Tunis Business School

ConvoLink

A RESTful API designed for seamless user-to-user communication

Arij Kadhi

January 2026

Contents

1 Introduction

1.1 Background

In today's digital landscape, messaging systems are fundamental components of modern web applications. From social media platforms to enterprise collaboration tools, the ability to send and receive messages securely and efficiently is crucial. However, implementing a robust messaging system requires careful consideration of authentication, data modeling, scalability, and security concerns.

1.2 Motivation

This project was initiated to demonstrate the complete lifecycle of API development, from initial problem understanding through deployment and documentation. The primary motivations include:

- **Educational Value:** Providing a comprehensive example of modern API development practices.
- **Practical Application:** Creating a reusable messaging system template for real-world applications.
- **Technical Excellence:** Demonstrating industry-standard patterns and best practices.
- **Complete Coverage:** Addressing all aspects of software development including testing, security, and deployment.

1.3 Project Objectives

The Messaging API project aims to achieve the following objectives:

1. **Secure Authentication System:** Implement JWT-based authentication with password hashing and token management.
2. **Messaging Functionality:** Enable users to send, receive, and manage messages in threaded conversations.
3. **Clean Architecture:** Demonstrate separation of concerns with organized code structure.
4. **Comprehensive Testing:** Achieve high test coverage with unit and integration tests.
5. **Production Readiness:** Support containerized deployment and environment-based configuration.

2 Related Work and Technologies

2.1 Existing Messaging Systems

Several messaging systems and APIs have influenced this project's design:

Slack API Slack's REST API provides comprehensive messaging functionality with real-time capabilities through WebSockets. Key features include channel-based communication, direct messages, and rich message formatting. Our implementation adopts similar patterns for conversation management and message threading.

WhatsApp Business API WhatsApp's business API demonstrates secure end-to-end encrypted messaging at scale. While our implementation doesn't include encryption, the conversation model and message delivery patterns are inspired by WhatsApp's architecture.

2.2 Frameworks and Technologies

FastAPI Framework FastAPI is a modern, high-performance Python web framework that provides automatic API documentation, built-in data validation, and async support. It has gained significant adoption in the Python ecosystem due to its developer-friendly design and production-ready features. Our choice of FastAPI enables rapid development while maintaining code quality.

SQLAlchemy ORM SQLAlchemy is the de-facto standard for database interactions in Python applications. Its ORM layer provides database abstraction, migration support, and relationship management. Version 2.0+ introduces improved typing and async support, which we leverage in this implementation.

JWT Authentication JSON Web Tokens have become the standard for stateless authentication in RESTful APIs. Libraries like `python-jose` and `passlib` provide battle-tested implementations of JWT encoding/decoding and password hashing, which form the foundation of our authentication system.

SendGrid Email API SendGrid is a cloud-based email delivery platform that provides reliable transactional email services. It offers a comprehensive REST API for sending emails programmatically with features like email templates, delivery tracking, and analytics. Our implementation integrates SendGrid for user notifications, including welcome emails and message alerts. The API provides excellent deliverability rates, spam protection, and detailed delivery reports, making it ideal for production applications that require reliable email communications.

2.3 Gap Analysis

While existing solutions provide excellent messaging capabilities, they often:

- Require significant infrastructure investment (Slack, Twilio).
- Lack educational focus with accessible source code.
- Don't demonstrate complete development lifecycle.
- Have complex setup procedures.

Our implementation addresses these gaps by providing a self-contained, well-documented system that demonstrates the entire development process from design to deployment.

3 System Analysis and Requirements

3.1 Functional Requirements

FR1: User Management

- FR1.1: Users must be able to register with username, email, and password.
- FR1.2: System must validate email format and password strength.
- FR1.3: System must prevent duplicate usernames and emails.
- FR1.4: Users must be able to authenticate and receive access tokens.

FR2: Messaging

- FR2.1: Authenticated users must be able to send messages to other users.
- FR2.2: Users must be able to retrieve their sent and received messages.
- FR2.3: Messages must support pagination to handle large message volumes.
- FR2.4: Users must be able to filter messages by conversation.

FR3: Conversations

- FR3.1: System must automatically create conversations when first message is sent.
- FR3.2: Users must be able to list all their conversations.
- FR3.3: System must prevent duplicate conversations between same users.
- FR3.4: Conversations must track last activity timestamp.

FR4: Message Status

- FR4.1: Messages must have read/unread status.
- FR4.2: Only receivers can mark messages as read.

3.2 Non-Functional Requirements

NFR1: Security

- NFR1.1: Passwords must be hashed using Bcrypt with appropriate cost factor.
- NFR1.2: API must use JWT tokens with 30-minute expiration.
- NFR1.3: Endpoints must enforce authentication where required.
- NFR1.4: System must prevent SQL injection through ORM usage.

NFR2: Performance

- NFR2.1: API response time must be under 200ms for standard queries.
- NFR2.2: System must support pagination for large datasets.
- NFR2.3: Database queries must use appropriate indexes.

NFR3: Usability

- NFR3.1: API must provide clear error messages.
- NFR3.2: Documentation must be auto-generated and interactive.
- NFR3.3: HTTP status codes must follow REST conventions.

NFR4: Maintainability

- NFR4.1: Code must follow clean architecture principles.
- NFR4.2: Test coverage must exceed 80%.
- NFR4.3: Code must include appropriate documentation.
- NFR4.4: System must use environment-based configuration.

NFR5: Deployability

- NFR5.1: Application must support Docker containerization.
- NFR5.2: Database migrations must be version-controlled.
- NFR5.3: System must work with both SQLite and PostgreSQL.

3.3 Use Case Diagram

The following use case diagram illustrates the main interactions between users and the messaging system.

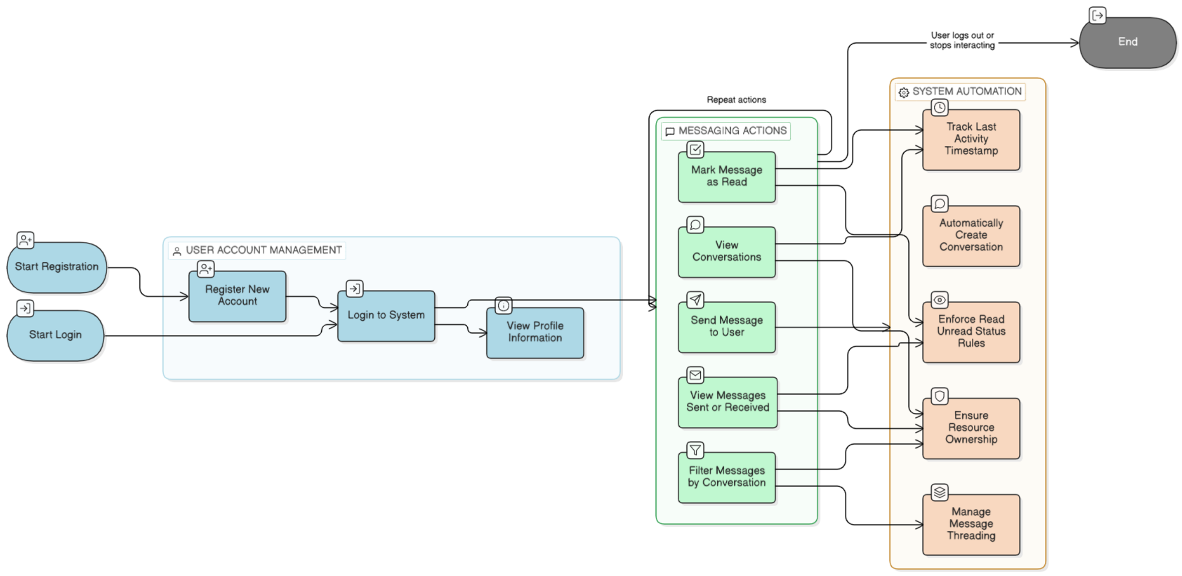

Figure 1: Use case diagram for ConvoLink.

3.4 Class Diagram

The following class diagram shows the database model structure and relationships.

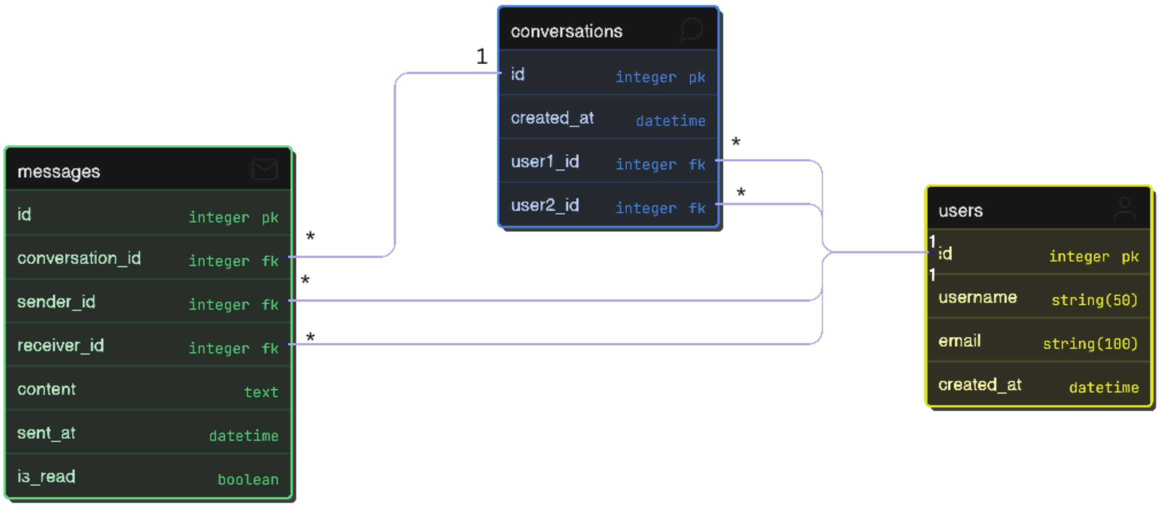

Figure 2: Use case diagram for ConvoLink.

4 System Design and Architecture

4.1 System Architecture

The Messaging API follows a layered architecture pattern with clear separation of concerns:

1. **Client Layer:** Frontend applications, API clients (Insomnia), and web browsers.
2. **Presentation Layer (Routers):** API endpoint definitions for authentication, messages, and conversations.
3. **Business Logic Layer (Services):** Core application logic including *UserService*, *MessageService*, *ConversationService*, and *EmailService*.
4. **Data Access Layer (Models + ORM):** SQLAlchemy models for *User*, *Message*, and *Conversation* entities.
5. **Database Layer:** SQLite for development and PostgreSQL for production environments.

This layered approach ensures clear separation of concerns, making the codebase maintainable, testable, and scalable.

4.2 API Design

The API follows RESTful design principles with resource-based URLs and HTTP method semantics.

Authentication Endpoints

Method	Endpoint	Description	Auth
POST	/api/v1/auth/register	Register new user	No
POST	/api/v1/auth/login	Login and get JWT token	No
GET	/api/v1/auth/me	Get current user info	Yes

Message Endpoints

Method	Endpoint	Description	Auth
POST	/api/v1/messages	Send a message	Yes
GET	/api/v1/messages	List user messages	Yes
GET	/api/v1/messages/{id}	Get specific message	Yes
PATCH	/api/v1/messages/{id} /read	Mark message as read	Yes

Conversation Endpoints

Method	Endpoint	Description	Auth
GET	/api/v1/conversations	List user conversations	Yes
GET	/api/v1/conversations/{id}	Get conversation details	Yes
GET	/api/v1/conversations/{id} /messages	Get conversation messages	Yes

5 Technical Solution and Implementation

5.1 Security Architecture

5.1.1 Authentication Flow

User Registration Process

1. Client submits registration data (username, email, password).
2. Server validates email format and password strength.
3. System checks for duplicate username/email.
4. Password is hashed using Bcrypt (cost factor 12).
5. User record created in database.
6. Response returns user data (excluding password).

Login Process

1. Client submits credentials (username, password).
2. Server retrieves user by username.
3. Password verified against stored hash.

4. JWT token generated with 30-minute expiration.
5. Token includes username in payload (sub claim).

Authenticated Request Process

1. Client includes JWT token in Authorization header (Bearer scheme).
2. Server validates token signature and expiration.
3. User identity extracted from token payload.
4. User record retrieved from database.
5. Request processed with user context.
6. Response returned with requested data.

5.1.2 Security Measures

Password Security

- Hashing Algorithm: Bcrypt with cost factor 12.
- Salting: Automatic unique salt for each password.
- Storage: Only hashed passwords stored, never plain text.
- Validation: Minimum 8 characters required.
- Transmission: HTTPS recommended for production.

Token Security

- Algorithm: HS256 (HMAC with SHA-256).
- Expiration: 30-minute token lifetime.
- Secret Key: Stored in environment variables.
- Payload: Contains only username (no sensitive data).
- Validation: Signature and expiration checked on every request.

Input Validation

- Email Validation: RFC-compliant email format checking.
- Type Safety: Pydantic enforces type constraints.
- Length Limits: Username (3–50), content (1–5000 characters).
- Sanitization: SQL injection prevented through ORM.
- XSS Protection: JSON-only responses prevent script injection.

Authorization

- Endpoint Protection: JWT required for all non-public endpoints.
- Resource Ownership: Users can only access their own data.
- Mark as Read: Only message receivers can mark messages as read.
- Conversation Access: Only participants can view conversation.

5.2 External API Integration

SendGrid Email Service The system integrates with SendGrid as an external API for reliable email delivery:

- API Communication: RESTful HTTP requests to SendGrid API endpoints.
- Authentication: API key stored securely in environment variables.
- Email Templates: Dynamic content generation for different email types.
- Error Handling: Graceful fallback when email service is unavailable.

Email Notification Types:

1. **Welcome Emails:** Sent upon successful user registration; contains account confirmation and getting started information; personalized with username and registration details.
2. **Message Notifications:** Triggered when a user receives a new message; includes sender information and message preview; configurable per-user notification preferences (future enhancement).

6 Technical Solution and Implementation

6.1 Project Structure

The project follows clean architecture principles with logical separation of concerns:

- `app/` – core application logic
- `routers/` – API endpoints
- `services/` – business logic
- `models/` – database entities
- `schemas/` – request/response validation
- `tests/` – unit and integration tests
- `frontend/` – basic web interface
- `alembic/` – database migrations

7 Conclusion and Discussion

The Messaging API project successfully demonstrates that modern Python technologies combined with industry best practices can produce production-ready applications suitable for both educational purposes and real-world deployment. The clean architecture, comprehensive testing, and thorough documentation make it an excellent reference for developers learning API development or building similar systems.

This project proves that building a professional-grade API doesn't require complex infrastructure or overwhelming complexity. With the right tools, architectural decisions, and attention to best practices, even academic projects can achieve production-level quality and serve as valuable learning resources.

The Messaging API stands as a testament to modern software engineering practices, demonstrating that careful planning, clean implementation, and thorough testing can result in a robust, secure, and scalable application ready for real-world use.