

API Gateway: Simple microservice application using the API gateway architecture in order to support advanced non-functional properties

Kaleabe Seifu Desta

July 14, 2025



Contents

1 Project Overview	3
2 Key Objectives	3
3 Architecture: API Gateway as the Entry Point	3
API Gateway flow	5
Technologies Used	5
4 Demonstration of non-functional properties	5
2. Load Balancing	6
3. Authentication Middleware	6
4. Fault Tolerance	7
5 Service Configuration (ServiceConfig):	8
5.1 Backend Microservices	8
5.1.1 User Service (user-service.go)	8
5.1.2 Product Service (product-service.go)	9
5.1.3 Checkout Service (checkout-service.go)	9
6 Conclusion	9
Project Repository:	9

1 Project Overview

This project focuses on a Go-based API Gateway, built with the Echo web framework, that serves as a centralized entry point for routing client HTTP requests to backend microservices (e.g., User, Product, and Checkout Services). It provides essential non-functional features like load balancing, fault tolerance, and security, making it a robust solution for managing distributed microservices. Deployed on port 8080, the gateway intelligently routes requests based on predefined base paths and target URLs, and enforces access control using JWT authentication.

2 Key Objectives

This setup highlights key microservice principles, with the API Gateway playing a crucial part in achieving them:

- **Unified Access:** abstracts the complexity of multiple microservices by providing a single API endpoint for clients
- **Decoupling:** By acting as an intermediary, the gateway allows backend services to remain independent, simplifying their development, deployment, and scaling.
- **Scalability:** distributes traffic across multiple service instances using round-robin load balancing.
- **Reliability:** Ensures fault tolerance through retry logic with exponential backoff for backend service requests.
- **Security:** Protects sensitive routes (e.g., /api/checkout) with JWT-based authentication.

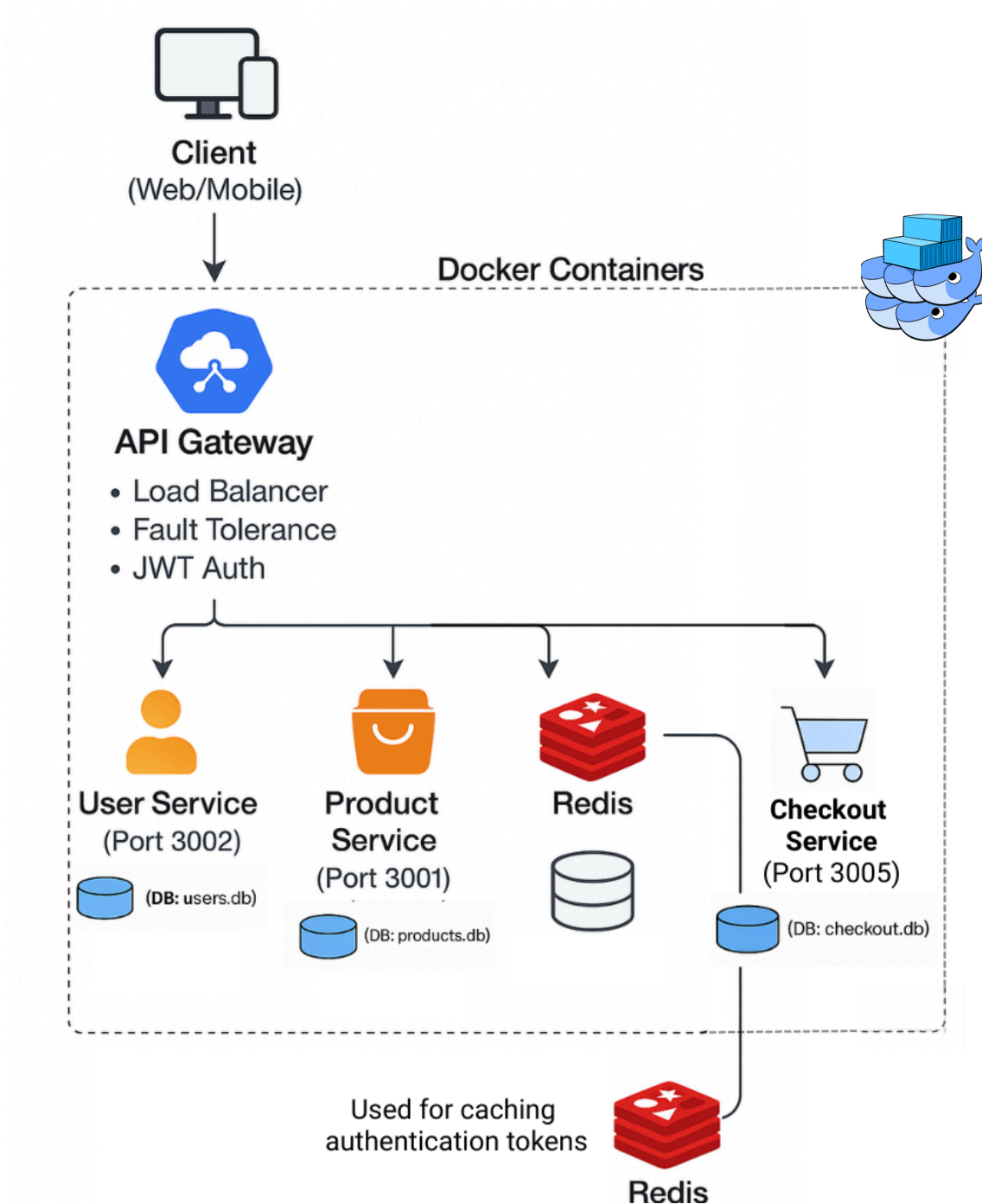
3 Architecture: API Gateway as the Entry Point

The system's design emphasizes the API Gateway as the singular interface for clients.

- **Clients:** Web/mobile apps or third-party services sending HTTP requests to the API Gateway.
- **API Gateway (Dockerized):** The primary focus. It's the intelligent router and orchestrator for external traffic.
 - Runs on port 8080 using Go and Echo framework.
 - Handles routing, load balancing, authentication, and fault tolerance.
 - Routes requests based on path prefixes (e.g., /api/products, /api/users, /api/checkout).
- **Microservices (Dockerized):**
 - **Product Service:** A backend microservice handling product-related operations, also managed and exposed by the API Gateway. Multiple instances (ports

3001, 3003) for product-related requests.

- **User Service:** A backend microservice handling user-related operations, managed and exposed by the API Gateway. Multiple instances (ports 3002, 3004) for user-related requests.
- **Checkout Service:** A backend microservice handling checkout-related operations merging products, Single instance (port 3005) for checkout requests.



API Gateway flow

The client sends a request to the API Gateway on port 8080. Middleware handles logging, CORS, and recovery. For sensitive routes like `/api/checkout`, JWT-based authentication is enforced. The gateway identifies the target microservice based on the request path and uses round-robin load balancing to forward the request to one of its instances. If a request fails, the gateway retries it with exponential backoff (up to 3 times), providing fault tolerance. It then relays the response (headers, status, and body) back to the client, acting as a secure, unified, and resilient entry point for all services.

Technologies Used

- **Go (Golang):** The programming language for all services.
- **Echo Framework:** A high-performance, minimalist Go web framework used for building the HTTP APIs, particularly beneficial for the API Gateway's routing capabilities.
- **SQLite:** A lightweight, file-based database for the backend services.
- **Docker Swarm:** As a container, it orchestrates the API Gateway, multiple instances of product and user services, and the checkout service, enabling seamless communication, scalability, and dependency management within a shared network for the API Gateway project.
- **Redis:** Enhances the API Gateway's performance by caching authentication tokens and data, enabling fast access and validation, particularly for secure routes like `/api/checkout`, as hinted in the code's optional Redis check comment.

4 Demonstration of non-functional properties

- API Gateway (api-gateway.go) - Core Concepts and Implementation

The API Gateway is the orchestrator of this microservice system, providing crucial functionalities beyond simple proxying.

- **Port:** 8080 (default)
- **Framework:** Echo

1. Dynamic Routing:

- Routes incoming requests to appropriate backend microservices based on predefined base paths (e.g., `/api/products`, `/api/users`, `/api/checkout`).
- Uses a `ServiceConfig` struct to define service names, base paths, and target URLs for each microservice.

2. Load Balancing

- Implements a simple round-robin algorithm via a map `currentIndex[service.Name]` to alternate between targets and distribute requests across for each service (product-service, user-service, etc.), multiple target instances are defined.
- Tracks the current target index for each service using a map (`currentIndex`)

```
// Simple round-robin load balancer
currentIndex = make(map[string]int)

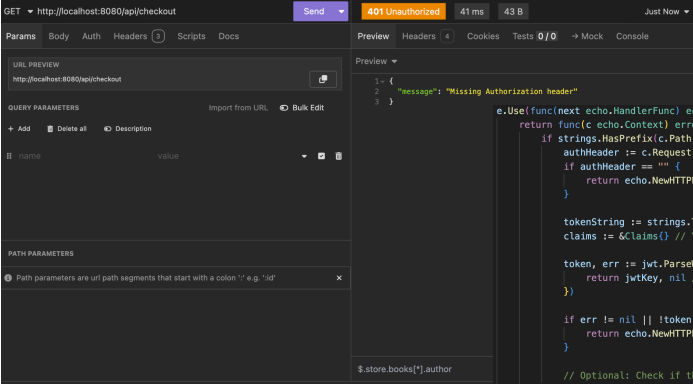
// Simple round-robin load balancing
idx := currentIndex[service.Name]
target := service.Targets[idx]
currentIndex[service.Name] = (idx + 1) % len(service.Targets)
fmt.Println(target)
```

round-robin algorithm to alternate between targets.

```
{"time":"2025-07-15T14:24:52.436695+02:00","id":"","remote_ip":"127.0.0.1","host":"localhost:8080","method":"GET","uri":"/api/products","user_agent":"insomnia/11.2.0","status":502,"error":{"code":502,"message":"Get \"http://product-service-1:3001/\": dial tcp: lookup product-service-1: no such host"},"latency":686875ms,"latency_human":"686875ms","bytes_in":0,"bytes_out":103}
{"time":"2025-07-15T14:24:58.700000+02:00","id":"","remote_ip":"127.0.0.1","host":"localhost:8080","method":"GET","uri":"/api/products","user_agent":"insomnia/11.2.0","status":502,"error":{"code":502,"message":"Get \"http://product-service-2:3003/\": dial tcp: lookup product-service-2: no such host"},"latency":612585291ms,"latency_human":"612.585291ms","bytes_in":0,"bytes_out":103}
```

3. Authentication Middleware

- Secures specific routes (e.g., `/api/checkout`) with JWT-based authentication.
- Validates JWT tokens in the Authorization header, extracting claims (e.g., email) and ensuring token validity.
- Uses Redis to cache valid JWT tokens and store their state, enabling faster validation and support for token revocation (e.g., invalidating a token before expiration).
- Checks Redis for token existence and validity before processing the request, reducing JWT parsing overhead.
- Stores tokens in Redis with an expiration time matching the JWT's `exp` claim to ensure consistency.
- Returns HTTP 401 (Unauthorized) for missing or invalid tokens.



```
e.Use(func(next echo.HandlerFunc) echo.HandlerFunc {
    return func(c echo.Context) error {
        if strings.HasPrefix(c.Path(), "/api/checkout") { // Protect the checkout route
            authHeader := c.Request().Header.Get("Authorization")
            if authHeader == "" {
                return echo.NewHTTPError(http.StatusUnauthorized, "Missing Authorization header")
            }

            tokenString := strings.TrimPrefix(authHeader, "Bearer ")
            claims := &Claims{} // You need the Claims struct here as well

            token, err := jwt.ParseWithClaims(tokenString, claims, func(token *jwt.Token) (interface{}, error) {
                return jwtKey, nil // And the jwtKey
            })

            if err != nil || !token.Valid {
                return echo.NewHTTPError(http.StatusUnauthorized, "Invalid token")
            }

            // Optional: Check if the token is in Redis for an extra layer of security

            return next(c)
        }
        return next(c)
    })
})
```

```
{"time":"2025-07-15T13:55:35.671844+02:00","id":"","remote_ip":"127.0.0.1","host":"localhost:8080","method":"GET","uri":"/api/checkout","user_agent":"insomnia/11.2.0","status":401,"error":{"code":401,"message":"Missing Authorization header"},"latency":45167,"latency_human":"45.167µs","bytes_in":0,"bytes_out":43}
```

4. Fault Tolerance

- Incorporates retry logic with exponential backoff for HTTP requests to backend services, attempting up to three retries with increasing delays.
- Uses a configurable HTTP client with a 5-second timeout to handle unreliable network conditions.
- If the microservice instance is temporarily down or slow to respond, the gateway will **retry** the request up to 3 times before giving up.

```
// Make the request with retry logic for fault tolerance
client := &http.Client{Timeout: 5 * time.Second}
var resp *http.Response
maxRetries := 3

for i := 0; i < maxRetries; i++ {
    resp, err = client.Do(req)
    if err == nil {
        break
    }
    time.Sleep(time.Duration(i+1) * 100 * time.Millisecond) // Exponential backoff
}

if err != nil {
    return echo.NewHTTPError(http.StatusBadGateway, err.Error())
}
defer resp.Body.Close()
```

5. Middleware:

- Utilizes Echo middleware for logging (middleware.Logger), error recovery (middleware.Recover), and CORS support (middleware.CORS).
- Ensures robust request handling and cross-origin compatibility.

6. Request Forwarding:

- Forwards incoming requests to the appropriate backend service, preserving HTTP method, headers, and body.
- Strips the service base path (e.g., /api/products) from the request URL to match the backend service's expected path.
- Copies response headers, status codes, and body from the backend service to the client.

5 Service Configuration (ServiceConfig):

The services array defines the backend microservices that the gateway manages and their routing parameters:

```
services = []ServiceConfig{
{
    Name:      "product-service", // Logical name for the group
    BasePath:  "/api/products",
    Targets:   []string{"http://product-service-1:3001", "http://product-service-2:3003"},
},
{
    Name:      "user-service", // Logical name for the group
    BasePath:  "/api/users",
    Targets:   []string{"http://user-service-1:3002", "http://user-service-2:3004"},
},
{
    Name:      "checkout-service", // Logical name for the checkout service
    BasePath:  "/api/checkout",    // Gateway path for checkout-related requests
    Targets:   []string{"http://checkout-service:3005"}, // Docker Compose service name and port
},
}
```

Endpoints Handled by the Gateway:

The API Gateway routes all requests matching the following patterns:

- ANY /api/users/*
- ANY /api/products/*
- ANY /api/Checkout/*

Example Routing by Gateway:

- A client request to `http://localhost:8080/api/users/123` will be routed by the gateway to `http://localhost:3002/123` (or `http://localhost:3004/123` via load balancing).
- A client request to `http://localhost:8080/api/products/item` will be routed by the gateway to `http://localhost:3001/item` (or `http://localhost:3003/item`). The gateway intelligently trims the `BasePath` before forwarding, so the backend service only sees its relevant sub-path (e.g., `/item`).
- A client request to `http://localhost:8080/api/checkout` will be routed by the gateway to `http://localhost:3005`. The gateway intelligently trims the `BasePath` (`/api/checkout`) before forwarding. Since the checkout service is configured with a single target (`http://checkout-service:3005`) in the `ServiceConfig`, no load balancing is applied, and all requests are forwarded to this instance.

5.1 Backend Microservices

These services contained in a docker swarm operate independently and are exposed to clients only through the API Gateway.

5.1.1 User Service (user-service.go)

Manages user data with standard CRUD operations.

- **Port:** 3002 (default)
- **Database:** SQLite (users.db)
- **Data Model:** User { ID, Name, Email }
- **Initialization:** Connects to users.db, creates the users table, and seeds initial data.
- **API Endpoints:** (Accessed via /api/users from the Gateway)

5.1.2 Product Service (product-service.go)

Manages product data with standard CRUD operations.

- **Port:** 3001 (default)
- **Database:** SQLite (products.db)
- **Data Model:** Product { ID, Name, Price }
- **Initialization:** Connects to products.db, creates the products table, and seeds initial data.
- **API Endpoints:** (Accessed via /api/products from the Gateway)

5.1.3 Checkout Service (checkout-service.go)

Manages checkout operations, processing orders and payments.

- **Port:** 3005 (default)
- **Database:** SQLite (orders.db)
- **Data Model:** Order { ID, UserID, ProductID, Quantity, TotalPrice, Status }
- **Initialization:** Connects to orders.db, creates the orders table, and seeds initial data if needed.
- **API Endpoints:** (Accessed via /api/checkout from the Gateway)

6 Conclusion

The API Gateway, implemented in Go with the Echo framework, serves as an efficient entry point for microservices, providing dynamic routing, round-robin load balancing, JWT-based authentication for secure routes like `/api/checkout` with Redis-backed token caching, and fault tolerance via retry logic with exponential backoff. Compared to alternatives like direct service calls or gateways, this solution, orchestrated with Docker Swarm, offers a lightweight, customizable, and developer-friendly approach for managing containerized microservices. Its key advantages include seamless scalability and high availability through Docker Swarm's orchestration, simplified service deployment, and extensibility for adding new services without code changes, ensuring low-latency performance and robust security with minimal complexity.

Project Repository:

<https://github.com/seifukaleab/API-Gateway-Project2025.git>