Verteilte Systeme und Cloud Technologien - Laborübung 02: Service-Implementierung

Wintersemester 2025
(c) 2025
Simon Kranzer, Gerald Lochner

1. Einführung

Diese zweite Laborübung baut auf der in LB01 erarbeiteten Architektur auf und fokussiert sich auf die konkrete Implementierung eines produktionsreifen Microservice. Der Energy Data Service soll als zentraler Baustein für die Verwaltung von Energiedaten in der Bürgerenergiegemeinschaft dienen.

2. Ziele der Übung

- Implementierung eines vollständigen Microservice in C#/.NET 9
- Integration von RabbitMQ und MongoDB
- Deployment in k3s mit Helm Charts
- API Gateway Setup und Security-Implementierung
- Monitoring und Observability etablieren

3. Voraussetzungen

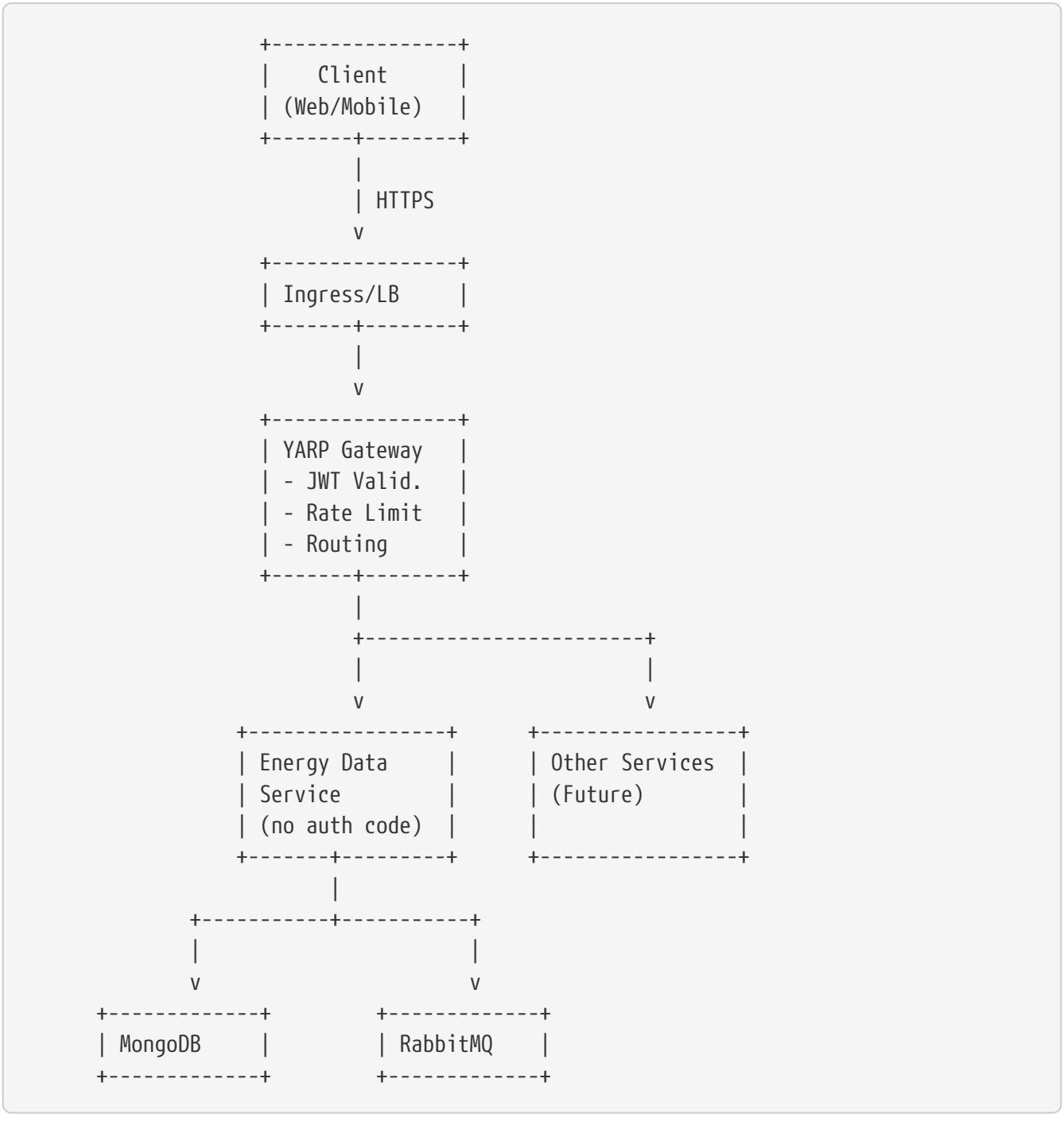
- Funktionierende k3s Umgebung aus LB01
- Installierte Services (RabbitMQ, MongoDB) aus LB01
- Grundkenntnisse in C#/.NET
- k3s installiert und konfiguriert

4. Architektur-Übersicht

4.1. Microservice-Architektur mit API Gateway

In dieser Übung implementieren Sie eine moderne Microservice-Architektur mit zentralem API Gateway. Diese Architektur bietet mehrere Vorteile gegenüber direktem Service-Zugriff:

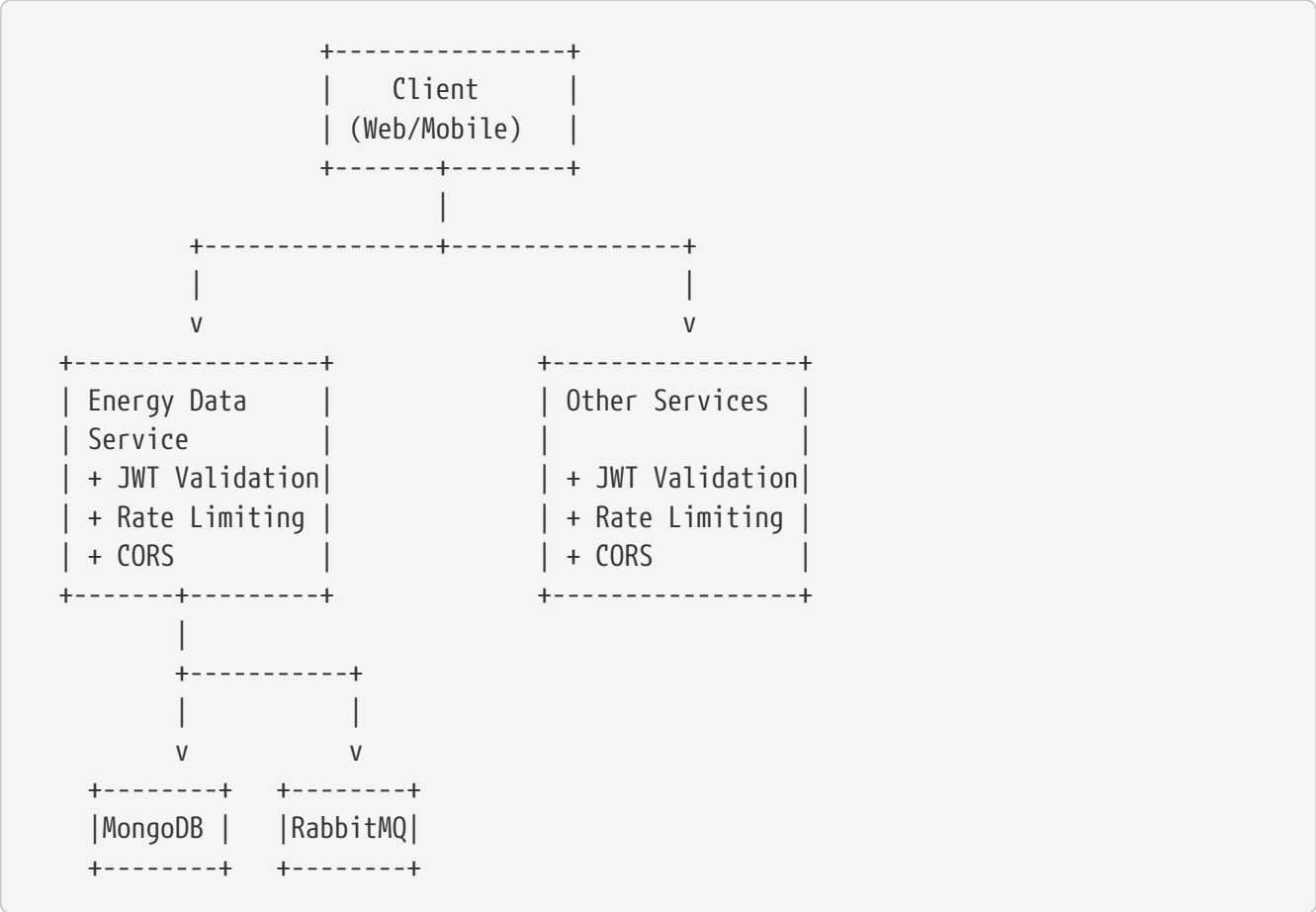
4.1.1. Architektur MIT API Gateway (Empfohlen)



Vorteile:

- **Single Entry Point**: Clients kennen nur einen Endpoint
- **Zentrale Security**: JWT-Validierung nur im Gateway
- **Service Isolation**: Services müssen sich nicht um Auth kümmern
- **Cross-Cutting Concerns**: Rate Limiting, Logging zentral
- **Flexible Routing**: A/B Testing, Canary Deployments möglich
- **Service Discovery**: Clients müssen Service-Topologie nicht kennen

4.1.2. Architektur OHNE API Gateway (Nicht empfohlen)



Nachteile:

- **Code-Duplikation**: Auth-Logic in jedem Service
- **Komplexer Client**: Muss alle Service-Endpoints kennen
- **Inkonsistenz**: Unterschiedliche Security-Implementierungen
- **Schwierige Wartung**: Änderungen in jedem Service nötig
- **Performance**: Mehrere Netzwerk-Hops vom Client

4.2. Warum ein Authentication Provider?

JWT-Tokens müssen von einer **vertrauenswürdigen, zentralen Stelle** ausgestellt werden:

- **Token-Ausstellung**: Nach erfolgreicher User-Authentifizierung
- **Signierung**: Mit privatem Schlüssel (RSA/ECDSA)
- **User-Management**: Zentrale Verwaltung von Users, Rollen, Permissions
- **Token-Refresh**: Automatische Erneuerung abgelaufener Tokens
- **SSO**: Single Sign-On über mehrere Services

Ohne Authentication Provider müssten Sie:

- In jedem Service User-Management implementieren

- Passwort-Hashing und Validierung duplizieren
- Token-Signing-Keys in jedem Service verwalten
- Sicherheitsrisiken durch inkonsistente Implementierung

In dieser Übung verwenden wir Keycloak als Open-Source Identity and Access Management Lösung.

5. Aufgabenstellung

5.1. Aufgabe 1: Energy Data Service Implementierung

Implementieren Sie den Energy Data Service als Microservice. Für die Simulation von Nachrichten verwenden Sie erst mal z. B. PowerShell oder ähnliches.

5.1.1. Consumer-Komponente

Entwickeln Sie einen robusten Message Consumer:

- **RabbitMQ Integration**
  - Consumer für Smart Meter Nachrichten aus LB01
  - Batch-Processing für Performance-Optimierung
  - Implementierung von Retry-Policies
  - Dead Letter Queue für fehlerhafte Nachrichten
- **Message Processing**
  - Validierung eingehender Daten
  - Transformation in MongoDB-Dokumente

5.1.2. MongoDB-Integration

Implementieren Sie die Datenpersistierung:

- **Datenmodell-Beispiel**

```
public class EnergyReading
{
    public ObjectId Id { get; set; }
    public string MeteringPointNumber { get; set; }
    public DateTime Timestamp { get; set; }
    public double Value { get; set; }
    public string Unit { get; set; } // kWh
    public Dictionary<string, object> Metadata { get; set; }
}
```

5.1.3. REST API

Erstellen Sie eine vollständige REST API:

- **Endpoint-Beispiele**

```
GET    /api/v1/energy-data/{meteringPointNumber}
GET    /api/v1/energy-data/{meteringPointNumber}/aggregated
POST   /api/v1/energy-data/query
GET    /api/v1/energy-data/statistics
DELETE /api/v1/energy-data/{meteringPointNumber}
```

- **Features**
  - Pagination mit Cursor-basiertem Ansatz
  - Filtering (Zeitraum, Typ, Status)
  - Sorting und Projektion
  - Response Caching
  - OpenAPI/Swagger Dokumentation
- **Error Handling**
  - Problem Details (RFC 7807)
  - Structured Error Responses
  - Correlation IDs für Tracing

5.2. Aufgabe 2: Containerisierung und k3s Deployment

5.2.1. Docker Image

Erstellen Sie ein optimiertes Docker Image:

```
# Multi-stage build für .NET 9
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
# Build stage...

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS runtime
# Runtime stage mit Security-Härtung
USER nonroot
# ...
```

- **Optimierungen**
  - Multi-stage Build
  - Layer Caching
  - Minimale Base Images

- Health Check Integration

5.2.2. Helm Chart

Entwickeln Sie einen produktionsreifen Helm Chart:

```
# values.yaml Struktur
replicaCount: 3
image:
  repository: energy-registry:5000/energy-data-service
  tag: "1.0.0"
resources:
  limits:
    memory: "512Mi"
    cpu: "500m"
  requests:
    memory: "256Mi"
    cpu: "250m"
```

- **Kubernetes Ressourcen**
  - Deployment mit Rolling Updates
  - Service (ClusterIP)
  - ConfigMap für Konfiguration
  - Secret für Credentials
  - HorizontalPodAutoscaler
  - PodDisruptionBudget
  - NetworkPolicy
- **Advanced Features**
  - Init Containers für DB Migration
  - Liveness/Readiness/Startup Probes
  - Volume Mounts für Logs
  - Service Monitor für Prometheus

5.3. Aufgabe 3: Authentication Provider Setup

Implementieren Sie Keycloak als zentralen Authentication Provider:

5.3.1. Keycloak Installation via Helm

```
# Bitnami Keycloak Helm Chart
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

```
helm install keycloak bitnami/keycloak \
  --namespace auth \
  --create-namespace \
  --set auth.adminUser=admin \
  --set auth.adminPassword=admin123 \
  --set postgresql.enabled=true \
  --set service.type=ClusterIP
```

5.3.2. Keycloak Konfiguration

Realm erstellen:

1. Login in Keycloak Admin Console
2. Neuen Realm erstellen: `energy-community`
3. Client erstellen: `api-gateway`
   - Client Protocol: `openid-connect`
   - Access Type: `confidential`
   - Valid Redirect URIs: `http://localhost:5000/*`
   - Web Origins: `*`

User und Rollen:

```
Rollen:
- energy-admin (voller Zugriff)
- energy-user (read-only)
- meter-operator (Smart Meter Management)

Test-User:
- Username: test.user
- Password: password123
- Rolle: energy-user
```

JWT Token Konfiguration:

- Realm Settings → Tokens
  - Access Token Lifespan: 5 minutes
  - Refresh Token Lifespan: 30 minutes
  - Client Session Idle: 30 minutes

5.3.3. Token-Abruf testen

```
# Token abrufen
TOKEN=$(curl -X POST "http://keycloak.auth.svc.cluster.local/realms/energy-
```

```
community/protocol/openid-connect/token" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "username=test.user" \
  -d "password=password123" \
  -d "grant_type=password" \
  -d "client_id=api-gateway" \
  -d "client_secret=<your-client-secret>" | jq -r '.access_token')

# Token verwenden
curl -H "Authorization: Bearer $TOKEN" \
  http://api-gateway.default.svc.cluster.local/api/v1/energy-data
```

5.4. Aufgabe 4: API Gateway mit YARP

Implementieren Sie ein API Gateway mit YARP (Yet Another Reverse Proxy) inklusive JWT-Validierung:

5.4.1. Projekt Setup

```
dotnet new web -n ApiGateway
cd ApiGateway
dotnet add package Yarp.ReverseProxy
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

5.4.2. YARP Configuration mit JWT-Validierung

Program.cs:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;

var builder = WebApplication.CreateBuilder(args);

// JWT Authentication konfigurieren
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.Authority = "http://keycloak.auth.svc.cluster.local/realms/energy-
community";
        options.Audience = "api-gateway";
        options.RequireHttpsMetadata = false; // Nur für Development!

        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
```

```
            ValidateIssuerSigningKey = true,
            ClockSkew = TimeSpan.FromMinutes(5)
        };
    });

builder.Services.AddAuthorization(options =>
{
    // Policy für Admin-Zugriff
    options.AddPolicy("AdminOnly", policy =>
        policy.RequireClaim("realm_access", "energy-admin"));

    // Policy für authentifizierte User
    options.AddPolicy("AuthenticatedUser", policy =>
        policy.RequireAuthenticatedUser());
});

// YARP konfigurieren
builder.Services.AddReverseProxy()
    .LoadFromConfig(builder.Configuration.GetSection("ReverseProxy"));

// Rate Limiting
builder.Services.AddRateLimiter(options =>
{
    options.GlobalLimiter = PartitionedRateLimiter.Create<HttpContext, string>(context
=>
        RateLimitPartition.GetFixedWindowLimiter(
            partitionKey: context.User.Identity?.Name ??
context.Request.Headers.Host.ToString(),
            factory: _ => new FixedWindowRateLimiterOptions
            {
                PermitLimit = 100,
                Window = TimeSpan.FromMinutes(1)
            }));
});

var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();
app.UseRateLimiter();

app.MapReverseProxy();

app.Run();
```

appsettings.json - YARP Routing Configuration:

```
{
  "ReverseProxy": {
    "Routes": {
```

```
      "energy-data-route": {
        "ClusterId": "energy-data-cluster",
        "AuthorizationPolicy": "AuthenticatedUser",
        "RateLimiterPolicy": "fixed",
        "Match": {
          "Path": "/api/v1/energy-data/{**catch-all}"
        },
        "Transforms": [
          {
            "RequestHeader": "X-User-Id",
            "Set": "{user.sub}"
          },
          {
            "RequestHeader": "X-User-Roles",
            "Set": "{user.realm_access.roles}"
          }
        ]
      },
      "admin-route": {
        "ClusterId": "energy-data-cluster",
        "AuthorizationPolicy": "AdminOnly",
        "Match": {
          "Path": "/api/v1/admin/{**catch-all}"
        }
      }
    },
    "Clusters": {
      "energy-data-cluster": {
        "Destinations": {
          "destination1": {
            "Address": "http://energy-data-service.default.svc.cluster.local"
          }
        },
        "HealthCheck": {
          "Active": {
            "Enabled": true,
            "Interval": "00:00:10",
            "Timeout": "00:00:05",
            "Policy": "ConsecutiveFailures",
            "Path": "/health"
          }
        },
        "LoadBalancingPolicy": "RoundRobin"
      }
    }
  }
}
```

5.4.3. Custom Middleware für Request Logging

```
public class RequestLoggingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<RequestLoggingMiddleware> _logger;

    public RequestLoggingMiddleware(RequestDelegate next,
ILogger<RequestLoggingMiddleware> logger)
    {
        _next = next;
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var correlationId = Guid.NewGuid().ToString();
        context.Request.Headers.Add("X-Correlation-Id", correlationId);

        _logger.LogInformation(
            "Request {Method} {Path} from {User} with CorrelationId {CorrelationId}",
            context.Request.Method,
            context.Request.Path,
            context.User.Identity?.Name ?? "Anonymous",
            correlationId);

        await _next(context);

        _logger.LogInformation(
            "Response {StatusCode} for CorrelationId {CorrelationId}",
            context.Response.StatusCode,
            correlationId);
    }
}

// In Program.cs registrieren:
app.UseMiddleware<RequestLoggingMiddleware>();
```

5.4.4. Energy Data Service - Authentifizierung entfernen

Da YARP die JWT-Validierung übernimmt, kann der Energy Data Service vereinfacht werden:

```
// Energy Data Service Program.cs
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

// KEINE JWT-Authentifizierung mehr nötig!
// YARP leitet bereits validierte Claims weiter
```

```
var app = builder.Build();

app.MapControllers();
app.Run();
```

Controller kann User-Info aus Headers lesen:

```
[ApiController]
[Route("api/v1/energy-data")]
public class EnergyDataController : ControllerBase
{
    [HttpGet("{meteringPointNumber}")]
    public IActionResult GetEnergyData(string meteringPointNumber)
    {
        // User-Info aus YARP-Headers
        var userId = Request.Headers["X-User-Id"].ToString();
        var userRoles = Request.Headers["X-User-Roles"].ToString();

        _logger.LogInformation(
            "User {UserId} with roles {Roles} accessing metering point
{MeteringPoint}",
            userId, userRoles, meteringPointNumber);

        // Business Logic...
        return Ok(data);
    }
}
```

5.4.5. Deployment

Dockerfile:

```
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src
COPY ["ApiGateway.csproj", "./"]
RUN dotnet restore
COPY . .
RUN dotnet publish -c Release -o /app/publish

FROM mcr.microsoft.com/dotnet/aspnet:9.0
WORKDIR /app
COPY --from=build /app/publish .
USER nonroot
ENTRYPOINT ["dotnet", "ApiGateway.dll"]
```

Helm Chart erstellen: - Deployment mit 2-3 Replicas - Service (ClusterIP) - ConfigMap für appsettings.json - Ingress für externen Zugriff - Secret für Keycloak Client Secret

5.4.6. Testing

```
# Token abrufen
TOKEN=$(curl -X POST "http://keycloak/realms/energy-community/protocol/openid-
connect/token" \
  -d "username=test.user" \
  -d "password=password123" \
  -d "grant_type=password" \
  -d "client_id=api-gateway" \
  -d "client_secret=<secret>" | jq -r '.access_token')

# Über API Gateway zugreifen
curl -H "Authorization: Bearer $TOKEN" \
  http://api-gateway/api/v1/energy-data/AT0010000000000000001000012345678

# Ohne Token (sollte 401 zurückgeben)
curl http://api-gateway/api/v1/energy-data/AT0010000000000000001000012345678
```

5.5. Aufgabe 5: Observability & Monitoring

5.5.1. Structured Logging

Implementieren Sie Logging mit z. B. Serilog:

```
Log.Logger = new LoggerConfiguration()
    .WriteTo.Console(new JsonFormatter())
    .WriteTo.MongoDB(mongoDbConnection)
    .Enrich.WithCorrelationId()
    .CreateLogger();
```

- Log Aggregation
- Correlation IDs
- Request/Response Logging
- Performance Logging

5.5.2. Metrics & Tracing

- **Prometheus Metrics**
  - Custom Metrics (Request Count, Duration)
  - MongoDB Query Metrics
  - RabbitMQ Consumer Metrics
- **OpenTelemetry**
  - Distributed Tracing

    - Span Attributes
    - Context Propagation
- **Grafana Dashboard**
    - Service Health Overview
    - Request Rate & Latency
    - Error Rate Tracking
    - Resource Utilization

5.5.3. Health Checks

```
services.AddHealthChecks()
    .AddMongoDb(mongoConnectionString)
    .AddRabbitMQ(rabbitConnectionString)
    .AddCheck<CustomHealthCheck>("custom");
```

6. Sicherheitsanforderungen

Implementieren Sie folgende Security-Maßnahmen:

6.1. Application Security

- Input Validation
- SQL/NoSQL Injection Prevention
- Secure Configuration Management
- Secrets Management (k8s Secrets)

6.2. Container Security

- Non-root User
- Read-only Root Filesystem
- Security Context Configuration
- Resource Limits

6.3. API Security

- JWT Authentication
- Rate Limiting
- API Versioning
- HTTPS Only

7. Zeitplan

- **Ausgabe**: Nach Abschluss LB01
- **Bearbeitungszeit**: 2 Wochen
- **Präsentationen**: Am Ende des Labors

8. Hilfreiche Ressourcen

8.1. .NET & C#

- ASP.NET Core Web API
- .NET Microservices

8.2. MassTransit & RabbitMQ

- MassTransit Documentation
- RabbitMQ Transport

8.3. MongoDB

- MongoDB C# Driver
- Time-Series Collections

8.4. Kubernetes & Helm

- Helm Chart Development
- Pod Lifecycle

8.5. Keycloak

- Keycloak Documentation
- Keycloak Server Administration
- Bitnami Keycloak Helm Chart

8.6. YARP & JWT

- YARP Documentation
- YARP Samples
- JWT Authentication in ASP.NET Core
- Rate Limiting in ASP.NET Core

8.7. Observability

- Serilog
- OpenTelemetry .NET
- Prometheus Client Libraries

9. Hinweise

9.1. Allgemein

- Nutzen Sie die bestehende k3s Umgebung aus LB01
- Verwenden Sie Environment Variables für Konfiguration
- Implementieren Sie Unit Tests für kritische Komponenten
- Dokumentieren Sie Ihre API mit OpenAPI/Swagger
- Verwenden Sie Semantic Versioning für Images
- Beachten Sie die 12-Factor App Prinzipien
- Security First - implementieren Sie Sicherheit von Anfang an