Student Information System with API Gateway Architecture

Cloud Computing – Final Project

Cloud Computing Technologies
Spring 2024–2025

Student: Yusuf Kemahlı
Instructors: Prof. Marco Anisetti, Prof. Claudio A. Ardagna
Date: July 14, 2025

Contents

Chapter 1

Introduction

This project was developed as the final assignment for the course *Cloud Computing Technologies*, aiming to apply core concepts learned during the semester in a hands-on, practical way.

The goal of this project was to design and build a simple Student Information System using a microservice architecture supported by an API Gateway.

From the beginning, I aimed to not only get the basic functionality working,like user login and retrieving student grades,but also to explore how API Gateways can help manage important non-functional aspects such as authentication, scalability, and monitoring.

To achieve this, I decided to use Kong as the API Gateway, Flask for lightweight Python-based microservices, and Docker Compose for orchestration. Throughout the development, I tried to keep things modular and clean, ensuring each service had a single responsibility and all configuration was reproducible.

In the following sections, I describe the system's architecture, how each component works, and the advantages this approach brings over a more traditional or monolithic design.

Technologies used in this project:

- **Kong API Gateway**
- **Docker & Docker Compose**
- **Python Flask (microservices)**
- **JWT (JSON Web Tokens)**
- **Prometheus** (monitoring)
- **Postman** (API testing)

Chapter 2

System Architecture

The system is built around a microservice architecture, with each service running in its own container and communicating through a shared Docker network. Kong acts as the API Gateway, sitting between the client and the services, and handling all external traffic.

There are three main services in the system:

- **login-service** – responsible for authenticating users and generating JWT tokens
- **student-service** – serves student grade data, protected by token-based authentication
- **student-service-2** – an identical second instance to allow load balancing

These services are deployed using Docker Compose, along with a PostgreSQL container to store Kong's configuration.

I also included some extra components to support observability and scalability:

- **JWT plugin** – verifies access tokens on protected endpoints
- **Prometheus** – collects metrics from Kong (like request count, status codes, etc.)
- **file-log plugin** – logs each request made to protected services
- **rate-limiting plugin** – limits the number of requests a user can make per minute

All components are defined in a single `docker-compose.yml` file, which makes it easy to spin up or shut down the system with one command. The Kong configuration is also scripted using a shell file, which automatically registers services, routes, plugins, and consumers when the system starts.

The architecture was intentionally kept simple and easy to understand, while still including core features like authentication, load balancing, logging, and monitoring.

Chapter 3

Implementation Details

login-service

The `login-service` is responsible for authenticating users and generating JSON Web Tokens (JWT). Users send their `student_id`, `email`, and `password` in the login request. If the credentials are valid, the service returns a JWT that includes the student's ID, name, and email.

To keep things simple and focus on the architecture, I used a static dictionary as the "user database". Obviously, in a real application, this would be replaced with a proper database.

Each token also includes an expiration time and is signed with a secret using the HS256 algorithm. The `JWT_ISSUER` and `JWT_SECRET` values are injected through environment variables, which are automatically updated by the setup script during deployment.

student-service (and student-service-2)

This service is protected by Kong's JWT plugin. When a request is made, the token is extracted from the Authorization header and verified using the same secret and issuer from the login service.

If the token is valid, the service reads the `student_id` from the payload and looks up that student's grades from a predefined dictionary. The response includes the student's name, ID, a dictionary of grades, and a string identifying which instance served the request (e.g., "Instance 1" or "Instance 2"). This makes it easier to test load balancing.

Having two identical instances of this service allows Kong to distribute requests between them using a round-robin strategy, which simulates how real-world services handle scaling.

Containerization and Deployment

The entire system is containerized using Docker. Each service (login, student, Kong, database) runs in a separate container. This setup allows the system to be launched, scaled, or rebuilt consistently across different environments.

A single `docker-compose.yml` file defines and orchestrates all the containers, including network settings, exposed ports, environment variables, and volume bindings. With a single command (`docker compose up --build`), the full system can be started from scratch.

This level of automation is particularly important when using an API Gateway, as it allows us to reproduce consistent routing, authentication, and plugin setups every time.

Kong Configuration (setup_kong.sh)

To configure Kong, I wrote a shell script that defines everything programmatically:

- Registers the services and their routes
- Creates an upstream with two targets for the student service
- Adds the required plugins:
  - **jwt** – secures the student-service endpoints
  - **rate-limiting** – restricts requests to 5 per minute per consumer
  - **file-log** – writes logs to `/tmp/kong_logs.log`
  - **prometheus** – exposes system metrics for monitoring
- Creates a consumer (named “studentPortal”) and generates a JWT key pair
- Writes the key and secret into `.env` files for all services

In addition to this, I also used `deck` to export the full Kong configuration into a YAML file. This file makes it easy to version, store, and reapply the configuration in different environments or teams, supporting integration and repeatability.

Logging, Monitoring, and Testing

To test the system end-to-end, I used Postman to send login requests and access the protected grades endpoint using the returned JWT.

For observability, I developed two custom scripts:

- **log_monitor.sh** – tails the Kong access logs from the container in real time
- **prometheus_monitor.sh** – fetches selected Prometheus metrics (like request counts and HTTP status codes)

These scripts make it easy to monitor what’s happening in the system without entering the containers manually. I could see rate limits being triggered, traffic switching between student-service instances, and overall request statistics directly in the terminal.

Because Kong sits at the center of the system, all logs and metrics are collected in a single point. This greatly simplifies monitoring and helps avoid duplication across services.

Chapter 4

Advantages of the API Gateway Architecture

Using an API Gateway like Kong brought several clear advantages to this project, both in terms of functionality and system quality.

1. Centralized authentication and access control. Instead of writing JWT verification logic inside every service, I used Kong's built-in JWT plugin to handle all token validation. This reduced code duplication and made the system easier to maintain and secure.

2. Easy load balancing. By creating an upstream and defining multiple targets, Kong automatically balanced requests between two instances of the student-service. This allowed me to simulate scalability without changing any service code.

3. Built-in rate limiting. Thanks to the rate-limiting plugin, I could control how many requests a user can make in a given time window. Without Kong, I would have to implement this manually in each service.

4. Centralized logging and monitoring. All requests are logged and measured at the gateway level. This makes it much easier to observe the system's behavior using a single point of control. Instead of adding Prometheus and logging logic into every microservice, I used Kong's plugins to expose metrics and write access logs.

5. Easy configuration and redeployment. The entire system,including Kong's configuration,is reproducible using two files: `docker-compose.yml` and `setup_kong.sh`. I also used `deck` to export the full configuration to a YAML file, making it easy to share or redeploy the system in another environment.

Without an API Gateway, I would need to implement security, monitoring, rate limiting, and service connection logic separately in every microservice. This would not only be inefficient but also error-prone and hard to scale.

Using Kong allowed me to keep each microservice focused only on its core logic and delegate intersecting concerns to a single, consistent layer.

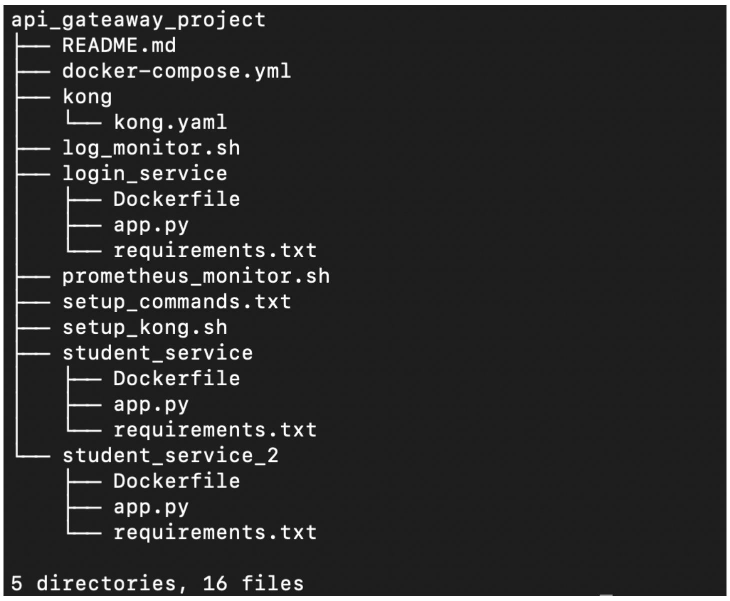
Chapter 5

Conclusion

This project helped me understand how an API Gateway can simplify and strengthen a microservice architecture. By centralizing common concerns like authentication, logging, rate limiting, and monitoring, I was able to keep each service small, clean, and focused.

The use of Kong, along with Docker Compose and automation scripts, made the system easy to deploy and manage. I also gained hands-on experience with containerization, token-based security, service orchestration, and observability tools like Prometheus.

Overall, the final system is modular, scalable, and reproducible. It reflects many of the core principles we studied during the course, and could be extended further with persistent storage or a frontend interface if needed.



```
api_gateaway_project
├── README.md
├── docker-compose.yml
├── kong
│   └── kong.yaml
├── log_monitor.sh
├── login_service
│   ├── Dockerfile
│   ├── app.py
│   └── requirements.txt
├── prometheus_monitor.sh
├── setup_commands.txt
├── setup_kong.sh
├── student_service
│   ├── Dockerfile
│   ├── app.py
│   └── requirements.txt
└── student_service_2
    ├── Dockerfile
    ├── app.py
    └── requirements.txt

5 directories, 16 files
```

Figure 5.1: Project folder structure with services and configuration files