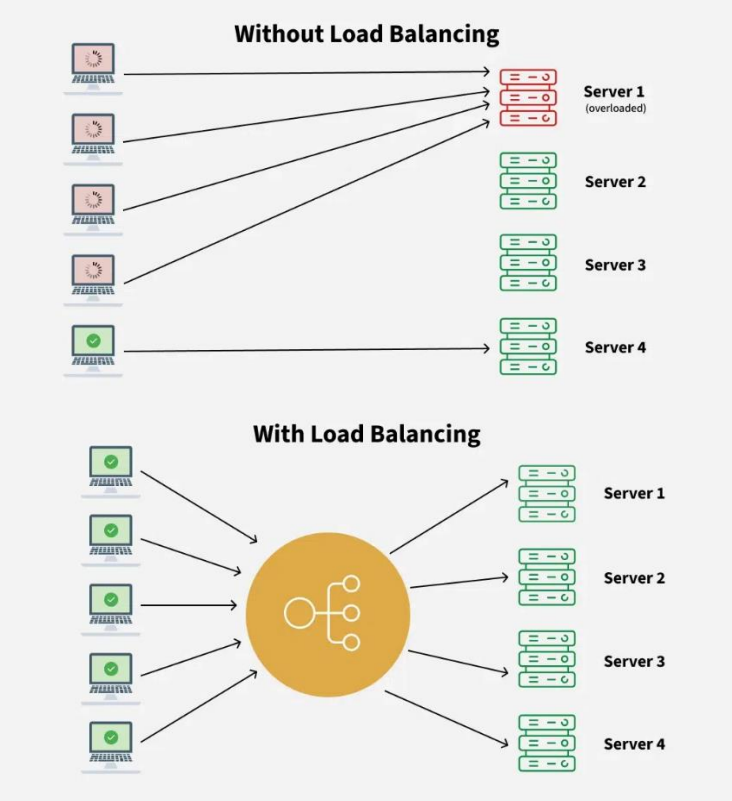


Figure 2 Traffic within a system with and without load balancing

4. Introducing Traefik

Traefik is an open-source Application Proxy that helps users in publishing their services. It receives requests on behalf of the system and identifies which components are responsible for handling them, and routes them securely. It is suitable for adding both reverse proxying and load balancing into your infrastructure.

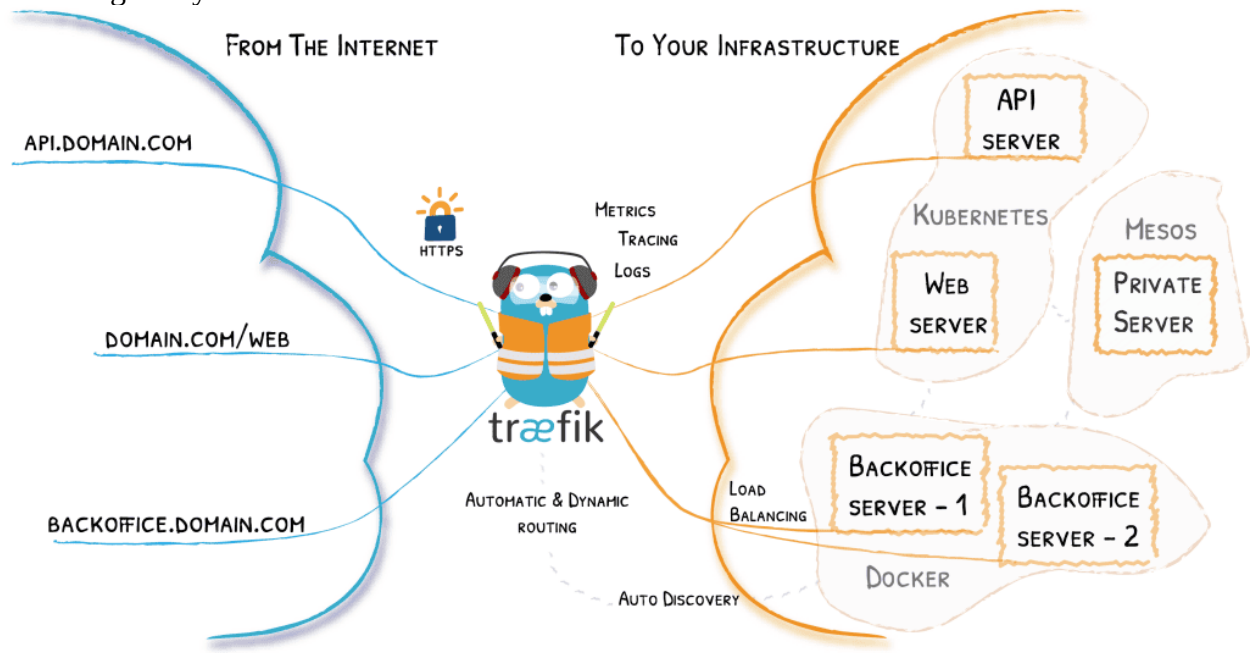


Figure 3 Traefik overview

4.1 How it works

The Traefik Architecture is designed to handle incoming requests efficiently, directing them through a series of well-defined components: **Entry Points**, **Routers**, and **Services**. Each component plays a distinct role in ensuring that requests reach the correct backend services smoothly.

When a request first enters Traefik, it encounters the **Entry Points**. These are the points where Traefik listens for incoming traffic, typically configured by specifying ports or protocols (for example, HTTP on port **80** or HTTPS on port **443**). By defining one or more entry points, you control where Traefik listens, allowing it to filter incoming traffic based on port and/or protocol.

After passing through the entry points, the request moves to the **Routers**. Routers are responsible for connecting incoming requests to the appropriate backend services. They examine key aspects of the request, such as the **host** (e.g., the domain name), **path** (like /login or /api), and other details like headers to determine the destination. Routers use **rules** to decide how to route

each request; for instance, a rule might state, "if the request is for ***api.myapp.com***, direct it to the ***API service***." Additionally, Routers can apply **middlewares**—optional components that modify the request or response. Common middlewares might add authentication, enforce rate limiting, or adjust headers.

Once a router has matched the request to a destination, the request reaches the **Services layer**. Services define **where** Traefik should send the request after routing, specifying backend servers, load balancers, or other destinations. They determine the specific server or service instance that will handle the request. When **multiple** instances of a service are available, Traefik can distribute the load across these instances using load balancing, ensuring efficient handling of high traffic and reducing the likelihood of any single instance being overloaded.

Finally, the request arrives at its destination—whether this is a backend server, service, or application—completing its journey through the Traefik architecture. The request flow moves systematically from Entry Points, through Routers, to Services, with optional middleware adjustments along the way. This architectural design allows Traefik to manage traffic efficiently, improve security, and balance load across multiple service instances, providing a reliable and scalable setup for your applications.

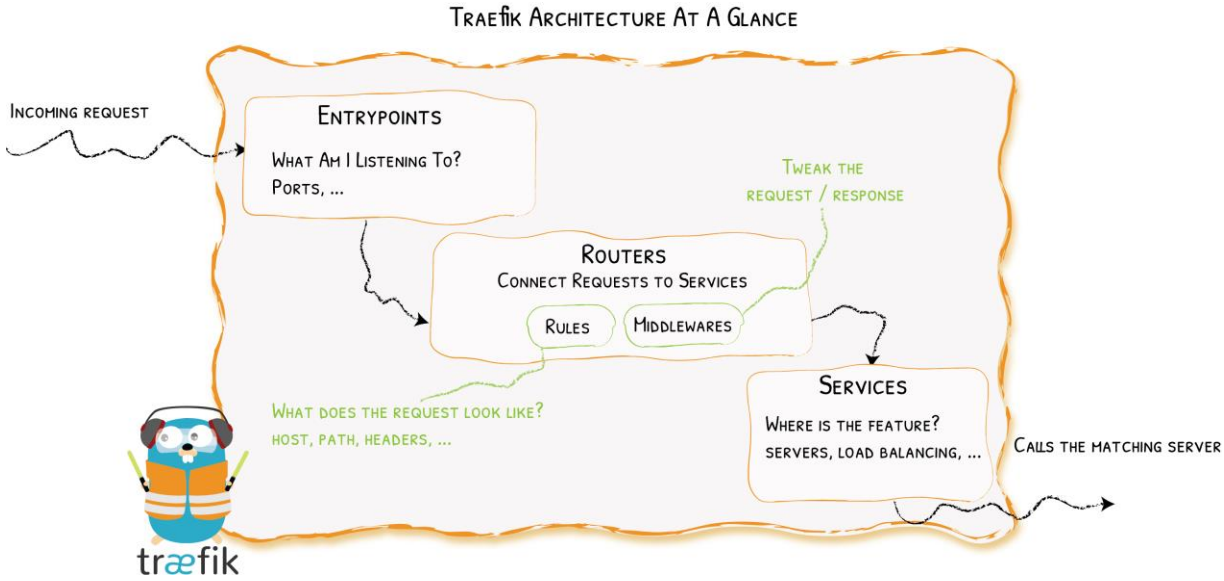


Figure 4 Traefik architecture

5. Traefik and Docker - example

Integrating Traefik with Docker enables dynamic and efficient management of your containerized services. Traefik automatically discovers Docker containers and configures routing, simplifying the deployment and scaling of applications. In this setup, the *reverse-proxy* service utilizes the *traefik:v3.2* image to function as the central routing component for proxying and load balancing.

```yaml
version: '3.8'

networks:
  shared-network:
    driver: bridge

services:
  reverse-proxy:
    image: traefik:v3.2
    command:
      - --api.insecure=true
      - --providers.docker
      - --accesslog=true
      - --accesslog.filepath=/var/log/traefik/access.log
      - --log.level=DEBUG
      - --entrypoints.web.address=:80
    ports:
      - "80:80"
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - traefik_logs:/var/log/traefik
    networks:
      - shared-network
```

Figure 5 Traefik configuration example

Command Arguments:

Argument	Description
--api.insecure=true	Activates the Traefik dashboard and API without authentication. **Caution:** *This setting is insecure and should only be used in development environments lik your assignments! Do not use it in production!*

--providers.docker	Enables the Docker provider, allowing Traefik to monitor Docker events and update its configuration dynamically
--accesslog=true	Enables access logging, facilitating monitoring and debugging
--accesslog.filepath=/var/log/traefik/access.log	Specifies the file path for storing access logs
--log.level=DEBUG	Sets the logging level to DEBUG, providing detailed logs for troubleshooting
--entrypoints.web.address=:80	Defines an entry point named 'web' that listens on port 80 for incoming HTTP traffic

Ports:

- **80:80**: maps port 80 of the host to port 80 of the container, allowing external HTTP traffic to reach Traefik;
- **8080:8080**: maps port 8080 of the host to port 8080 of the container, providing access to the Traefik dashboard – useful to monitor if Traefik correctly discovers all docker containers used in your configuration.

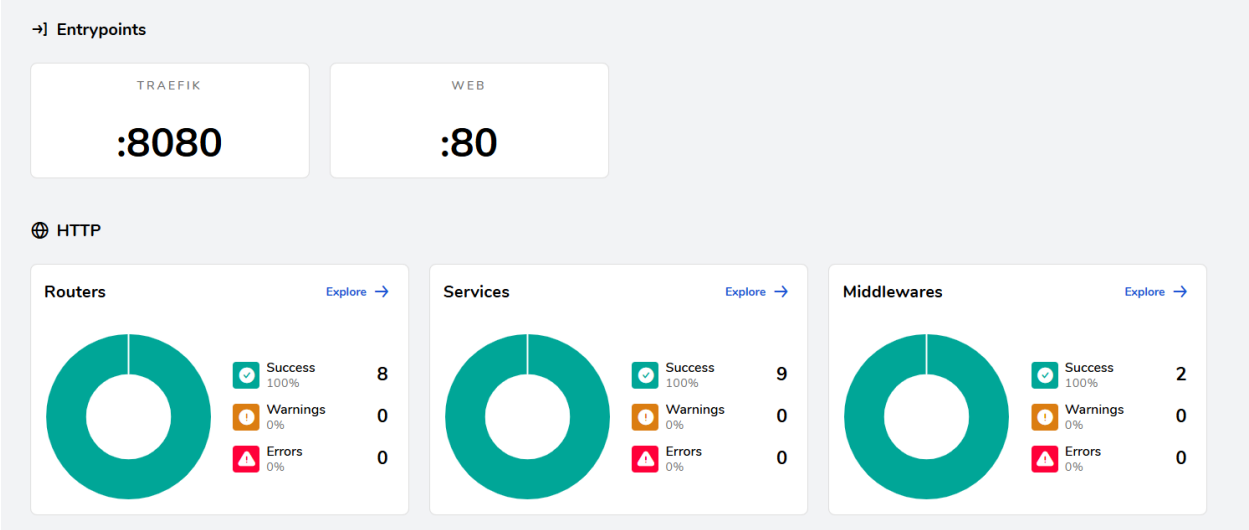


Figure 6 Screenshot taken from the Traefik Dashboard, running on the exposed port 8080

Volumes:

- **/var/run/docker.sock:/var/run/docker.sock:** mounting the Docker socket into the Traefik container allows Traefik to interact directly with the Docker daemon. This setup enables Traefik to automatically detect and configure routes for your Docker containers as they start and stop, simplifying the management of your services;
- **traefik_logs:/var/log/traefik**: mounts a volume for storing Traefik's logs, ensuring persistence and accessibility.

Networks:

- **shared-network**: connects the Traefik container to the shared-network, facilitating communication with other services on the same network – OPTIONAL, but better for network isolation and control over network settings.

5.1. Using Traefik to route between multiple services

The following Docker Compose service configuration defines a containerized Spring Boot application called ***app1***. As you already know from the lab, each part of this setup plays a specific role, from building the application image, all the way to setting up Traefik as a reverse proxy. We will be focusing on the extra configurations, that were not covered for the previous assignment, namely *deploy* and *labels* sections. The ***deploy*** section in a Docker Compose file specifies deployment configurations for services. The **replicas** setting within deploy defines the number of instances (or replicas) of the service that Docker should run. For example, setting replicas: 4 in the deploy section means that Docker will launch four identical containers for this service, allowing for load balancing and higher availability. Traefik can distribute incoming requests across these replicas, which is especially useful for handling high traffic or ensuring redundancy in case a replica fails. The ***labels*** section is used to enable and configure Traefik specifically for that service. Let's take a closer look.

```
app1:
  build:
    context: ./producer
    dockerfile: Dockerfile
  depends_on:
    - db
  deploy:
    replicas: 4
  environment:
    SPRING_DATASOURCE_URL: jdbc:mysql://db:3306/users_db
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: root
  networks:
    - shared-network
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.app1.rule=Host(`app1.localhost`)"
    - "traefik.http.services.app1.loadbalancer.server.port=8081"
```

Figure 7 Docker Compose service configuration and Traefik labels for routing

Step	Label Format	Example
Enable Traefik for Service	*traefik.enable=true*	*traefik.enable=true*
Define a Router with a Rule	*traefik.http.routers.<router_name>.rule=Host(<domain_name>)*	*traefik.http.routers.app1.rule=Host(app1.localhost)*
Specify Service's Port	*traefik.http.services.<service_name>.loadbalancer.server.port=<internal_port>*	*traefik.http.services.app1.loadbalancer.server.port=8081*

Here's how each label works:

1. **Enable Traefik for the Service**:
   adding traefik.enable=true to the service's labels tells Traefik to manage routing for this particular service. Without this label, Traefik will ignore the service and not route any traffic to it.
2. **Define a Router with a Rule**:
   the label traefik.http.routers.<router_name>.rule=Host(<domain_name>) sets up a routing rule. Here:
   - <router_name> is a unique identifier for the router, which you can choose (e.g., app1);
   - <domain_name> specifies the hostname that should be routed to this service (e.g., app1.localhost).

   This **rule** ensures that any request with the specified hostname (like **app1.localhost**) will be directed to the service associated with this router.
3. **Specify the Service's Internal Port**:
   The label traefik.http.services.<service_name>.loadbalancer.server.port=<internal_port> tells Traefik the port on which the service listens inside the container. Here:
   - <service_name> should match the name of the service (e.g., app1);
   - <internal_port> is the port that Traefik routes traffic to within the container (e.g., 8081).

These labels allow Traefik to dynamically route requests based on the hostname and forward them to the correct service on the specified internal port, providing an efficient setup for handling traffic across services in Docker.

When configuring Traefik for multiple services, you can reuse the **same label structure** by simply adjusting the service-specific names. The setup for each service will look almost identical, but you'll change the names of routers, services, and internal ports as needed. This approach keeps your configuration consistent and easy to maintain, while allowing Traefik to handle each service separately.

Alternative for running multiple instances of the same container

If you don't want to specify replicas in the Compose file, you can scale services dynamically using the ***--scale*** flag when you run the *docker-compose up* command. This is a quick way to adjust the number of replicas from the command line without modifying the Compose file

directly. This approach is useful when testing different replica counts on the fly or when you need flexibility in scaling services based on real-time needs.

Example command: *docker-compose up --scale app1=4*

Bonus

In Traefik, you can use „*middlewares"* to manipulate request paths before they reach your services. One useful middleware is StripPrefix, which removes specified prefixes from incoming requests. This can be helpful in cases where your services don't expect certain path structures. Here's how to set it.

To remove unwanted path prefixes from incoming requests, we start by defining the middleware that performs this action. The label *traefik.http.middlewares.<service_name>-strip-prefix.stripprefix.prefixes=/api* creates a middleware named *<service_name>-strip-prefix* that removes the /api prefix from incoming requests. For example, you could use *traefik.http.middlewares.app1-strip-prefix.stripprefix.prefixes=/api* to define this middleware for a service called app1.

Once the middleware is defined, we attach it to the service's router using the label *traefik.http.routers.<service_name>.middlewares=<service_name>-strip-prefix*. By linking this middleware to the router, we ensure that all incoming requests with the /api prefix are automatically adjusted before they reach the service. This setup is useful when the service itself doesn't expect or recognize the /api prefix. For instance, if your application expects requests to start at /users (e.g., **/users/123**) but you want external requests to go to **/api/users/123**, this middleware will strip out **/api**, so the service only sees **/users/123**.

This approach is particularly helpful when you have multiple services organized under a common API structure (like /api) but each service requires specific paths without that prefix. By using StripPrefix in this way, Traefik ensures that incoming requests are routed in a format compatible with each service's internal path structure, keeping configurations consistent and requests clean.

Request routing between Docker containers

The connection between two Docker containers (e.g. from user microservice to device microservice) through Traefik can be established by defining a PathPrafix rule in the router configuration of the target container.

However, it is important to note that while Traefik matches requests based on the specified path prefix, it does not automatically strip the prefix from the forwarded request. This means the backend service will receive the full path (including the prefix), which might not match the expected routing in the backend application.

To address this, a middleware needs to be added to explicitly strip the prefix from the request path before forwarding it to the target container.

Here is an example of how to configure this:

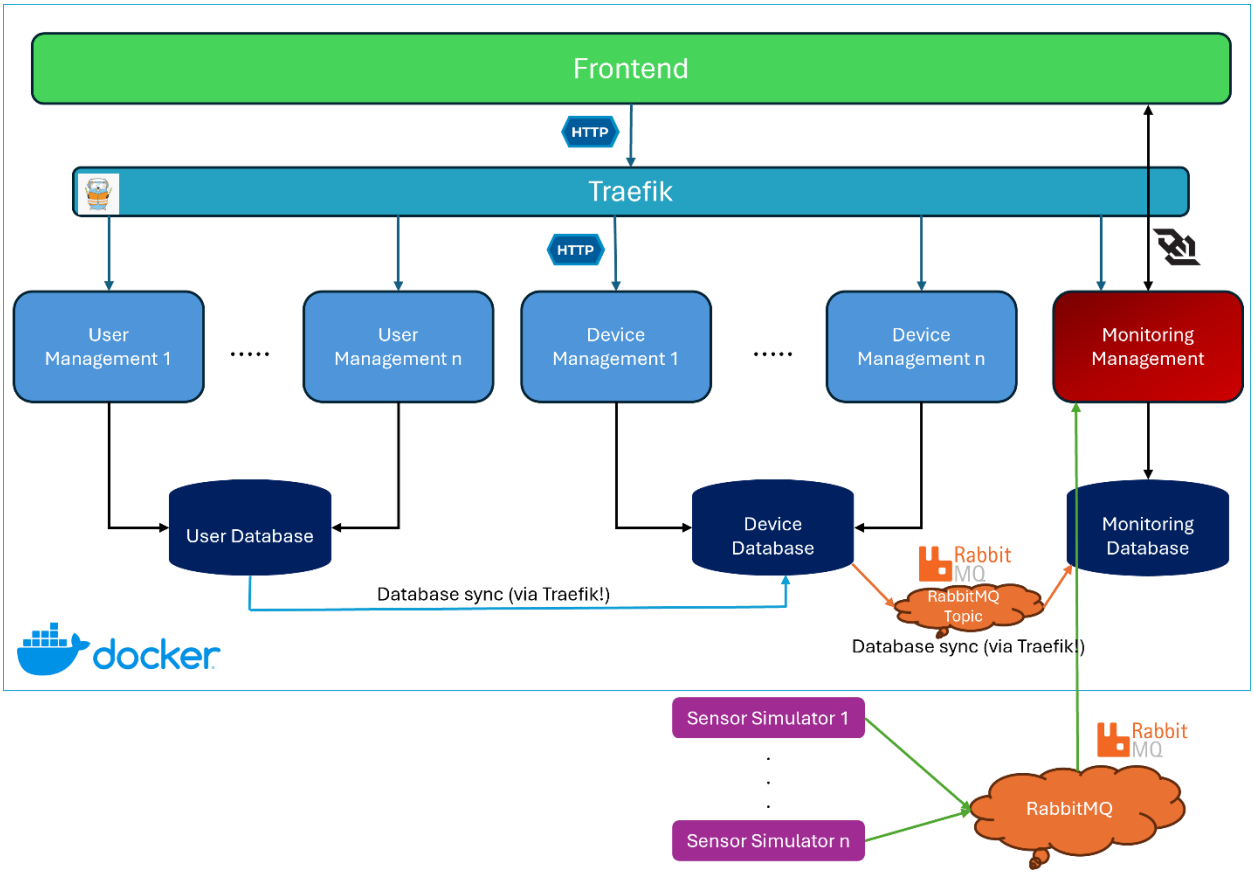
```
app1:
  build:
    context: ./producer
    dockerfile: Dockerfile
  depends_on:
    - db
  deploy:
    replicas: 4
  environment:
    SPRING_DATASOURCE_URL: jdbc:mysql://db:3306/users_db
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: root
  networks:
    - shared-network
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.app1.rule=Host(`app1.localhost`) || PathPrefix(`/app1`)"
    - "traefik.http.middlewares.app1-strip.stripprefix.prefixes=/app1"
    - "traefik.http.routers.app1.middlewares=app1-strip "
    - "traefik.http.services.app1.loadbalancer.server.port=8081"
```

In this example, it is shown how to configure a PathPrefix rule and attach a middleware to strip the prefix from the incoming request before it is forwarded to the target container. This ensures that the backend service receives the request path in the expected format.

Requests from another Docker container can be made to the Traefik container (named reverse-proxy) using the specified PathPrefix. Traefik will then process these requests according to the defined routing rules and forward them to the appropriate target container after applying the middleware.

E.g.: A request to http://reverse-proxy/app1/... from another Docker container within the same network, will be routed by Traefik to the app1 container. Traefik will strip the /app1 path prefix (using the configured middleware).

6. System Architecture Blueprint



7. Bibliography

1. https://doc.traefik.io/traefik/ Traefik Official Documentation
2. https://dev.to/karvounis/basic-traefik-configuration-tutorial-593m - Basic Traefik configuration tutorial
3. https://www.ibm.com/topics/load-balancing - What is load balancing?
4. https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/ - What is a reverse proxy?