"# Conversation Service

Overview

ConversationService is a core microservice component of the OllamaNet platform that manages all aspects of user conversations with AI models. It provides real-time chat capabilities with streaming responses, conversation organization, message history persistence, note management, and feedback collection. The service implements sophisticated caching strategies, RAG (Retrieval-Augmented Generation) capabilities, and document processing to enhance conversation context and quality.

Core Functionality

Conversation Management

- Create, read, update, delete conversations
- Organize conversations in hierarchical folders
- Persist and retrieve message history
- Support for conversation metadata and properties
- Conversation search and filtering

Real-time Chat

- Process user messages and generate AI responses
- Stream responses in real-time via Server-Sent Events (SSE)
- Support for different AI models and parameters
- Handle conversation context and history
- Support for conversation continuation

Note Management

- Create and manage notes associated with conversations
- Organize notes within conversations
- Search and filter notes
- Support for rich text formatting

Feedback Collection

- Collect user feedback on AI responses
- Store feedback for quality assessment
- Support for different feedback types (thumbs up/down, ratings)
- Associate feedback with specific messages

RAG Capabilities

- Upload and process documents for context enhancement
- Extract text from various document formats
- Generate embeddings for semantic search
- Retrieve relevant context for conversation enhancement
- Ground AI responses in document context

- Provide citations for information sources

Folder Organization

- Create hierarchical folder structure
- Move conversations between folders
- Folder permissions and sharing
- Folder metadata and properties

Architecture

Layered Architecture

The ConversationService follows a clean, layered architecture:

- **API Layer**: Controllers handling HTTP requests and responses
- **Service Layer**: Business logic implementation
- **Data Access Layer**: Repository pattern for data operations
- **Integration Layer**: External service connections (Ollama, vector DB)
- **Infrastructure Layer**: Cross-cutting concerns (caching, logging)

Key Components

ConversationController

- Handles conversation CRUD operations
- Manages folder organization
- Implements conversation search and filtering

ChatController

- Processes chat messages
- Implements streaming response endpoints
- Handles model selection and parameters

NoteController

- Manages note operations
- Implements note search and filtering

FeedbackController

- Collects and manages user feedback
- Implements feedback analytics endpoints

DocumentController

- Handles document upload and processing
- Manages document organization and retrieval

ConversationService

- Implements conversation business logic
- Coordinates between repositories and cache
- Handles conversation organization and metadata

ChatService

- Processes chat messages through AI models
- Manages conversation context and history
- Implements streaming response generation
- Coordinates RAG capabilities

DocumentProcessingService

- Processes uploaded documents
- Extracts text from various formats
- Generates embeddings for semantic search
- Manages document storage and retrieval

VectorSearchService

- Implements semantic search capabilities
- Retrieves relevant context for conversations
- Manages vector database integration
- Implements relevance scoring and ranking

CacheManager

- Provides high-level caching abstraction
- Implements domain-specific caching strategies
- Handles cache invalidation and updates
- Manages TTL for different data types

API Design

RESTful Endpoints

Conversation Management

- **GET /api/conversation** - Get all conversations with pagination
- **GET /api/conversation/{id}** - Get conversation by ID
- **POST /api/conversation** - Create new conversation
- **PUT /api/conversation/{id}** - Update conversation
- **DELETE /api/conversation/{id}** - Delete conversation
- **GET /api/conversation/search** - Search conversations by criteria

Folder Management

- **GET /api/folder** - Get all folders
- **GET /api/folder/{id}** - Get folder by ID
- **POST /api/folder** - Create new folder
- **PUT /api/folder/{id}** - Update folder
- **DELETE /api/folder/{id}** - Delete folder
- **POST /api/folder/{id}/move/{conversationId}** - Move conversation to folder

Chat Interaction

- **POST /api/chat/{conversationId}** - Send message to conversation
- **GET /api/chat/{conversationId}/stream** - Stream chat responses (SSE)
- **GET /api/chat/{conversationId}/history** - Get chat history
- **POST /api/chat/{conversationId}/model** - Change AI model for conversation

Note Management

- **GET /api/note/{conversationId}** - Get notes for conversation
- **POST /api/note/{conversationId}** - Create note for conversation
- **PUT /api/note/{id}** - Update note
- **DELETE /api/note/{id}** - Delete note

Feedback Collection

- **POST /api/feedback/{messageId}** - Submit feedback for message
- **GET /api/feedback/stats** - Get feedback statistics

Document Processing

- **POST /api/document/upload** - Upload document
- **GET /api/document/{id}** - Get document details
- **DELETE /api/document/{id}** - Delete document
- **GET /api/document/search** - Search documents by content

Streaming Implementation

Server-Sent Events (SSE)

The service implements SSE for streaming chat responses:

```
[HttpGet("{conversationId}/stream")]
public async Task StreamChatResponse(string conversationId, [FromQuery] string message)
{
    Response.Headers.Add("Content-Type", "text/event-stream");
    Response.Headers.Add("Cache-Control", "no-cache");
    Response.Headers.Add("Connection", "keep-alive");

    var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
    await foreach (var chunk in _chatService.StreamResponseAsync(conversationId, 
message, userId))
    {
        var json = JsonSerializer.Serialize(new { text = chunk });
        await Response.WriteAsync($"data: {json}\n\n");
        await Response.Body.FlushAsync();
    }
}
```

RAG Implementation

Document Processing Pipeline

1. **Document Upload**: User uploads document via API
2. **Text Extraction**: Service extracts text based on document format
3. **Chunking**: Text is divided into manageable chunks
4. **Embedding Generation**: Chunks are converted to vector embeddings
5. **Vector Storage**: Embeddings stored in vector database
6. **Metadata Storage**: Document metadata stored in SQL database

Conversation Enhancement

1. **User Query**: User sends message in conversation
2. **Query Embedding**: Message converted to vector embedding
3. **Semantic Search**: System searches for relevant chunks
4. **Context Retrieval**: Most relevant chunks retrieved
5. **Prompt Construction**: System builds prompt with retrieved context
6. **AI Response**: Enhanced response generated with context
7. **Citation**: Response includes citations to source documents

Document Processors

- **PDFProcessor**: Extracts text from PDF documents
- **WordProcessor**: Extracts text from Word documents
- **TextProcessor**: Processes plain text documents
- **MarkdownProcessor**: Processes markdown documents

Caching Strategy

Conversation Caching

- **ConversationList:{userId}**: User's conversations (5 min TTL)
- **ConversationDetail:{id}**: Conversation details (10 min TTL)
- **ConversationMessages:{id}**: Conversation messages (15 min TTL)
- **UserFolders:{userId}**: User's folder structure (30 min TTL)

Model Caching

- **ModelParameters:{modelId}**: Model parameters (60 min TTL)

- **ModelCapabilities:{modelId}**: Model capabilities (60 min TTL)

Document Caching

- **DocumentMetadata:{id}**: Document metadata (30 min TTL)
- **DocumentChunks:{id}**: Document chunks (30 min TTL)

Vector Database Integration

Pinecone Integration

The service integrates with Pinecone for vector storage and retrieval:

```csharp
public async Task<IEnumerable<DocumentChunk>> SearchSimilarChunks(string query,
int limit = 5)
{
    var embedding = await _embeddingService.GenerateEmbeddingAsync(query);

    var searchRequest = new SearchRequest
    {
        Vector = embedding.ToArray(),
        TopK = limit,
        IncludeMetadata = true,
        Namespace = "document-chunks"
    };

    var searchResponse = await _pineconeClient.SearchAsync(_indexName,
searchRequest);

    return searchResponse.Matches.Select(match => new DocumentChunk
    {
        Id = match.Metadata["chunkId"].ToString(),
        DocumentId = match.Metadata["documentId"].ToString(),
        Content = match.Metadata["content"].ToString(),
        Score = match.Score
    });
}
```

Error Handling

Exception Hierarchy

- **ConversationServiceException**: Base exception for all service exceptions
- **ConversationNotFoundException**: Thrown when a requested conversation is not found
- **FolderNotFoundException**: Thrown when a requested folder is not found
- **DocumentProcessingException**: Thrown when document processing fails
- **AIModelException**: Thrown when AI model interaction fails
- **VectorSearchException**: Thrown when vector search operations fail

Error Responses

```
{
  "status": 404,
  "message": "Conversation not found",
  "details": "The conversation with ID '12345' does not exist",
  "timestamp": "2023-06-15T10:30:45Z",
  "path": "/api/conversation/12345"
}
```

Configuration Management

ConversationServiceSettings

```
"ConversationServiceSettings": {
  "DefaultPageSize": 10,
  "MaxPageSize": 100,
  "DefaultModelId": "llama2-7b",
  "MaxMessageHistoryCount": 50,
  "DefaultStreamingEnabled": true,
  "MaxDocumentSizeMB": 10,
  "ChunkSize": 1000,
  "ChunkOverlap": 200,
  "MaxRelevantChunks": 5
}
```

RedisCacheSettings

```
"RedisCacheSettings": {
  "ConnectionString": "content-ghoul-
42217.upstash.io:42217,password=xxx,ssl=True,abortConnect=False",
  "ConversationListTTLMinutes": 5,
  "ConversationDetailTTLMinutes": 10,
  "ConversationMessagesTTLMinutes": 15,
  "UserFoldersTTLMinutes": 30,
  "ModelParametersTTLMinutes": 60,
  "ModelCapabilitiesTTLMinutes": 60,
  "DocumentMetadataTTLMinutes": 30,
  "DocumentChunksTTLMinutes": 30
}
```

VectorDatabaseSettings

```
"VectorDatabaseSettings": {
  "Provider": "Pinecone",
  "ApiKey": "your-api-key-here",
  "Environment": "us-west1-gcp",
  "IndexName": "ollamanet-documents",
  "Dimensions": 1536,
  "Metric": "cosine"
}
```

Integration Points

Ollama API

- Integration with Ollama for AI model operations
- Chat completion requests
- Model information retrieval
- Streaming response handling

Vector Database

- Pinecone for vector storage and retrieval
- Semantic search capabilities
- Document chunk management
- Relevance scoring

Redis Cache

- Distributed caching for performance optimization
- Domain-specific TTL configurations
- Cache invalidation strategies

Document Storage

- File system or blob storage for document files
- Metadata storage in SQL database
- Access control and permissions

Performance Optimization

Efficient Data Retrieval

- Optimized database queries for conversation listing
- Pagination to limit result set size
- Projection queries to retrieve only needed fields
- Eager loading of related entities where appropriate

Strategic Caching

- Frequently accessed data cached with appropriate TTL
- Cache invalidation on data changes
- Two-tier caching architecture with fallback
- Circuit breaker to prevent cascade failures

Streaming Optimization

- Chunked response streaming for real-time updates
- Efficient buffer management
- Connection pooling for external services
- Timeout handling for long-running operations

Known Issues

- Limited support for multimedia in conversations
- Vector search performance can degrade with large document collections
- No real-time collaboration features
- Limited formatting options for notes

Future Enhancements

- Enhanced multimedia support (images, audio)
- Real-time collaboration features
- Advanced document processing capabilities
- Improved semantic search with hybrid retrieval
- Conversation summarization features
- Enhanced analytics and insights
- Multi-model conversations"