# API Gateway

## Introduction:

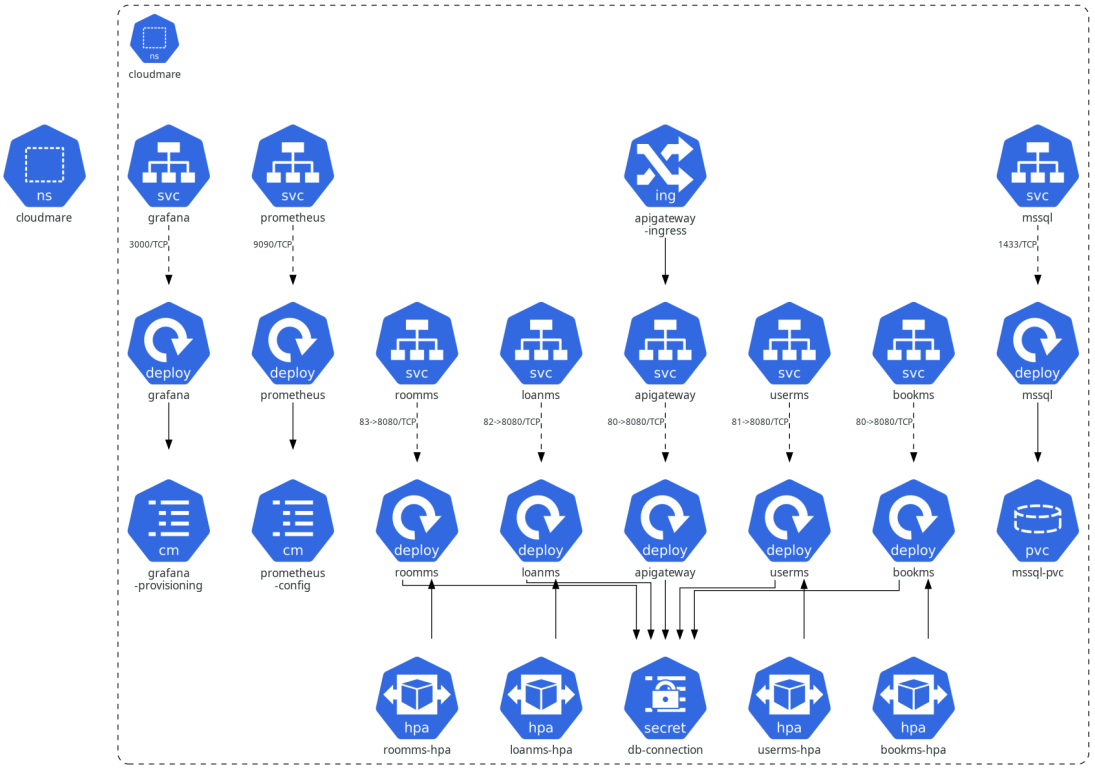
I developed an API Gateway to manage communication between multiple microservices with the following Tech Stack:

- **.NET 8.0** for gateway and microservices
- **DockerHub** for hosting container images (here)
- **MSSQL** as a shared relational database
- **Kubernetes** and **Minikube** for (local) deployment and orchestration
- **Angular** for a simple fronted (used to test)
- **Prometheus** and **Grafana** for observability and monitoring

I chose this stack because I already had some experience with these tools, and I think that they also are reliable and stable for building cloud native systems.

## System architecture:

A system architecture diagram built by the manifest file is provided (thanks to Philippe Merle):



## Overview of the architecture

The system runs on *Kubernetes* (apart from the frontend) using *Minikube*, creating a microservices architecture that should be both resilient and scalable.

**The Gateway Layer** acts as the single entry point that clients interact with, handling all the routing decisions to direct requests to the right microservices. It takes care of user authentication through *JWT tokens*, implements rate limiting to prevent system overload, and includes fault tolerance mechanisms using *Polly* libraries. The gateway gets external access through an *NGINX ingress controller* that also handles TLS termination for secure connections.

**Microservices Architecture** forms the core business logic layer. You have four independent services (*Books*, *Users*, *Loans*, and *Rooms*) each running in their own containerized environment. Each microservice has its own *Horizontal Pod Autoscaler* (*HPA*), so they can scale up and down independently.

**Data Persistence** is handled by a centralized *MSSQL* database that uses Kubernetes *persistent volume claims* (*PVC*) to ensure data survives pod restarts and includes proper service discovery so all microservices can reliably connect to it. Each service connects securely using Kubernetes secrets for database credentials.

**Resilience and Monitoring** are part of the system. The *API Gateway* implements *circuit breaker* patterns, *retry logic* with exponential back-off, and *bulkhead isolation* to ensure one failing service doesn't bring down the entire system. *Prometheus* continuously scrapes metrics from both the gateway and all microservices, while *Grafana* provides real-time dashboards so you can visualize system health, performance trends, and scaling events as they happen.

# The Gateway

Below is an overview of the parameters chosen for this project. They serve as initial guidelines and should be adjusted as the system scales and requirements evolve.

## Resilience with Polly

*Polly* provides resilience patterns that are applied individually to each microservice through named `HttpClient`s using *.NET 8*'s `AddStandardResilienceHandler()` method.

## Retry Policy

```
options.Retry.MaxRetryAttempts = 5;
options.Retry.BackoffType = DelayBackoffType.Exponential;
options.Retry.MaxDelay = TimeSpan.FromSeconds(2.5);
```

The system will attempt failed requests up to 5 times using exponential backoff to give it some time between attempts, capping the maximum delay at 2.5 seconds.

## Circuit Breaker

```
options.CircuitBreaker.FailureRatio = 0.05;
options.CircuitBreaker.MinimumThroughput = 75;
options.CircuitBreaker.SamplingDuration = TimeSpan.FromSeconds(35);
options.CircuitBreaker.BreakDuration = TimeSpan.FromSeconds(6);
```

The Circuit Breaker pattern acts like an electrical circuit breaker, it opens when just 5% of requests fail, but only after seeing at least 75 requests in a 35 second window. Once triggered, it stays open for 6 seconds, giving the downstream service time to recover while failing fast for new requests.

## Bulkhead Isolation

```
Policy.BulkheadAsync<HttpResponseMessage>(20, 50);
```

Each named HttpClient has its own pool of execution slots and queuing slots, ensuring that a slow Books service can't starve the Users service of resources.

## Rate Limiting

The system implements a dual-layer approach using *.NET 8*'s built-in rate limiting middleware. Both layers work together and are applied to all controller endpoints, except for health check endpoints, ensuring that the system remains protected from abuse while still allowing critical monitoring to function uninterrupted.

### Per-IP Policy

Every IP address gets its own allowance to prevent any single source from monopolizing resources:

```
PermitLimit = 400/min
QueueLimit = 100
```

This gives each IP 400 requests per minute with an additional buffer of 100 queued requests for handling short bursts.

### Global Fixed Policy

On top of per-IP limits, the system have a system-wide cap to protect overall system resources:

```
PermitLimit = 800/min
QueueLimit = 225
```

## Authentication

The system use *JWT*-based authentication that validates every aspect of incoming tokens. Only the *API Gateway* needs to handle authentication. Microservices trust the gateway to validate requests, assuming all traffic flows through it. Once a request passes the gateway, microservices can rely on its authentication, creating a clean separation of concerns.

## Health Checks

*Kubernetes* needs to know when our services are healthy, so it exposes standardized health check endpoints:

- `/health/liveness`: confirms the application process is running
- `/health/readiness`: verifies the app is ready to handle requests

# Kubernetes Configuration

> ⚠️ As I was working with limited hardware every component is optimized for a modest laptop environment.

## MSSQL Database

The database layer uses *Kubernetes* best practices for stateful applications:

- **Secret Objects** securely store connection strings away from application code
- **Persistent Volume Claims** ensure data survives pod restarts and redeployments
- **Deployments and Services** provide reliable database access with built-in health monitoring

## Microservice Template

Each microservice follows a consistent deployment pattern that balances resource efficiency with scalability:

Every service gets its own Deployment with carefully tuned CPU requests and limits, plus health check configurations that work with Kubernetes lifecycle management. Services are annotated for Prometheus scraping, and Horizontal Pod Autoscalers monitor CPU usage to make scaling decisions:

```
averageUtilization: 20
minReplicas: 1
maxReplicas: 4
```

This 20% CPU threshold means services scale up quickly under load while the 1-4 replica range respects the hardware constraints.

## API Gateway

The gateway deployment includes environment variables that define how to reach each downstream service:

```
MicroServicesOptions__Room = "http://roomms:83/api" # repeated for each microservice
```

I use a LoadBalancer service type with *NGINX* Ingress because *Minikube* doesn't have native cloud load balancer support. This combination gives us external access through tunneling while *NGINX* provides sophisticated routing, *CORS* handling, and efficient load balancing.

# Observability

## Logging

NLog handles all our logging needs, writing structured logs to `/app/output.log` with timestamps, log levels, and detailed exception information. This gives us a permanent audit trail of system behavior.

### Prometheus Metrics

I've instrumented the entire system for metrics collection using `app.UseHttpMetrics()` and `app.MapMetrics()`. Every service exposes metrics on `/metrics` endpoints with proper Kubernetes annotations for automatic discovery by Prometheus.

### Grafana Integration

Grafana is pre-integrated and ready for creating dashboards that visualize service health, request volumes, response latencies, and error trends. The foundation is there for adding alerting rules and custom business metrics as the system evolves.

## Future Improvements

### Authentication Evolution

The next step is deploying a dedicated authentication service that can handle API key management, role-based access control, and more sophisticated authorization patterns. Even now, some endpoints are restricted to Admin roles, showing the foundation is already there (the health check endpoints are Admin restricted, for example).

### Intelligent Rate Limiting

Moving beyond IP-based limiting to API key-based quotas would enable proper multi-tenant usage tracking. This opens doors for usage analytics, billing integration, and sophisticated abuse detection based on behavioral patterns rather than just request counts.

### Production Readiness

The path to production involves migrating from *Minikube* to managed *Kubernetes* services, implementing centralized log aggregation, and hardening security practices around *CORS* configuration and secret management.

> I attempted to deploy it on Azure but encountered student tier limitations (mostly due to *MSSQL*), but I have also converted my manifest into Terraform file.

## Conclusion

The project aimed to demonstrate how microservice architecture principles can be implemented. The combination of scalability through *HPA*, resilience via *Polly* patterns, security through *JWT* authentication, comprehensive observability, and intelligent traffic control creates a solid foundation.

What should make this architecture valuable is how it mirrors production patterns while remaining approachable for development and learning. Every component from circuit breakers to rate limiting to monitoring tries to follow industry best practices, creating a realistic playground for understanding how these systems work at scale.