PHENIKAA UNIVERSITY

SCHOOL OF COMPUTING

Course Project Report: Microservice Architecture for Real-Time Chat Application using Event-Driven and gRPC

Student Details: Hoàng Lê Đức Huy - 23010298 - Group 8

Nguyễn Hà Nguyên - 23010310 - Group 8

Nguyễn Đức Minh - 23010302 - Group 8

Course Info: Software Architecture

Ha Noi, January 22, 2026

Table of Contents

List of Tables

List of Figures

List of Listings

1. Cover Page & Info

- Project title: Microservice Architecture for Real-Time Chat Application using Event-Driven and gRPC
- Course Name: Software Architecture
- Student Info: Nguyễn Hà Nguyên 23010310, Hoàng Lê Đức Huy 23010298, Nguyễn Đức Minh 23010302
- Date: 22/01/2026

2. Executive summary

This project presents the engineering process for building and optimizing a high-performance Real-time Chat Application, based on a modern Microservices architecture and Event-Driven model. The system is developed using NestJS for backend services, ReactJS for the user interface, and MongoDB as the database to ensure storage flexibility. The architectural highlight lies in the decoupled Gateway strategy, including a specialized Real-time Gateway for WebSocket connections and a custom-built API Gateway. This API Gateway serves as the central hub for system orchestration, directly handling authentication and communicating with internal services via the gRPC protocol to optimize bandwidth and minimize latency to the lowest possible level.

To address the challenges of scalability and reliability, the system integrates RabbitMQ as the central Message Broker, ensuring service decoupling and efficient processing of asynchronous tasks. Notably, the project infrastructure leverages the power of Redis as a key component for managing state and socket connection lists in a distributed environment, while also serving as a fast-retrieval store for the Rate Limiting mechanism at the Gateway. The tight integration of these technologies creates a sustainable overall solution, helping the system maintain stability under heavy traffic and providing a smooth real-time user experience.

3. Project Requirements & Goals

3.1 Core Functional Requirements

Table 1: Core Functional Requirements of the System

ID	Description
FR-01	The application must allow users to register accounts using a valid email and password.
FR-02	The system must verify the user's email before activating the account.
FR-03	The application must allow users to log in using their email and password..

FR-04	The system must use JWT or token-based authentication to authenticate requests after login.
FR-05	The application must allow users to log out and revoke valid tokens.
FR-06	Users must be able to search for other users via email or username.
FR-07	Users must be able to send friend requests via email or username.
FR-08	The system must send real-time notifications when a user receives a friend request.
FR-09	Friend request recipients must be able to Accept or Reject the invitation.
FR-10	When an invitation is accepted, the system must automatically add both users to each other's friend list.
FR-11	The application must allow users to send 1-1 messages (private messages) to their friends.
FR-12	The system must ensure 1-1 messages are transmitted and received using a real-time mechanism (WebSocket).
FR-13	The application must allow users to create Group Chats (n-n).
FR-14	The group creator must be able to add friends to the group.
FR-15	The system must allow sending and receiving group messages via a real-time broadcasting mechanism..
FR-16	When a new message arrives in a conversation (1-1 or group), the system must send real-time notifications to relevant members.
FR-17	The system must store message history and allow users to reload it when reopening a conversation.
FR-18	The application must display user status (online / offline / typing...) in real-time.
FR-19	The system must ensure that only members within a conversation have the right to access message content.
FR-20	The system must handle multiple simultaneous connection sessions and ensure messages are delivered accurately to the corresponding clients.

3.2 Key Quality Attributes (Architectural Goals - Non-functional Requirements)

Table 2: Key Quality Attributes of the System

ID	Attributes	Description	Constraint

QA-01	Performance (Low Latency))	The system must minimize the latency from the moment a message is sent until the recipient sees it. This is the core element of the "Real-Time" feature.	End-to-End Latency: < 200ms for message transmission under normal load conditions.
QA-02	Reliability & Availability	The system must ensure the chat service is always available and messages are not lost during transmission, even if a sub-service encounters an issue.	Uptime: 99.9%. Message Delivery: Ensure "At-least-once delivery" mechanism via Event-Driven architecture.
QA-03	Data Consistency	The order of messages must be strictly preserved. Conversation history must display in the correct chronological order for all participants.	Messages must be displayed accurately according to the timestamp generated at the server.
QA-04	Security	Communication must be protected to prevent eavesdropping. Only users authenticated with a valid JWT can access the system..	Authentication: Strict JWT verification at every gRPC/API call. Access Control: Users cannot view messages from groups they have not joined.
QA-05	Maintainability	The system must be loosely coupled to allow individual services to be updated without affecting the entire system.	Services communicate entirely via gRPC interfaces or asynchronous Events; no direct database sharing between services.
QA-06	Interoperability	Communication between microservices must achieve peak efficiency to minimize internal network latency.	Protocol: Use gRPC (Protobuf) for internal communication instead of traditional REST JSON to reduce payload size by 30-40%.

3.3 Proposal model of ASR

Table 3: Architectural Significant Requirements (ASR) and Impact on Architectural Design

ASR	Statement	Impact
ASR-01	Real-time Latency: The system must ensure that 1-1 and Group messages are delivered to recipients with minimal latency (near-instant) without requiring the user to reload the page.	Use of WebSockets: Mandatory replacement of traditional HTTP Requests with persistent WebSocket connections, allowing the server to proactively push messages to clients immediately.
ASR-02	Inter-service Efficiency: Since the system is divided into multiple services, internal communication must be extremely fast and lightweight to avoid slowing down the total response time.	Use of gRPC (Protobuf): Replace REST API (JSON) with gRPC for internal (Service-to-Service) communication to reduce payload size and accelerate data serialization/deserialization.
ASR-03	Decoupling & Extensibility: Auxiliary functions (such as sending notifications, sending emails, saving history) must not affect or slow down the core messaging flow.	Event-Driven Architecture: Utilize a Message Broker (e.g., Kafka/RabbitMQ) to decouple services. The chat service only needs to emit a "MessageSent" event, while other services automatically listen and process it asynchronously.
ASR-04	Data Integrity & Ordering: Messages in a conversation must be displayed in the correct chronological order for all participants, regardless of varying network latencies.	Consistent Timestamping: Design the database and backend logic to store accurate timestamps at the moment the server receives the message, ensuring consistent sorting when clients reload history.
ASR-05	Security & Isolation: Users must strictly not be able to access or eavesdrop on messages from chat groups of which they are not members.	Token-based Auth & Middleware: Implement authentication middleware at the Gateway and each Service. Use JWTs containing User ID information to verify ownership (Authorization) before allowing a user to join a WebSocket channel.

3.4 Use Case Modeling

Table 4: Use Case Model of the System

Use Case	Actor	Flow Description
Sign Up	User	The user enters personal information (email, password, etc.) to request a new account. The system validates and stores the information.
Sign In	User	The user enters credentials. The system verifies them and, if correct, grants access to the application.
Logout	User	The user requests to end the session. The system invalidates the authentication token and redirects the user to the login screen.
Send Message	User, Admin (in a group)	The user composes and sends content. • Include: The system automatically triggers Authenticate user and Validate message before sending. • Extend: Users can optionally Send file/media, Reply to message, or Forward message.
Receive Message	User, Admin(in a group)	The system listens for new message events and displays the incoming content on the user interface in real-time.
Create Group Chat	User, Admin(in a group)	The user initializes a new group conversation. • Include: This process must include the Add members step to complete group creation.
Join Group Chat	User	The user accepts an invitation or actively joins an existing chat group via a link or group code.
Add Friends	User, Admin(in a group)	The user searches for and sends friend requests to others or accepts incoming friend requests.
Block Members	User, Admin(in a group)	The user chooses to block another account to prevent receiving messages or interactions from that account.

3.5 API Gateway

3.5.1 API Gateway

- Function: Receives HTTP/REST requests from the Client and serves as the outer security layer for the system.
- Key Features:

- Business Logic Handling: Performs tasks such as registration, login, user data queries, and system configuration.
- Internal Orchestration: Utilizes gRPC to communicate with Microservices (User, Chat, Notification). Data from gRPC Observable streams is converted into Promises to remain compatible with Client HTTP responses.
- Authentication: Employs JWT (JSON Web Token) to control access via AuthGuard.
    - Supports an Auth Bypass mechanism using Metadata (Decorator @Public) for endpoints that do not require login.
    - Post-authentication user info (e.g., userId, username) is attached directly to the Request object for use by downstream services.
- Rate Limit:
    - Integrates RateLimitGuard to prevent Brute-force attacks and request spamming.
    - Uses a dual-identification strategy: prioritizes userId for logged-in users and IP address for guests.
- Redis: Acts as a distributed cache to store counters for Rate Limiting. Utilizing Atomic Operations (such as INCR and EXPIRE) on Redis ensures accuracy and high performance in a distributed, multi-instance environment.

3.5.2 Realtime Gateway

- Function: Manages WebSocket (Socket.io) connections between the Client and the Server.
- Key Features:
    - Stateful: Keeps track of online users and maintains persistent, open connection pipes.
    - Bi-directional Interaction: Enables the Server to proactively push data to the Client immediately without a client-side request (e.g., new messages, notifications).
    - Event-driven: Processes data based on events such as sendMessage or user-online. It integrates with RabbitMQ to listen for and distribute events from the Chat Service to the end-users.
    - Redis: Uses Redis to store the mapping between a userId and its list of active socketIds.
        - Resource Optimization: When a user disconnects, the system automatically removes the corresponding socketId from Redis to free up memory.

4. Architecture Design & Implementation

4.1 Architecture Pattern: Event-Driven Architecture (EDA) via WebSockets

The core of the system is built upon Event-Driven Architecture (EDA). This is the optimal model for real-time chat applications, where the Server functions as a message distribution and orchestration hub.

Specific Operational Mechanism:

- The Server listens for and responds to events initiated by the Client (e.g., the 'send message' event).
- Immediately, the Server processes and broadcasts new events (e.g., 'chat message') to all other connected Clients.
- Implementing EDA allows the system to perfectly meet critical technical requirements: bi-directional communication, maintaining persistent connections, and ensuring ultra-low latency.

C4 Model

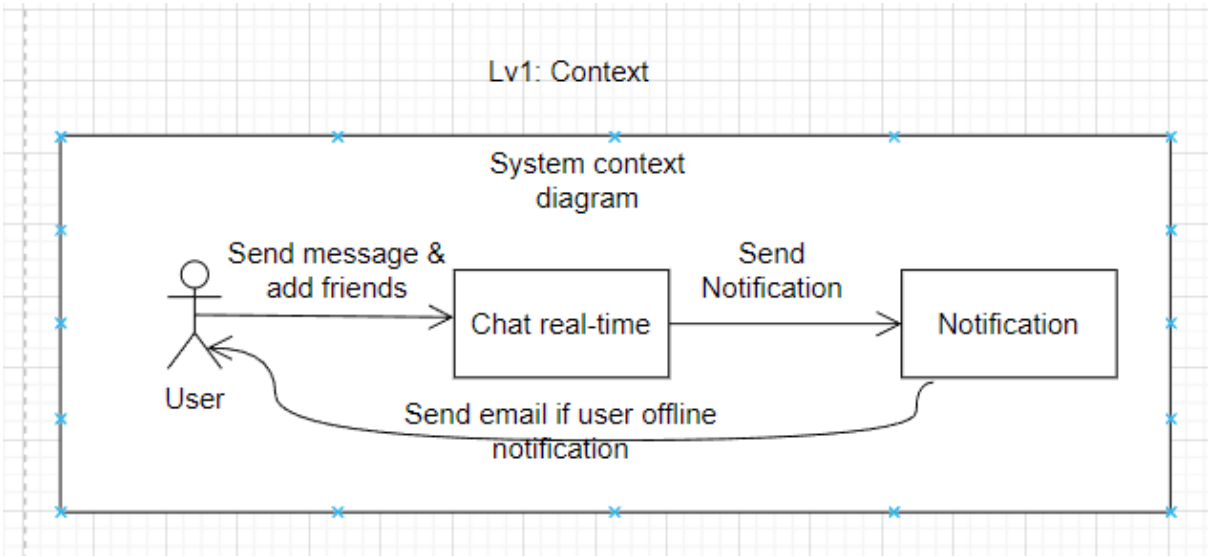

Figure 1: Level 1: System Context

The User/Client interacts with the system through a persistent WebSocket connection. No other external systems (such as third-party databases or external APIs) are required for the core functionality, which keeps the design streamlined and focused entirely on real-time messaging.

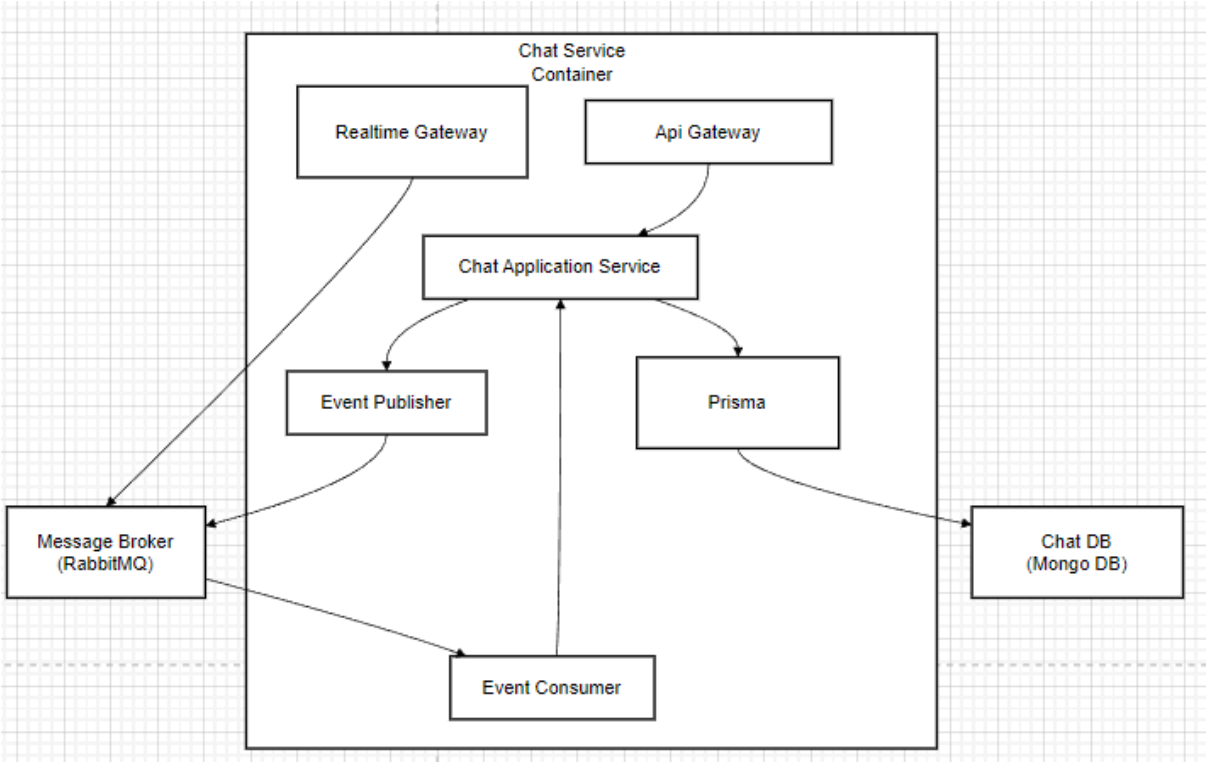

Figure 2: Level 2: Container

The Level 2 Container Diagram delves into the internal architecture of the Chat Service (the core component of the Level 1 diagram). This model is designed using a Microservices approach combined with Event-Driven Architecture (EDA) to ensure real-time processing capabilities and data integrity.

The Chat Service operates as an independent container, responsible for handling all business logic related to sending/receiving messages, state management, and data synchronization.

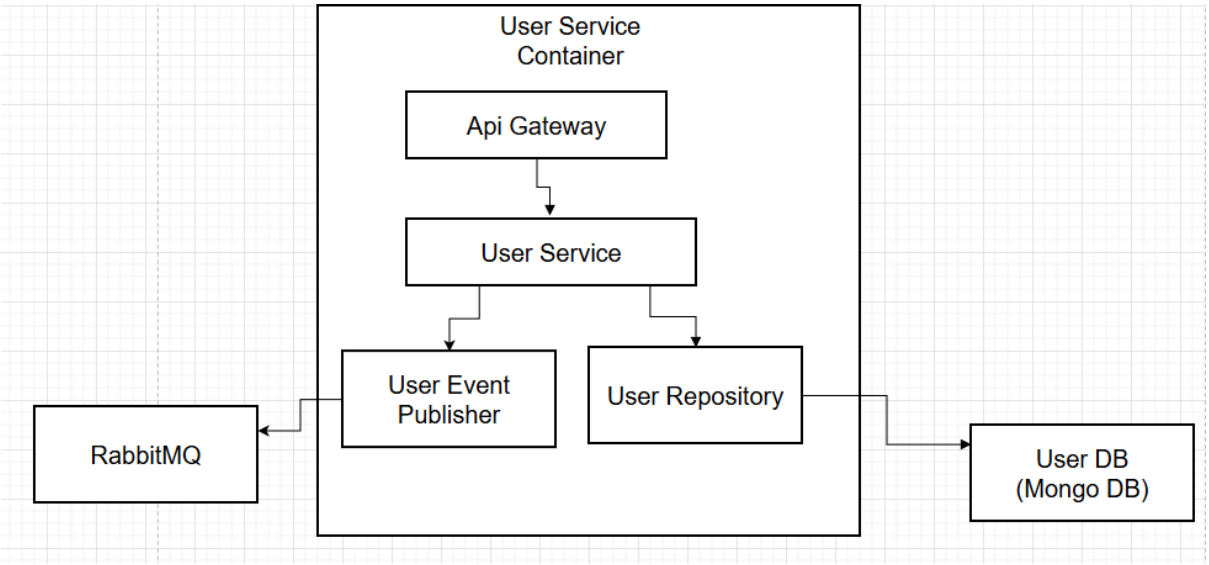

Figure 3: Level 2:User Service Container

The Level 2 Container Diagram details the internal structure of the User Service, the component responsible for managing user identities and profiles within the system. This service is designed following the principles of data encapsulation and separation of concerns. Operating as an independent container, the User Service focuses on handling business logic such as registration, profile updates, and authentication. Through the User Repository and User DB (MongoDB), it ensures data integrity and security, while the User Event Publisher broadcasts state changes to the rest of the system via RabbitMQ, facilitating seamless data synchronization across the microservices architecture.

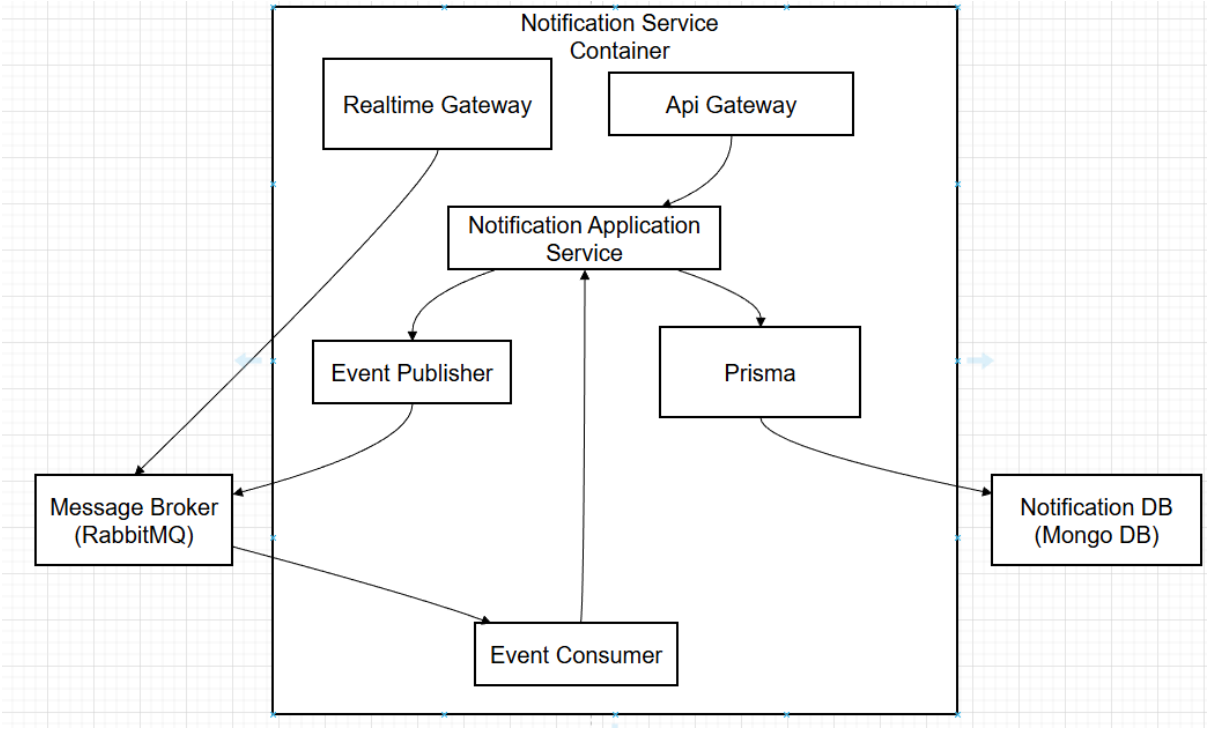

Figure 4: Level 2:Notification Service Container

This diagram illustrates the internal architecture of the Notification Service, a key component for maintaining real-time interaction and delivering alerts to users. Similar to the Chat Service, this model utilizes an Event-Driven Architecture (EDA) to optimize responsiveness. The Notification Service operates as a standalone container, employing an Event Consumer to listen for triggers from the Message Broker (RabbitMQ), which then initiates the relevant notification workflows. The combination of the Realtime Gateway and Notification Application Service enables the delivery of instant push notifications, while Prisma and the Notification DB handle the storage of notification history and states, ensuring a consistent and timely user experience.

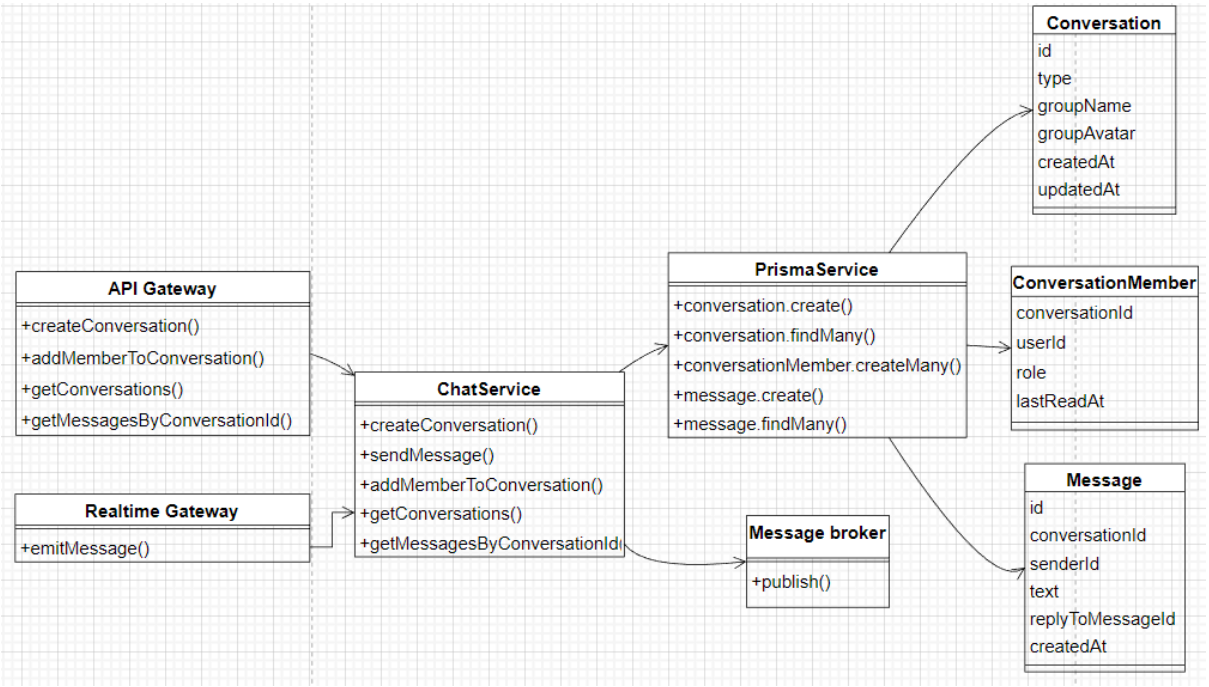

Figure 5: Level 3:Chat Service Components

This diagram illustrates the source code structure and detailed data flow within the Chat Application Service. It clarifies how user requests are transformed into specific actions through the processing layers (Services), the data access layer (Prisma/Repository), and the table structures (Entities).

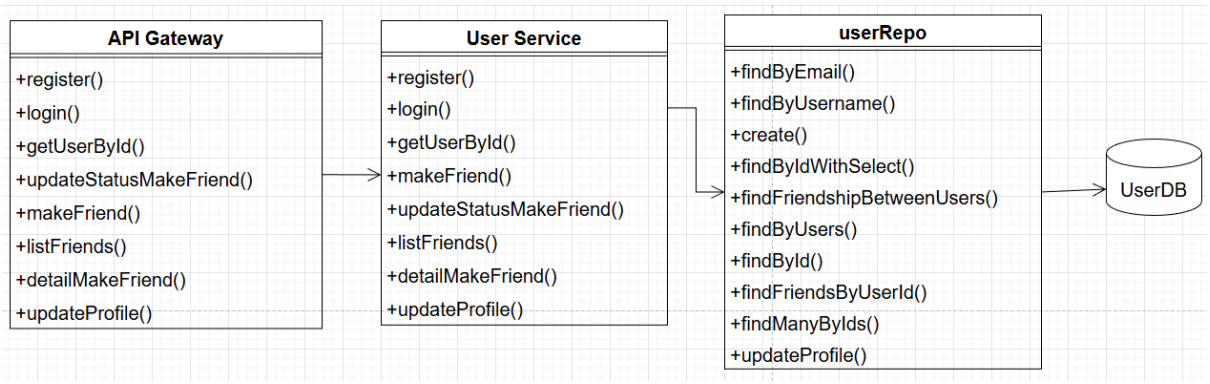

Figure 6: Level 3:User Service Components

This diagram illustrates the source code structure and detailed data flow within the User Service Component. It clarifies how user requests—such as registration, login, and friend management—are transformed into specific actions through the business logic layer (User Service), the data access layer (User Repository), and ultimately interacting with the table structures within the UserDB.

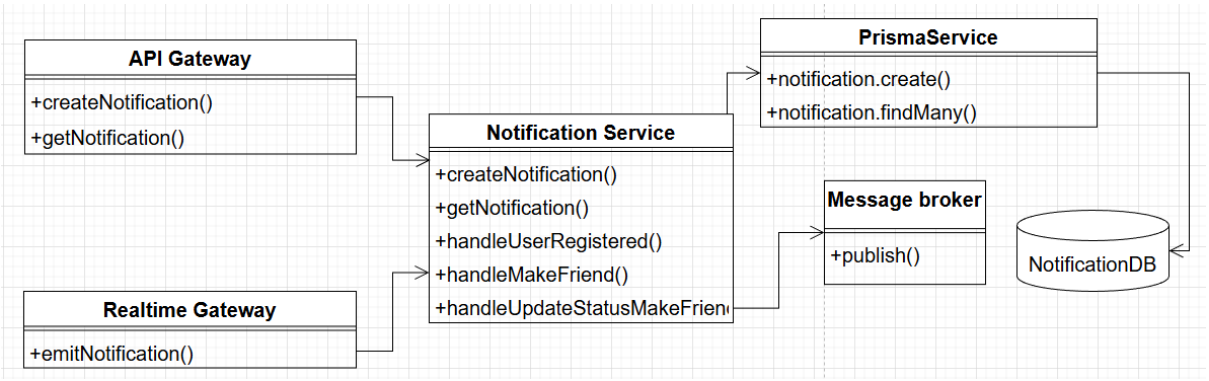

Figure 7: Level 3:Notification Service Components

This diagram provides a detailed view of the internal architecture of the Notification Service Component, focusing on the processing and distribution of alerts. It outlines how the service captures system events (such as new friendships or registrations) via the API Gateway and Realtime Gateway, processes the logic within the Notification Service layer, and utilizes the Prisma Service to manage notification data in the NotificationDB while coordinating with the Message Broker for synchronization.

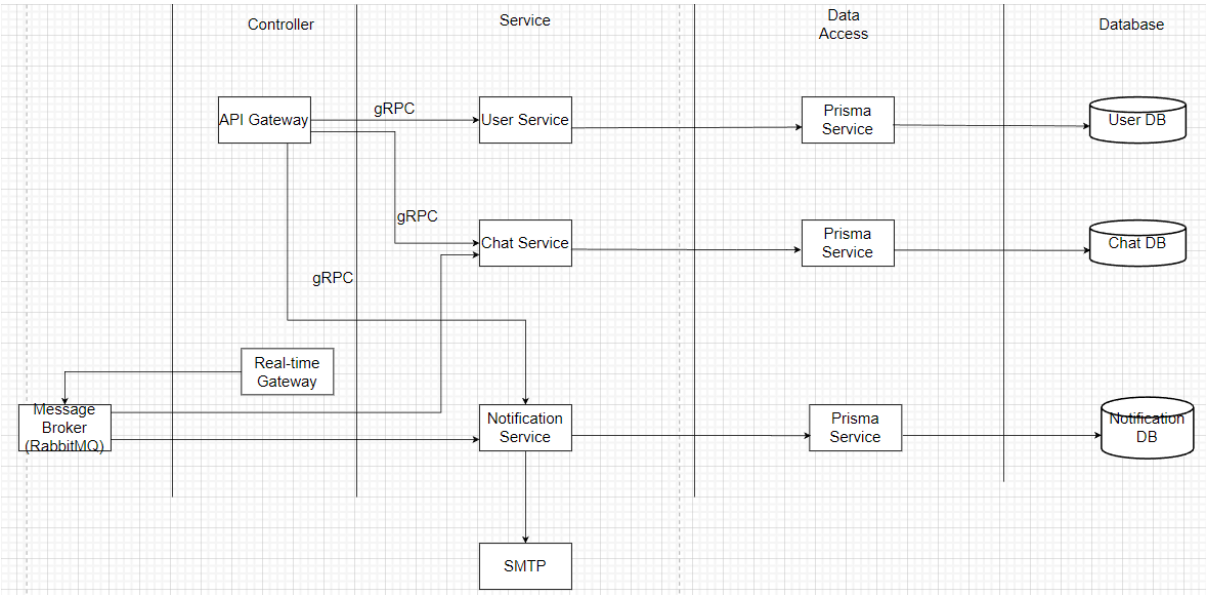

Figure 8: Level 4: Code

The Level 4 Diagram illustrates the overall system architecture based on a Microservices model combined with Layered Architecture. The system is divided into vertical processing layers and horizontal business blocks (Services), ensuring the principles of "Separation of Concerns" and "Database-per-service".

4.2 Deployment Diagram

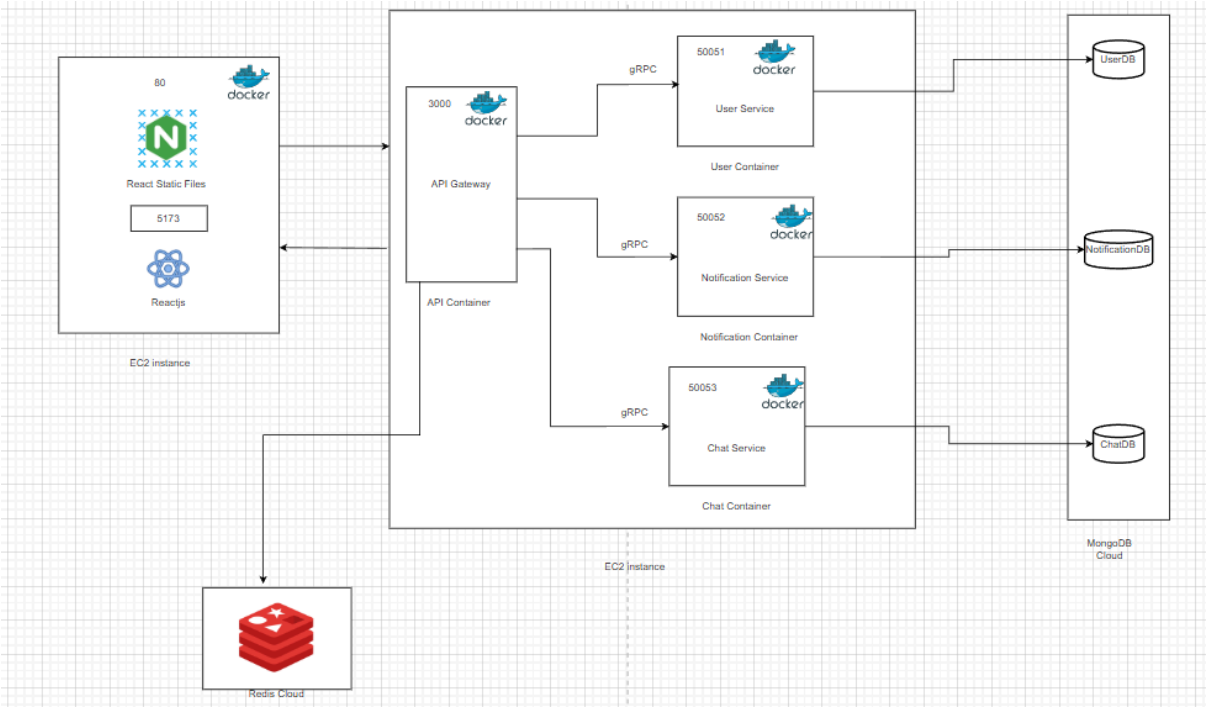

Figure 9: Deployment Diagram

4.3 Technical Stack and Data Model

Table 5: Technical Stack and System Components

Component	Technology	Role & Function
Language	TypeScript	Ensures Type-safety consistency between Frontend and Backend. Helps detect errors early during development, optimizes source code structure, and supports defining strict Interfaces for gRPC messages.
Server	NestJS	Builds the Server-side architecture following a Modular approach, efficiently managing Microservices and handling business logic.
Real-time	Socket.io	Establishes and maintains stable bi-directional connections between the browser and the Real-time Gateway to power instant chat features.
Client	ReactJS	Builds a flexible User Interface (UI), optimizing component reusability and managing browser-side logic.
Internal Communication	gRPC	A high-performance binary transport protocol used for communication between the Gateway and Microservices.

State Management	Redux	Stores and manages centralized data for the entire Web application (e.g., user info, conversation lists), ensuring data consistency across all components.
UI Library	shadcn/UI	Provides a set of standardized UI Components, ensuring a modern, professional, and consistent Web interface.
ORM & Database	Prisma & MongoDB	Prisma acts as an intermediary layer to query data from MongoDB safely via TypeScript, replacing raw queries.
Queue & Cache	RabbitMQ & Redis	RabbitMQ orchestrates messages between services; Redis stores online status and implements Rate Limiting to protect the system.

4.4 Implementation: Server-Side Logic (NestJS/Typescript)

4.4.1 API Gateway & gRPC Integration

The API Gateway serves as the single entry point for the system's HTTP requests from clients. Upon receiving a request, the Gateway does not process business logic directly but forwards it to specialized microservices via gRPC.

The workflow includes several steps: receiving the REST request, mapping data from the DTO to the gRPC request structure, invoking the corresponding service through a gRPC Client, and converting the returned result from an Observable to a Promise to respond to the client. This approach allows the Gateway to maintain a purely orchestrating role and avoids overloading it with business logic.

Listing 1: API Gateway Implementation & gRPC Integration Logic

```
// [File: auth.controller.ts] - API Gateway Entry Point
@Controller('auth')
export class AuthController {

  constructor(private readonly authService: AuthService) {}

  // 1. Receiving HTTP Requests (REST) from Clients
  @Post('registe     r')
  async register(@Body() dto: RegisterUserDto) {
    // Forwarding to the Service for processing
    return await this.authService.register(dto);
  }
}

// [File: auth.service.ts] - API Gateway Logic
@Injectable()
export class AuthService {
```

```
  // Inject the gRPC Client configured in the module
  @Inject('USER_PACKAGE')
  private readonly userGrpcClient: ClientGrpc;

  private userService: IUserService;

  onModuleInit() {
    // Initialize the service proxy from the gRPC Client
    this.userService =
this.userGrpcClient.getService<IUserService>('UserService');
  }

  async register(dto: RegisterUserDto):
Promise<UserRegisterResponse> {
    // 2. Mapping data from DTO to gRPC Request Interface
    const payload: UserRegisterRequest = {
      email: dto.email,
      password: dto.password,
      username: dto.username,
    };

    // 3. Executing the gRPC call:
    // Convert Observable (gRPC's Reactive Stream) into a Promise
    // to synchronize with the HTTP Controller's execution flow.
    const observable = this.userService.register(payload);

    return await firstValueFrom(observable);
  }
}
// [File: user.pb.ts] - Generated by ts-proto
// Interface representing the data structure sent via gRPC

export interface UserRegisterRequest {
sample
}
```

- Flow:

1. When a client sends a POST /auth/register request, the AuthController receives the HTTP request and passes the DTO data to AuthService.register() for processing.
2. Inside the AuthService, the data from the DTO is mapped to the UserRegisterRequest structure, adhering to the gRPC contract.
3. The service then invokes the register() method of the UserService via the gRPC client.
4. Since gRPC in NestJS returns an Observable (reactive stream), it is converted into a Promise using firstValueFrom() to align with the standard HTTP request-response model.
5. Once the User Microservice finishes processing and returns the response via gRPC, the result is sent back to the Controller and delivered to the client as an HTTP response.

4.4.2 Gateway Security: Distributed Rate Limiting

To protect the system against spam or brute-force behavior, a Distributed Rate Limiting mechanism is implemented at the API Gateway. Redis is utilized as a distributed cache to store request counters per user or IP address.

The use of atomic operations such as INCR and EXPIRE ensures accuracy and safety across multiple Gateway instances. When the number of requests exceeds the permitted limit, the system returns an HTTP 429 (Too Many Requests) error to prevent resource abuse.

Listing 2: Distributed Rate Limiting Guard using Redis Atomic Operations

```
// [File: rate-limit.guard.ts] - Distributed Rate Limiting Guard
@Injectable()
export class RateLimitGuard implements CanActivate {

  constructor(
    private reflector: Reflector,
    @Inject('REDIS_CLIENT') private redisClient: Redis, // Inject
Redis Connection
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean>
{
    // 1. Retrieve the limit configurations (Limit & TTL) from
the Endpoint Metadata
    const options = this.reflector.get(RATE_LIMIT_OPTIONS_KEY,
context.getHandler());
    const limit = options?.limit || 100; // Default 100 requests
    const ttl = options?.ttl || 60;      // Default 60 giây

    // 2. Identify the user (Hybrid Strategy)
    // Prioritize UserID (if logged in); otherwise, use the IP
Address
    const req = context.switchToHttp().getRequest();
    const identifier = req.user?.userId ?
`user:${req.user.userId}` : `ip:${req.ip}`;
    const key = `throttle:${identifier}`;

    // 3. Increment the counter in Redis (Atomic Operation)
    // INCR: Increments the value by 1 and returns the new value.
Extremely fast and safe in a distributed environment.
    const current = await this.redisClient.incr(key);

    // If it is the first request in the cycle, set the
expiration time (TTL)
    if (current === 1) {
      await this.redisClient.expire(key, ttl);
    }

    // 4. Check the limit
    if (current > limit) {
      throw new HttpException({
```

```
        message: 'Too many requests, please try again later.',
        retryAfter: ttl
      }, HttpStatus.TOO_MANY_REQUESTS);
    }
    return true; // Allow the request to proceed
  }
}
```

- Flow:

1. When a request is sent to an endpoint with the RateLimitGuard applied, the canActivate() method is executed before the request reaches the Controller.
2. First, the Guard retrieves the limit configuration (limit and ttl) from the endpoint's metadata via the Reflector. If no specific configuration exists, the system uses default values, such as 100 requests per 60 seconds.
3. Next, the system identifies the requester. If the user is logged in, the system uses the userId as the identifier; otherwise, it uses the IP address. This generates a Redis key in the format throttle:user:123 or throttle:ip:192.168.1.1.
4. Then, the Guard executes the INCR command in Redis to increment the counter by 1. As an atomic operation, this is safe to use in a distributed environment with multiple servers.
5. If this is the first request in the cycle (the returned value equals 1), the system sets the Time-To-Live (TTL) for that key.
6. Finally, if the current request count exceeds the permitted limit, the system throws a 429 Too Many Requests error. If the limit has not been reached, the request is allowed to proceed to the Controller.

4.4.3 Service Orchestration: gRPC & Authentication

The authentication mechanism is centrally implemented through a JWT Auth Guard at the Gateway. Every request, unless explicitly marked as public, must undergo a token validation step before being processed.

Once validated, the JWT provides the userId and username, which are attached directly to the request object. This allows downstream services to utilize user information without re-authenticating, reducing processing overhead and ensuring consistency across the entire system.

Listing 3: Centralized JWT Authentication Guard Implementation

```
// [File: auth.guard.ts] - Centralized JWT Guard
@Injectable()
export class AuthGuard implements CanActivate {
  @Inject(JwtService) private jwtService: JwtService;
  @Inject(Reflector) private reflector: Reflector;

  canActivate(context: ExecutionContext): boolean {
    // 1. Check 'without-login' Metadata (Auth Bypass Mechanism)
```

```
    // If the endpoint is marked with the @Public decorator,
allow passage without a token.
    const isPublic =
this.reflector.getAllAndOverride<boolean>('without-login', [
      context.getHandler(),
      context.getClass(),
    ]);

    if (isPublic) {
      return true;
    }

    // 2. Extract the Token from the Header
    const request = context.switchToHttp().getRequest();
    const authorization = request.headers.authorization;

    if (!authorization) {
      throw new UnauthorizedException('Missing token');
    }

    // 3. Verify Token & Attach User Information
    try {
      const token = authorization.split(' ')[1]; // Format:
"Bearer <token>"

      // Verify the token signature using the Secret Key
      const payload = this.jwtService.verify(token);

      // Attach user information to the Request Object for use by
Controllers/Services
      request.user = {
        userId: payload.userId,
        username: payload.username,
      };

      return true;
    } catch (error) {
      throw new UnauthorizedException('Invalid or Expired
Token');
    }
  }
}
```

- Flow:

1. When a request is sent to the system, the AuthGuard is executed before the request reaches the Controller.
2. First, the Guard checks the endpoint's metadata to see if it is marked as @Public (no login required). If the endpoint has an auth bypass decorator, the request is allowed to proceed without a token check.
3. If the endpoint requires authentication, the system extracts the Authorization Header from the request. If the header containing the token is missing, the system returns a 401 Unauthorized error.

4. If a token is present, the Guard extracts it from the "Bearer <token>" string and uses the JwtService to verify the signature and check for validity (such as expiration).
5. If the token is valid, the userId and username from the payload are attached to request.user so that downstream Controllers or Services can utilize them
6. If the token is invalid or expired, the system throws a 401 Unauthorized error.

4.4.4 Connection Management & State Storage (Redis)

The Realtime Gateway is responsible for managing the entire lifecycle of WebSocket connections. Upon a successful connection, the system performs JWT authentication right at the handshake step and stores the user's socketId in Redis.

Redis is utilized as a Set to support a single user connecting from multiple devices or browser tabs simultaneously. When a user disconnects, the corresponding socketId is removed from Redis, which helps free up resources and ensures an accurate online status.

Listing 4: WebSocket Connection Lifecycle & Redis State Management

```
// [File: realtime.gateway.ts] - Connection Logic
@WebSocketGateway({ namespace: 'realtime' })
export class RealtimeGateway implements OnGatewayConnection,
OnGatewayDisconnect {

  constructor(
    private jwtService: JwtService,
    // UserStatusStore is a wrapper class that interacts with the
Redis Set
    private userStatusStore: UserStatusStore
  ) {}

  // 1. Handling User Connections
  async handleConnection(client: Socket) {
    try {
      // Authenticate the Token from the Handshake
      const token = client.handshake.auth?.token;
      const payload = this.jwtService.verify(token);

      // 2. STORE STATUS (Redis):
      // Add the SocketID to the user's list of active sockets in
Redis
      // Redis Structure: Set<userId, [socketId1, socketId2]>
      await this.userStatusStore.addConnection(payload.userId,
client.id);

      // Notify that the User is Online
      this.server.emit('user_status', { userId: payload.userId,
status: 'online' });
    } catch (error) {
      client.disconnect();
    }
```

```
  }

  // 3. Handling User Disconnection
  handleDisconnect(client: Socket) {
    const userId = client.data.userId;
    // Remove SocketID from Redis to free up resources
    this.userStatusStore.removeConnection(userId, client.id);
  }
}
```

- Flow:

1. When a client connects to the realtime namespace, the handleConnection() method is executed.
2. First, the system extracts the JWT token from handshake.auth. This token is verified using JwtService.verify() to ensure the user is valid.
3. If the token is valid, the system saves the user's connection status to Redis via the UserStatusStore. Specifically, the socket.id is added to a Redis Set corresponding to the userId. This allows a user to have multiple simultaneous connections (e.g., multiple tabs or devices).
4. After successfully saving the status, the server broadcasts a user_status event to notify others that this user is now online.
5. If the token is invalid or an error occurs during authentication, the client is disconnected immediately.

4.4.5 Asynchronous Event Processing (RabbitMQ Integration)

RabbitMQ is utilized as the central Message Broker for the Event-Driven architecture. When the Chat Service triggers an event such as message_sent, this event is pushed into a queue, and the Realtime Gateway acts as the consumer to process it.

This mechanism completely decouples the primary message sending flow from the message delivery process to the client. This ensures the system does not bottleneck even during high spikes in message volume.

Listing 5: Event Consumer & Message Distribution Strategy via RabbitMQ

```
// [File: realtime.gateway.ts] - Event Consumer Logic
export class RealtimeGateway {

  // 1. Consumer: Listening for "New Message" events from the
Chat Service
  @RabbitSubscribe({
    exchange: 'CHAT_EVENTS',
    routingKey: 'message_sent',
    queue: 'realtime_messages_queue',
  })
  async handleNewMessageSent(payload: any): Promise<void> {
    // Payload contains: content, senderId, and a list of
receiver memberIds
```

```
    const receivers = payload.memberIds.filter(id => id !==
payload.senderId);

    // Call the helper function to distribute the message
    await this.emitToUser(receivers, 'new_message', payload);
  }

  // 2. Helper: Distribute the message to the list of Users
  async emitToUser(userIds: string[], event: string, data: any) {
    for (const userId of userIds) {
      // REDIS QUERY:
      // Retrieve all open sockets for the User (supporting
multiple devices)
      const sockets = await
this.userStatusStore.getUserSockets(userId);

      // Send the message to each socket
      sockets.forEach((socketId) => {
        this.server.to(socketId).emit(event, data);
      });
    }
  }
}
```

- Flow:

1. When the Chat Service emits a message_sent event into RabbitMQ (specifically through the CHAT_EVENTS exchange), the RealtimeGateway acts as a consumer and listens for this event using the @RabbitSubscribe decorator.
2. Once the event is received, the handleNewMessageSent() method is executed. The event payload includes the message content, the senderId, and a list of memberIds.
3. The system removes the senderId from the recipient list (as the sender usually has the message updated locally on their client). Then, it calls the emitToUser() helper function to distribute the message to the remaining users.
4. Inside emitToUser(), the system iterates through each userId in the recipient list
5. For each user, the system queries Redis via the UserStatusStore to retrieve the list of all active socketIds for that specific user.
6. Finally, the server sends the new_message event to each individual socket using this.server.to(socketId).emit().

4.5 Implementation: Client-Side Logic (ReactJS/Typescript)

4.5.1 WebSocket Lifecycle Management

On the client side, the WebSocket connection is managed centrally at the root component of the application. The connection is established only after the user has successfully logged in and is automatically terminated when the user logs out or closes the app.

This lifecycle management approach prevents resource leaks (zombie connections) and ensures that every active WebSocket connection is tied to a valid, authenticated user.

Listing 6: Client-Side WebSocket Lifecycle Management in React

```
// [File: App.tsx] - Client Main Entry
function App() {
  const user = useSelector(selectUser); // Retrieve User
information from Redux
  const dispatch = useDispatch();

  // 1. Connection Lifecycle
  // Automatically connect when User info is available and
disconnect when the Component unmounts
  useEffect(() => {
    if (user?.id) {
      // Attach Token to Handshake for Gateway authentication
      socket.auth = { token: user.token };
      socket.connect();
    }

    return () => socket.disconnect(); // Cleanup function
  }, [user]);

  // (Event Listeners are registered below...)
}
```

- Flow:

1. When the React application renders, the App component retrieves the user's information from the Redux Store.
2. The useEffect() hook monitors changes to the user state.
3. If a user.id exists (indicating the user is logged in), the client attaches the JWT token to socket.auth to be sent during the handshake process with the WebSocket Gateway.
4. The client then calls socket.connect() to establish the connection to the server.
5. When the component unmounts—for instance, when the user logs out or closes the application—the cleanup function within useEffect is executed. This function calls socket.disconnect() to close the connection properly, preventing resource leaks and ensuring the server can immediately clean up the session in Redis.

4.5.2 Event Listeners & State Updates

The client registers to listen for events emitted by the server, such as new messages, new conversations, or system notifications. Each event is handled by updating the Redux Store, which automatically synchronizes the user interface in real-time.

This approach ensures the UI remains consistent with the backend state and provides a smooth, seamless user experience.

Listing 7: Real-time Event Listeners & Redux State Synchronization

```
// 2. Real-time Event Handling Logic

 // A. Handling New Messages (Chat Message)
  socket.on('chat.new_message', (message) => {
    // Update message content in the current chat segment
    dispatch(addMessage(message));

    // Update the preview line (Last Message) in the Sidebar
    dispatch(updateNewMessage({
      conversationId: message.conversationId,
      lastMessage: message
    }));

    // UX: If the user is not currently viewing this
conversation, increase the unread count
    if (currentActiveChat !== message.conversationId) {
      dispatch(upUnreadCount(message.conversationId));
    }

    // Play notification sound
    playSound();
  });

  // B. Handling New Conversations
  // (Example: Being added to a group chat or receiving a
first-time direct message)
  socket.on('chat.new_conversation', ({ conversation }) => {
    dispatch(addConversation(conversation));
  });

  // C. Handling System Notifications
  socket.on('notification.new_notification', (notification) => {
    dispatch(addNotification(notification));
    playSound();
  });
```

- Flow:

1. Once the WebSocket is successfully connected, the client registers various socket.on() listeners to capture events pushed from the server.
2. Every time the server emits an event, the corresponding callback on the client is executed. These callbacks primarily function as bridges to the Redux Store, dispatching actions that update the global state to synchronize the UI in real-time.

4.5.3 Client Socket Configuration & Authentication

The socket client is configured to use WebSocket transport from the start, skipping unnecessary fallback mechanisms (like HTTP Long Polling) to minimize latency. The JWT token is attached directly to the handshake packet, allowing the Gateway to authenticate the connection immediately.

Controlling connections this way strengthens security and ensures that only authorized users can enter the real-time system, preventing unauthorized resource consumption.

Listing 8: Optimized Socket.io Client Configuration

```
// [File: socket.config.ts] - Socket Instance Initialization
import { io } from 'socket.io-client';

export const socket = io(REALTIME_GATEWAY_URL, {
  // 1. Transport Optimization:
  // Force WebSocket usage immediately, skipping the "HTTP
Long-polling" fallback mechanism
  // Helps reduce initial latency and save bandwidth.
  transports: ['websocket'],

  // 2. Connection Control:
  // Disable automatic connection (autoConnect: false).
  // Connection is only manually opened when the user is verified
as logged in.
  autoConnect: false,

  // 3. Authentication Mechanism (Authentication Injection)
  // The JWT token is included directly within the initial
Handshake packet.
  // The Gateway will immediately reject the connection attempt
if this token is found to be invalid or missing.
  auth: {
    token: localStorage.getItem('token'),
  },
});
```

- Flow:

1. When the application loads, a socket instance is created using io() with optimized configurations for production.
2. First, the system sets transports: ['websocket'] to enforce WebSocket usage from the start, bypassing the HTTP Long-polling fallback mechanism. This reduces connection latency and saves bandwidth.
3. Next, autoConnect is set to false. This ensures the socket does not automatically connect upon initialization; instead, the connection is controlled manually (e.g., after a user successfully logs in).

4. Finally, a JWT token is attached to the auth property. This token is sent within the initial handshake packet. If the token is invalid, the WebSocket Gateway will reject the connection immediately during the authentication phase.

5. Testing & Verification

5.1 Objective of the Testing Phase

The primary goal of this testing phase is to demonstrate the system's real-time, synchronous broadcast capabilities. This is a core functionality essential for ensuring a seamless, uninterrupted user experience.

5.2 Real-Time One-to-One Messaging Test (WebSocket)

Table 6: Testing results for real-time messaging (1–1) via WebSocket

Step	Action	Expected Result	Actual Result	Status
1	Open Browser Tab A (User: Alice)	WebSocket connection established successfully.	OK	Pass
2	Open Browser Tab B (User: Bob)	WebSocket connection established successfully.	OK	Pass
3	Alice sends a message: "Hello Bob!"	Bob's screen instantly updates with: "Alice: Hello Bob!"	Content updated instantaneously on both tabs (latency < 200ms).	Pass

Verification Method:

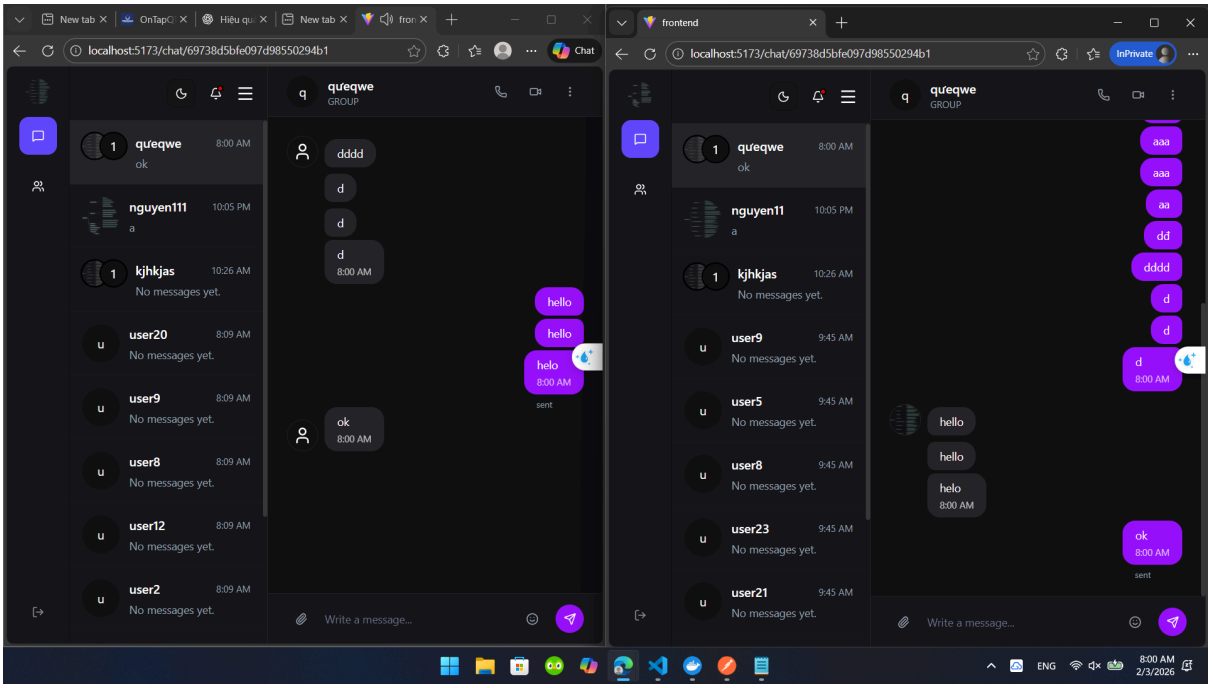

Figure 10: Testing results for real-time messaging (1–1) via WebSocket

5.3 Real-Time Group Messaging Test

Table 7: Test Results: Real-time Group Messaging Functionality

Step	Action	Expected Result	Actual Result	Status
1	User A creates a group chat and adds Users B and C.	Group chat is created. Users B and C receive a "Added to group" notification.	OK	Pass
2	Open 3 separate browser tabs for Users A, B, and C.	All 3 users successfully connect to the WebSocket server.	OK	Pass
3	User A sends a message to the group chat.	The screens for Users B and C update instantly with the new message.	Message received on both B and C's browsers simultaneously (latency < 200ms).	Pass

Verification Method:

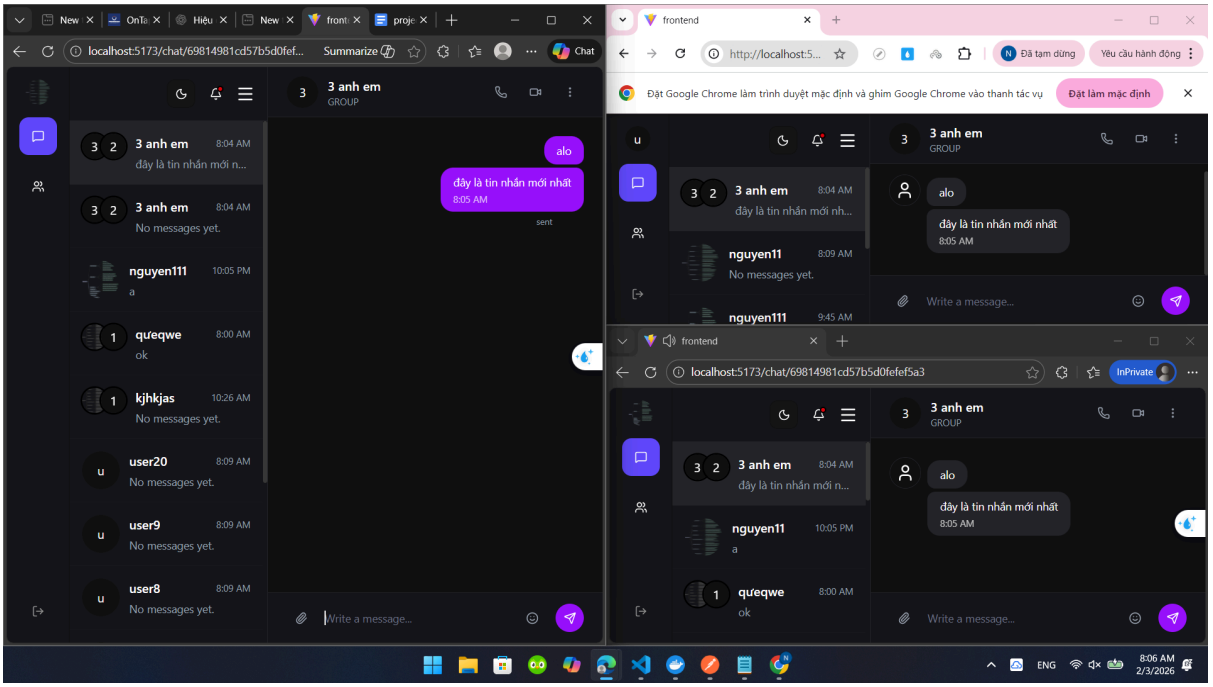

Figure 11: Test Results: Real-time Group Messaging Functionality

5.4 Connection Recovery and Chat History Synchronization

Table 8: Test Results: Connection Recovery and Chat History Synchronization

Step	Action	Expected Result	Actual Result	Status
1	User A and User B exchange messages.	Messages are displayed on both screens in real-time.	OK	Pass
2	User A closes the tab, then reopens it.	Connection is re-established. The application requests historical data from the API.	OK	Pass
3	User A clicks on the conversation with User B again.	Previous messages are fetched from the database and displayed correctly.	Conversation history is 100% accurately retrieved and displayed from MongoDB.	Pass

Verification Method:

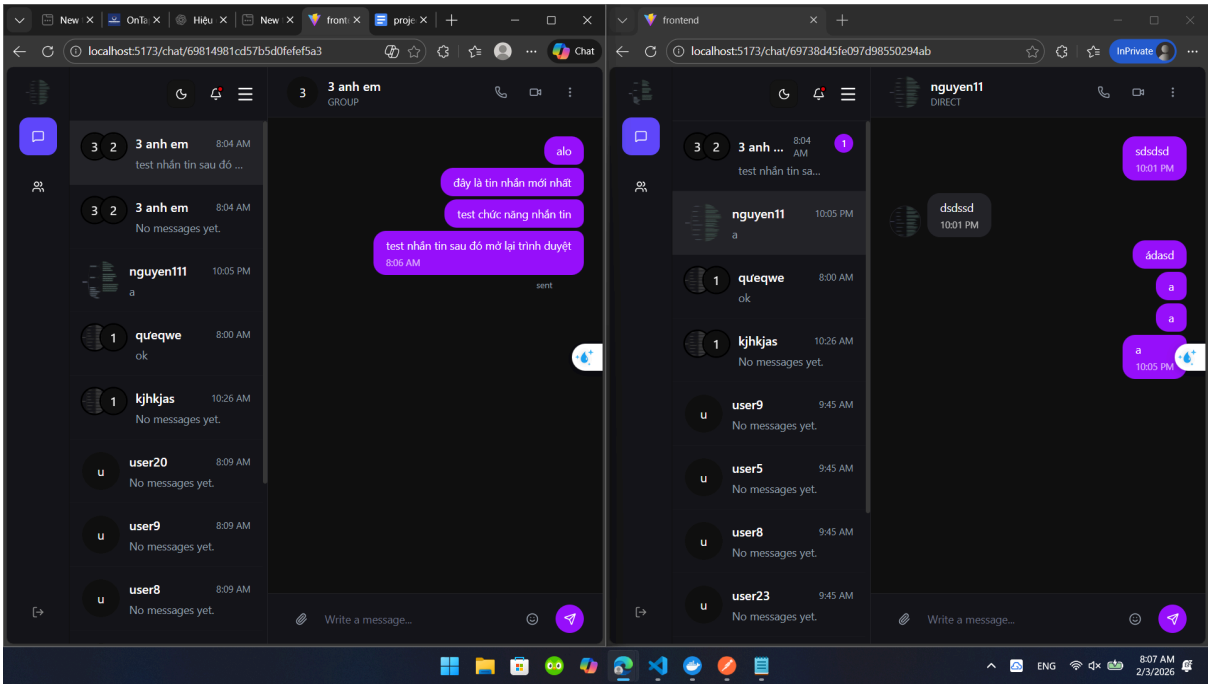

Figure 12: Test Results: Connection Recovery and Chat History Synchronization

5.5 Performance, Security, and Scalability Testing

Table 9: Performance, Security, and Scalability Test Results

Step	Action	Expected Result	Actual Result	Status
1	Access conversation list with 10,000 records	React loads only the first 20–50 records using Lazy Loading/Pagination. API Gateway retrieves data via gRPC with very low latency (<500ms).	Conversation list is displayed immediately; additional records are loaded smoothly on scroll without browser freezing.	Pass
2	Continuous message sending (Stress Test & Rate Limiting)	Send more than 10 messages per second. RateLimitGuard at the API Gateway detects excessive requests via Redis and throttles them to protect the server.	System displays a “Too Many Requests” notification when the rate limit is exceeded.	Pass

3	Receive messages on two devices (multiple web tabs)	A message is sent to User A. Realtime Gateway queries Redis to retrieve all active Socket IDs of User A and pushes the message to each connection.	Both active web tabs of User A receive the message simultaneously.	Pass
4	Search for a specific conversation among 10,000 records	The user enters a search keyword. API Gateway calls Chat Service to query indexed MongoDB collections.	Search results are returned in under 500ms despite the large dataset.	Pass
5	Message delivery via RabbitMQ under high load	Simulate 1,000 incoming messages simultaneously. Messages are routed through RabbitMQ to prevent API Gateway overload.	Messages are queued and delivered in order to the UI without loss or duplication.	Pass
6	Stress test concurrent user creation and group chat creation (k6 – 100 VUs × 20 iterations = 2,000 users)	The system concurrently handles user registration, authentication, friend requests, and group chat creation without errors. API Gateway and User/Chat Services remain stable under high load with no failed requests.	Successfully completed 2,000 iterations with a total of 8,002 HTTP requests. No failed requests (0% failure rate), no breached thresholds. Average response time ~1.9s. All group chats were created successfully with all members.	Pass

7	Real-time chat load test via WebSocket with 200 Virtual Users (k6), each user logs in, establishes a Socket.IO connection to the /realtime namespace, periodically sends messages via HTTP API, and receives real-time messages for 60 seconds	The system maintains stable WebSocket connections for all users, correctly processes real-time message sending and receiving, with no failed requests or breached thresholds. Messages send a success rate ≥ 95%.	Executed with 200 VUs and 200 iterations, totaling 2,011 HTTP requests. No failed requests and no breached thresholds. Message send success rate reached 100%, with 1,811 messages sent and 1,805 messages received via real-time channels.	Pass

Trends & Times

	AVG	MIN	MED	MAX	P(90)	P(95)
http_req_blocked	0.02	0.00	0.00	2.63	0.00	0.00
http_req_connecting	0.01	0.00	0.00	1.94	0.00	0.00
http_req_duration	1938.79	15.00	785.18	6861.26	5095.79	6312.48
http_req_receiving	0.07	0.00	0.00	2.50	0.52	0.54
http_req_sending	0.03	0.00	0.00	1.56	0.00	0.00
http_req_tls_handshaking	0.00	0.00	0.00	0.00	0.00	0.00
http_req_waiting	1938.69	14.45	785.03	6861.26	5095.79	6312.48
iteration_duration	7757.61	468.65	7792.96	9918.05	9221.42	9352.40

Figure 13: Test Results: Results of test number 6

Trends & Times

	AVG	MIN	MED	MAX	P(90)	P(95)
http_req_blocked	0.30	0.00	0.00	17.63	0.61	1.52
http_req_connecting	0.19	0.00	0.00	9.07	0.54	1.12
http_req_duration	6208.22	70.13	6559.83	14016.49	8523.67	9230.50
http_req_receiving	0.10	0.00	0.00	5.87	0.53	0.55
http_req_sending	0.09	0.00	0.00	15.34	0.51	0.58
http_req_tls_handshaking	0.00	0.00	0.00	0.00	0.00	0.00
http_req_waiting	6208.03	59.61	6559.64	14016.49	8523.15	9229.99
iteration_duration	69845.47	65918.94	70008.24	70147.46	70114.89	70122.24
ws_connecting	384.46	2.71	365.32	848.28	719.25	785.25
ws_session_duration	69490.36	65843.98	69590.12	70049.70	69923.50	69962.12

Figure 14: Test Results: Results of test number 7

6. Conclusion & Reflection

The project has successfully built a real-time Chat application based on an Event-Driven Microservices architecture. The core architectural objective of Real-Time Response has been fully achieved, with test results confirming that messages are propagated instantaneously across user sessions.

6.1 Lessons Learned

EDA Efficiency (Event-Driven Architecture): The combination of WebSockets and the Event-Driven model (via RabbitMQ) has demonstrated superior efficiency for real-time applications compared to traditional Request-Response (REST) or Polling models. This architecture effectively eliminates unnecessary latency by pushing data only when events occur.

Service Communication Optimization: Implementing gRPC for internal communication instead of REST JSON has significantly reduced packet sizes and accelerated response times between microservices (User, Chat, Notification). This ensures high data consistency across the distributed system.

Broadcast Simplicity: Decoupling the Real-time Gateway to manage socket connections has streamlined the business logic within the Chat Service. This separation of concerns allows the system to focus on data processing and persistence without the overhead of maintaining active connection states.

6.2 Future Improvements

Since the current system has finalized its core messaging features and foundational architecture, future enhancements will focus on Advanced Security, Operational Scaling, and Feature Expansion:

- Advanced Security:

- End-to-End Encryption (E2EE): Implementing E2EE using protocols like Signal or OTR (Off-the-Record). This ensures that private messages remain encrypted from sender to receiver, meaning not even system administrators or database intruders can read the content.

- Infrastructure & Scaling:
    - Kubernetes (K8s) Orchestration: Deploying the system to a K8s cluster to leverage Horizontal Pod Autoscaling (HPA). This allows the Chat Service and Real-time Gateway to automatically scale up during traffic spikes and scale down during idle periods.
    - Redis Caching Layer: Expanding the use of Redis to cache "hot" messages and user presence (Online/Offline) states. This significantly reduces the direct query load on MongoDB, optimizing overall system throughput.

- Feature Expansion:
    - Full-Text Search: Integrating Elasticsearch to enable high-speed, complex searching of message histories. This is essential for maintaining performance as the database grows into millions of records.
    - WebRTC Integration: Developing Video and Voice call capabilities using WebRTC to complete the communication ecosystem. This will provide peer-to-peer, low-latency media streaming directly within the application.