Gateway Service

Overview

The Gateway service is an API gateway using Ocelot to route requests to various microservices in the OllamaNet ecosystem. It provides a unified entry point for client applications to access different services, handling authentication, authorization, request routing, and claims forwarding.

Core Functionality

Request Routing

- Routes client requests to appropriate backend microservices (Auth, Admin, Explore, Conversation)
- Implements consistent URL schema for predictable routing patterns
- Supports versioning and path-based routing
- Handles request transformation when necessary

Authentication & Authorization

- Validates JWT tokens at the gateway level
- Implements role-based access control for protected endpoints
- Forwards user claims to downstream services
- Rejects unauthorized requests with appropriate status codes

Configuration Management

- Implements modular configuration with service-specific files
- Supports variable-based configuration for service URLs
- Enables dynamic configuration reloading without service restart
- Provides unified configuration from multiple source files

Resilience & Monitoring

- Implements basic rate limiting for abuse prevention
- Provides resilience and fallback mechanisms for service failures
- Supports logging and monitoring of request patterns
- Handles error responses from downstream services

Architecture

Configuration-as-Code

The Gateway implements a configuration-as-code approach with the following components:

- **Service-Specific Files**: Configuration split by service domain (Auth.json, Admin.json, etc.)
- **Variable Substitution**: Service URLs defined in ServiceUrls.json and referenced with ${variable} syntax
- **Dynamic Reloading**: File watching for configuration changes
- **Unified Configuration**: Multiple files combined into single Ocelot configuration

Key Components

- **ConfigurationLoader**: Combines multiple configuration sources into a unified configuration
- **ConfigurationChangeMonitor**: Watches for file changes and triggers reloads
- **ClaimsToHeaderMiddleware**: Forwards user claims to downstream services
- **RoleAuthorizationMiddleware**: Enforces role-based access control
- **JwtMiddleware**: Validates JWT tokens before request processing

Request Flow

1. Client sends request to Gateway
2. JWT validation middleware authenticates the request
3. Role authorization middleware checks permissions
4. Ocelot middleware determines the target service
5. Claims forwarding middleware adds user information
6. Request is forwarded to the appropriate service
7. Response is returned to the client

Configuration Structure

ServiceUrls.json

```
{
  "Services": {
    "Auth": {
      "Host": "localhost",
      "Port": 5249,
      "Scheme": "http"
    },
    "Admin": {
      "Host": "localhost",
      "Port": 5038,
      "Scheme": "http"
    },
    "Explore": {
      "Host": "localhost",
      "Port": 5167,
      "Scheme": "http"
    },
    "Conversation": {
      "Host": "localhost",
      "Port": 5156,
      "Scheme": "http"
    }
  }
}
```

Service-Specific Configuration Files

- **Auth.json**: Routes for authentication service
- **Admin.json**: Routes for administration service
- **Explore.json**: Routes for exploration service
- **Conversation.json**: Routes for conversation service

Configuration Format

Each service configuration follows this pattern:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/{endpoint}",
      "DownstreamScheme": "${Services:ServiceName:Scheme}",
      "DownstreamHostAndPorts": [
        {
          "Host": "${Services:ServiceName:Host}",
          "Port": ${Services:ServiceName:Port}
        }
      ],
      "UpstreamPathTemplate": "/api/servicename/{endpoint}",
      "UpstreamHttpMethod": [ "GET", "POST", "PUT", "DELETE" ],
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "Bearer"
      },
      "RouteClaimsRequirement": {
        "role": "User"
      }
    }
  ]
}
```

Integration Points

Frontend Application

- Web UI consuming the Gateway API via CORS policy
- Authentication flow through the Gateway
- Request routing for all service operations

Downstream Services

- **Auth Service**: User authentication and authorization
- **Admin Service**: Platform administration
- **Explore Service**: Model discovery and browsing
- **Conversation Service**: Conversation management and chat

Security Implementation

JWT Authentication

- Token validation with comprehensive checks
- Signature validation using shared secret key
- Audience and issuer validation
- Expiration time validation
- Claims extraction for authorization

Role-Based Authorization

- Role claims validation for protected endpoints
- Different role requirements for different routes
- Admin-only routes for administrative functions
- User-level access for standard operations

Claims Forwarding

- User ID forwarding via X-User-Id header
- Role forwarding via X-User-Role header
- Claims transformation for downstream service consumption
- Consistent user context across all services

Future Enhancements

Planned Improvements

- Advanced rate limiting with Redis
- Circuit breaker implementation for service resilience
- Request/response transformation
- Cache-control header management
- Enhanced monitoring and logging
- Configuration dashboard for management

Configuration Dashboard

A planned management interface for configuration with:

- Visual configuration editing
- Configuration validation
- History and rollback capabilities
- Environment-specific configurations
- Import/export functionality

Known Issues

- Rate limiting needs optimization for distributed scenarios
- Limited request transformation capabilities
- No circuit breaker for service failures
- Configuration changes require manual file editing

Performance Considerations

- Gateway becomes a potential bottleneck as traffic increases
- JWT validation adds processing overhead
- Configuration reloading impact on request processing
- Memory usage with large configuration files