# Admin Service

## Overview

AdminService is a critical microservice component of the OllamaNet platform that provides comprehensive administrative capabilities. It serves as the central control point for platform administrators to manage all aspects of the system, including users, AI models, tags, and inference operations. The service is designed with a clean architecture, domain-driven design, robust validation, and RESTful API principles to ensure secure and efficient platform administration.

## Core Functionality

### User Management

- Create, read, update, delete users
- Manage user roles and permissions
- Control account status
- User search and filtering
- Batch user operations

### AI Model Management

- Add, update, delete AI models
- Manage model metadata
- Assign and remove tags for categorization
- Model search and filtering
- Model status monitoring

### Tag Management

- Create, update, delete tags
- Organize models through tag relationships
- Tag search and filtering
- Tag assignment to models

### Inference Operations

- Install AI models through Ollama integration
- Uninstall AI models when no longer needed
- Monitor model installation progress through streaming
- Model status checking and validation

## Architecture

### Domain-Driven Design

The service follows a domain-driven design with clear separation of concerns:

- **User Domain**: User management services, DTOs, and controllers
- **AIModel Domain**: Model management services, DTOs, and controllers
- **Tag Domain**: Tag management services, DTOs, and controllers
- **Inference Domain**: Ollama integration services, DTOs, and controllers

## Layered Architecture

### API Layer

- Controllers/User/UserOperationsController
- Controllers/AIModel/AIModelOperationsController
- Controllers/Tag/TagOperationsController
- Controllers/Inference/InferenceOperationsController
- Controllers/Validators for each domain

### Service Layer

- Services/User/Implementation/UserService.cs
- Services/AIModel/Implementation/ModelService.cs
- Services/Tag/Implementation/TagService.cs
- Services/Inference/Implementation/InferenceService.cs
- Domain-specific DTOs and mappers

### Infrastructure Layer

- Infrastructure/Caching: Redis implementation
- Infrastructure/Authentication: JWT implementation (commented)
- Infrastructure/Configuration: Options pattern implementation
- Infrastructure/Integration: External service connectors
- Infrastructure/Logging: Comprehensive logging
- Infrastructure/ErrorHandling: Global exception handling

### Data Access Layer

- Repository pattern via IUnitOfWork from Ollama_DB_layer
- Domain-specific repositories

## Key Components

### InferenceEngineConnector

- Abstracts integration with the Ollama API
- Handles model installation and uninstallation
- Implements progress streaming via IAsyncEnumerable
- Manages connection to external Ollama service

### Caching System

- ICacheService interface with Redis implementation

- Domain-specific TTL values
- Cache invalidation on data changes
- Centralized cache key management

**Validation Framework**

- FluentValidation for comprehensive request validation
- Domain-specific validators
- Conditional validation rules
- Consistent error responses

**Error Handling**

- Global exception handler
- Exception type mapping to HTTP status codes
- Contextual logging
- Structured error responses

# API Design

## RESTful Endpoints

The service provides a comprehensive set of RESTful endpoints organized by domain:

**User Management**

- GET /api/Admin/User - Get all users with pagination
- GET /api/Admin/User/{id} - Get user by ID
- POST /api/Admin/User - Create new user
- PUT /api/Admin/User/{id} - Update user
- DELETE /api/Admin/User/{id} - Delete user
- GET /api/Admin/User/Search - Search users by criteria
- POST /api/Admin/User/Role - Assign role to user

**AI Model Management**

- GET /api/Admin/AIModel - Get all models with pagination
- GET /api/Admin/AIModel/{id} - Get model by ID
- POST /api/Admin/AIModel - Create new model
- PUT /api/Admin/AIModel/{id} - Update model
- DELETE /api/Admin/AIModel/{id} - Delete model
- GET /api/Admin/AIModel/Search - Search models by criteria
- POST /api/Admin/AIModel/Tag - Assign tag to model

**Tag Management**

- GET /api/Admin/Tag - Get all tags with pagination
- GET /api/Admin/Tag/{id} - Get tag by ID

- POST /api/Admin/Tag - Create new tag
- PUT /api/Admin/Tag/{id} - Update tag
- DELETE /api/Admin/Tag/{id} - Delete tag
- GET /api/Admin/Tag/Search - Search tags by criteria

**Inference Operations**

- POST /api/Admin/Inference/Install - Install model
- POST /api/Admin/Inference/Uninstall - Uninstall model
- GET /api/Admin/Inference/Status/{modelName} - Check model status
- GET /api/Admin/Inference/Stream/Install/{modelName} - Stream installation progress

### Status Codes

- 200: Success responses
- 201: Creation success
- 400: Bad request (validation failures)
- 404: Not found (resource not located)
- 500: Server errors

### Request/Response Models

All endpoints use strongly-typed DTOs for operations:

- Request models with validation attributes
- Response models with consistent structure
- Pagination support for collection endpoints
- Error responses with details

## Streaming Implementation

### Server-Sent Events

The service implements Server-Sent Events (SSE) for streaming progress updates during model installation:

```
[HttpGet("Stream/Install/{modelName}")]
public async Task InstallModelWithStreamingProgress(string modelName)
{
    Response.Headers.Add("Content-Type", "text/event-stream");

    var progress = new Progress<InstallationProgress>(p =>
    {
        var json = JsonSerializer.Serialize(p);
        Response.WriteAsync($"data: {json}\n\n");
        Response.Body.FlushAsync();
    });

    await _inferenceService.InstallModelWithProgressAsync(modelName, progress);
}
```

### Progress Reporting

- IProgress for progress notifications
- InstallationProgress DTO with percentage and status
- IAsyncEnumerable for streaming data
- Response.BodyWriter for direct streaming to clients

## Caching Strategy

### User Domain

- Cache user profiles (5 min TTL)
- Cache user roles (15 min TTL)

### AI Model Domain

- Cache model metadata (10 min TTL)
- Cache model tags (15 min TTL)

### Tag Domain

- Cache all tags (30 min TTL)
- Cache tag relationships (15 min TTL)

### Cache Implementation

```
public async Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> factory, TimeSpan
expiration)
{
    var cachedValue = await _redisCache.GetStringAsync(key);

    if (cachedValue != null)
    {
        return JsonSerializer.Deserialize<T>(cachedValue);
    }

    var result = await factory();

    await _redisCache.SetStringAsync(
        key,
        JsonSerializer.Serialize(result),
        new DistributedCacheEntryOptions { AbsoluteExpirationRelativeToNow =
expiration }
    );

    return result;
}
```

## Configuration Management

## Options Pattern

The service uses the options pattern for all configuration:

- InferenceEngineOptions for Ollama integration settings
- UserManagementOptions for user management settings
- ModelManagementOptions for model management settings
- RedisCacheOptions for caching settings
- LoggingOptions for logging configuration

## Sample Configuration

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=db19911.public.databaseasp.net;Database=Ollama_DB;User
Id=xxx;Password=xxx;TrustServerCertificate=True;"
  },
  "Redis": {
    "ConnectionString": "content-ghoul-
42217.upstash.io:42217,password=xxx,ssl=True,abortConnect=False"
  },
  "InferenceEngine": {
    "BaseUrl": "https://704e-35-196-162-195.ngrok-free.app",
    "Timeout": 300,
    "RetryCount": 3
  },
  "UserManagement": {
    "DefaultRole": "User",
    "PasswordRequireDigit": true,
    "PasswordRequireLowercase": true,
    "PasswordRequireUppercase": true,
    "PasswordRequireNonAlphanumeric": true,
    "PasswordMinLength": 8
  },
  "ModelManagement": {
    "DefaultPageSize": 10,
    "MaxPageSize": 100,
    "AllowedFileTypes": [".jpg", ".png", ".txt"]
  },
  "RedisCacheSettings": {
    "UserProfileTTLMinutes": 5,
    "UserRolesTTLMinutes": 15,
    "ModelMetadataTTLMinutes": 10,
    "ModelTagsTTLMinutes": 15,
    "AllTagsTTLMinutes": 30,
    "TagRelationshipsTTLMinutes": 15
  }
}
```

# Integration Points

## Ollama_DB_layer

- Database access through repository interfaces
- Entity definitions for domain objects
- UnitOfWork pattern for transaction management

## InferenceEngineConnector

- Integration with the Ollama API
- Model installation and management
- Progress streaming for long-running operations

## Frontend Application

- Web UI consuming the AdminService API
- CORS policy configuration for secure access

## Redis Cache

- Distributed caching for performance optimization
- Domain-specific TTL configurations

# Error Handling

## Global Exception Handler

The service implements a global exception handler that maps exception types to appropriate HTTP status codes:

- ValidationException → 400 Bad Request
- NotFoundException → 404 Not Found
- BusinessException → 400 Bad Request
- General exceptions → 500 Internal Server Error

## Error Response Structure

```
{
  "status": 400,
  "message": "Validation failed",
  "errors": [
    "Username is required",
    "Email must be a valid email address"
  ],
  "timestamp": "2023-06-15T10:30:45Z",
  "path": "/api/Admin/User"
}
```

# Validation Strategy

## FluentValidation Rules

```
public class CreateUserValidator : AbstractValidator<CreateUserRequest>
{
    public CreateUserValidator()
    {
        RuleFor(x => x.Username)
            .NotEmpty().WithMessage("Username is required")
            .MinimumLength(3).WithMessage("Username must be at least 3
characters")
            .MaximumLength(50).WithMessage("Username cannot exceed 50
characters");

        RuleFor(x => x.Email)
            .NotEmpty().WithMessage("Email is required")
            .EmailAddress().WithMessage("Email must be a valid email address");

        RuleFor(x => x.Password)
            .NotEmpty().WithMessage("Password is required")
            .MinimumLength(8).WithMessage("Password must be at least 8
characters")
            .Matches("[A-Z]").WithMessage("Password must contain at least one
uppercase letter")
            .Matches("[a-z]").WithMessage("Password must contain at least one
lowercase letter")
            .Matches("[0-9]").WithMessage("Password must contain at least one
digit")
            .Matches("[^a-zA-Z0-9]").WithMessage("Password must contain at least
one special character");
    }
}
```

# Logging Strategy

## Structured Logging

The service implements structured logging with Serilog:

- Request/response logging
- Error logging with context
- Performance monitoring
- Audit logging for administrative actions

## Log Levels

- Trace: Detailed debugging information
- Debug: Debugging information
- Information: General information

- Warning: Potential issues
- Error: Errors that don't crash the application
- Critical: Critical errors that crash the application

## Migration Plan

The service is undergoing a significant architectural restructuring organized into seven phases:

1. **Preparation and Analysis**: Set up foundation and analyze existing code
2. **Infrastructure Layer Setup**: Establish cross-cutting concerns
3. **Domain Services Layer**: Implement domain-specific services
4. **Controllers and Validators**: Reorganize by domain
5. **Error Handling and Validation**: Implement consistent approach
6. **Integration and Testing**: Integrate all components and test
7. **Documentation and Cleanup**: Finalize documentation and remove obsolete components

## Known Issues

- Batch operations for efficient management not implemented
- Limited monitoring tools for system administrators
- Incomplete backup and recovery capabilities

## Performance Considerations

- Caching strategy optimization for high-traffic scenarios
- Database query optimization for large datasets
- Connection pooling configuration
- Asynchronous operations for non-blocking I/O
- Background processing for long-running tasks