

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA E INFORMÁTICA
CONCEPCIÓN - CHILE



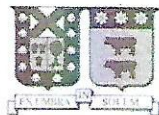
“Diseño, implementación y pruebas de un servicio de mensajería instantánea con WebSocket”

TOMÁS IGNACIO SOLAR CANALES

**MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO EN INFORMÁTICA**

Profesor Guía: CRISTIAN ANTONIO LARA VALENZUELA

Diciembre - 2025



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título; ☐ Tesis de Postgrado;

Título del trabajo: Diseño, Implementación y pruebas de un servicio de mensajería instantánea con WebSocket

Nombre del candidato(a): Tomás Ignacio Solar Canales

Carrera / Grado: Ingeniería en informática

Campus: Concepción; **Departamento:** Electrónica e Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Cristian Antonio Lara Valenzuela, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO** contiene información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis: Cristian Lara Valenzuela

Fecha: 19/12/2025

Firma:

Estudiante o Candidato(a): Tomás Solar Canales

Fecha: 19/12/2025

Firma:

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

DEDICATORIA

El presente trabajo es el fruto del esfuerzo y la constancia personal, y no puedo evitar emocionarme al alcanzar este gran hito en mi vida. Sin embargo, sería sumamente egoísta e injusto no dedicar este logro a quienes han sido mi mayor inspiración y apoyo incondicional: mi familia.

Desde mis primeros años de vida, ellos han sido el motor que me impulsó a perseguir mis sueños con pasión y a nunca rendirme. Han depositado en mí su confianza, su cariño y su amor inquebrantable durante todos estos años de estudio en la universidad, que algún día también fue el alma máter de toda mi familia.

A mis queridos padres, Juan Solar y Paz Canales, y a mi hermana Javiera Solar, dedico con todo mi corazón este trabajo. Este logro es tan suyo como mío, porque sin su apoyo, paciencia y amor, nada de esto habría sido posible.

Los amo.

Tomás Ignacio Solar Canales.

AGRADECIMIENTOS

De mi parte, sería sumamente egoísta no incluir ni mencionar a todas aquellas personas que hicieron que mi paso por la universidad fuera una experiencia inolvidable. Agradezco profundamente a mis amigos, profesores y funcionarios, quienes con su apoyo, compañía y consejos hicieron de esta etapa una de las más lindas de mi vida. A todos mis amigos de la universidad, que me demostraron que la vida universitaria no se trata solo de estudiar para certámenes o realizar proyectos, sino también de compartir momentos, hacer panoramas y disfrutar de actividades que forman parte del verdadero espíritu universitario. En especial, a Felipe Badilla, Fernando Cruces, Javier Choque, Ignacio Hormazábal, Dismar Quiroz, Alan Mac-Kay, Gabriel Farfán, Matías Torres y Sebastián Gallardo, por su amistad, compañía y por todos los momentos vividos juntos. También quiero expresar mi más sincero agradecimiento a los grandes profesores que me acompañaron durante mi formación profesional. En particular, al profesor José Arellano y al profesor Alonso Díaz Candia, por sus valiosos consejos, su guía constante y por las excelentes clases que marcaron mi desarrollo profesional y personal. Asimismo, al profesor Jonathan Briones, quien fue un gran apoyo durante mis primeros años en la universidad, brindándome siempre orientación y motivación para seguir adelante.

Finalmente, quiero agradecer a mis compañeros de trabajo en Buk, especialmente al equipo de Cálculos de Asistencia, con quienes he tenido la oportunidad de trabajar desde enero de 2025. Gracias por su paciencia, por compartir sus conocimientos y por motivarme día a día a ser un mejor ingeniero de software. Trabajar junto a ustedes ha sido una experiencia enriquecedora que me ha permitido crecer tanto personal como profesionalmente, aplicando todo lo aprendido en mi formación académica. A todos quienes formaron parte de este camino, muchas gracias por su apoyo, por su tiempo y por ser parte de una etapa que recordaré con enorme cariño.

RESUMEN

Resumen—La presente memoria aborda el diseño, implementación y validación de un módulo de mensajería instantánea para la plataforma *BeatMap*, orientada al ecosistema musical chileno. El objetivo principal fue desarrollar un *backend* capaz de permitir comunicación en tiempo real entre usuarios, mediante una arquitectura segura, escalable y preparada para migrarse a una arquitectura de microservicios. Para esto, se implementó un *backend* basado en *Spring Boot*, WebSocket con *STOMP* y *RabbitMQ* como *broker* de mensajería, junto con una base de datos *PostgreSQL* que soporta chats privados, historiales de mensajes paginados y estados de lectura. La metodología incluyó el análisis de requisitos tecnológicos para el diseño arquitectónico, diseño iterativo y validación exhaustiva mediante pruebas unitarias, de integración y pruebas funcionales por procesos *UAT* a través de vistas *HTML* especializadas. Los resultados demostraron que el módulo es estable, consistente y capaz de procesar, transmitir y persistir mensajes sin pérdidas de información, permitiendo un flujo de comunicación confiable en tiempo real. La solución constituye en aporte técnico significativo para *BeatMap*, habilitando la futura expansión del módulo hacia funcionalidades avanzadas como envío de archivos multimedia, optimización de servicios y cifrado end-to-end.

Palabras Clave—Websocket; Spring Boot; RabbitMQ; Mensajería instantánea; Backend.

ABSTRACT

Abstract—*This thesis addresses the design, implementation and validation of an instant messaging module for the BeatMap platform, aimed at the Chilean music ecosystem. The main objective was to develop a backend capable of enabling real-time communication between users, using a secure, scalable architecture that is ready to migrate to a microservices architecture. To this end, a backend based on Spring Boot, WebSocket's with STOMP and RabbitMQ as a message broker was implemented, along with a PostgreSQL database that supports private chats, paginated message histories and read statuses. The methodology included the analysis of technological requirements for architectural design, iterative design, exhaustive validation through unit testing, integration testing, and functional testing by UAT processes using specialized HTML views. The results demonstrated that the module is stable, consistent, and capable of processing, transmitting, and persisting messages without loss of information, enabling reliable real-time communication flow. The solution constitutes a significant technical contribution to BeatMap, enabling future expansion of the module toward advanced functionalities such as multimedia file sending, service optimization, and end-to-end encryption.*

Keywords— WebSocket; Spring Boot; RabbitMQ; Instant messaging; Backend.

GLOSARIO

AmqpException: Excepción lanzada en aplicaciones basadas en *SpringBoot* al producirse errores de comunicación con *brokers AMQP* como *RabbitMQ*.

API (Application Programming Interface): Conjunto de reglas y especificaciones que las aplicaciones pueden utilizar para comunicarse entre ellas.

API Gateway: Componente que actúa como punto de entrada único para todas las peticiones a un conjunto de microservicios, gestionando la autenticación, autorización y validación.

ArgumentCaptor: Clase de *Mockito* usada en Java para capturar argumentos pasados a método en pruebas unitarias, permitiendo verificaciones detalladas en invocaciones.

AWS (Amazon Web Services): Plataforma de servicios en la nube que ofrece infraestructura escalable y soluciones de almacenamiento, bases de datos, seguridad, monitorio y despliegue.

AWS S3 (Simple Storage Service): Servicio de almacenamiento de objetos de AWS. Permite guardar y recuperar archivos, imágenes, audios y videos de forma escalable y segura.

Back-End: Parte del desarrollo de software que se ejecuta desde el lado del servidor, la cual maneja la *lógica de negocio* y la gestión de los datos.

Bearer Authentication: Método de autenticación *HTTP* donde el cliente debe enviar un token de acceso en formato *JWT* en la cabecera de la petición usando el esquema *Bearer*.

Binding: Propiedad de *RabbitMQ* que establece la relación entre un *exchange* y una cola en *RabbitMQ*, que define como los mensajes se enrutan en función de la *routing key*.

Booleano: Tipo de datos que representa dos valores posibles: verdadero (*true*) o falso (*false*). Usando la lógica condicional y programación.

Buk: Empresa de software chilena que desarrolla un software integral de recursos humanos.

Broker de mensajería: Componente de software intermediario que recibe, enruta, almacena y entrega mensajes entre aplicaciones o servicios.

Configuration: Capa propia del proyecto de BeatMap que contiene la capa de seguridad de la aplicación por medio de la librería Spring Security.

ConvertAndSend: Método de la API de *RabbitTemplate* de *SpringBoot* para enviar un mensaje serializado a un *exchange* en *RabbitMQ*.

CRUD: Operaciones básicas de creación, edición, eliminación y lectura de datos en aplicaciones y bases de datos.

CSRF (Cross-site Request Forgery): Ataque que fuerza a un usuario autenticado a ejecutar acciones no deseadas en una aplicación web en la que ha iniciado sesión.

DELETE: Verbo *HTTP* utilizado en *APIs RESTful* para eliminar recursos del servidor.

DevOps: Práctica y cultura que integra equipos de desarrollo (*Dev*) y operaciones (*Ops*) automatizar la entrega continua de software, mejorar la colaboración y acelerar los ciclos de despliegue.

Docker: Plataforma que permite empaquetar, distribuir y ejecutar aplicaciones en contenedores ligeros y portables, estandarizando entornos de ejecución.

DTO (Data Transfer Object): Objeto simple diseñado para el transporte de datos entre procesos de negocio.

Endpoint: Punto de entrada de una API. Se pueden realizar peticiones a esta y se recibe una respuesta con datos.

End-to-end: Método de comunicación que garantiza que solo el emisor y el destinatario del mensaje puedan leer su contenido.

Entity: Capa del framework de *SpringBoot*, que contiene las entidades que serán mapeadas a la base de datos como tablas mediante el *ORM* del *framework*.

EntityNotFoundException: Excepción utilizada en aplicaciones Java para indicar que en una entidad buscada en la base de datos no existe.

Exchange: Componente en *RabbitMQ* que recibe mensajes de productores y los enruta a las colas correspondientes según reglas definidas.

IETF (Internet Engineering Task Force): El grupo de trabajo de ingenieros de internet es una organización fundada en los años sesenta con la finalidad de hacer del internet un lugar mejor.

IllegalArgumentException: Excepción estándar de Java lanzada cuando un método recibe un argumento inválido.

Interceptor: Componente que intercepta y procesa peticiones o mensajes antes o después de que lleguen a su destino, permitiendo aplicar lógica adicional de validación y seguridad.

Fail-fast: Principio de diseño de *software* que implica que una aplicación debe fallar inmediatamente al detectar una condición inesperada o un error, en lugar de intentar continuar.

Front-End: Parte del desarrollo de software que se encarga de la interfaz de usuario.

Framework: Conjunto de herramientas y librerías que proporcionan una estructura estándar y ágil para el desarrollo de software.

Full-Dúplex: Transmisión y recepción de datos a través de una canal de manera simultánea.

Git: Sistema de control de versiones de software.

GUI (Interfaz gráfica de usuario): Interfaz que permite la interacción entre usuarios y sistemas mediante elementos gráficos como ventanas, botones, menús y formularios, facilitando la usabilidad.

H2 Database: Sistema de base de datos relacional y ligera en memoria, comúnmente usado para tests y entornos de desarrollo.

HMAC-SHA256: algoritmo criptográfico de firma que combina una llave secreta con el hash SHA-256 para verificar la integridad.

HTML (HyperText Markup Language): Lenguaje estándar para estructurar páginas web y aplicaciones en internet mediante etiquetas y contenido multimedia.

JavaScript: Lenguaje de programación interpretado principalmente para el desarrollo d aplicaciones web interactivas en el navegador.

Jenkins: Herramienta de automatización de código abierto empleada para crear, ejecutar y gestionar *pipelines* de integración y entrega continuas (CI/CD), soportando tareas como compilación, pruebas y despliegue.

JSON (JavaScript Object Notation): Formato ligero y legible para el intercambio de datos estructurado entre aplicaciones.

JVM (Java Virtual Machine): Máquina virtual que puede ejecutar código Java en múltiples dispositivos.

Junit: Framework para pruebas de código de Java. Permite escribir y ejecutar tests unitarios automatizados de manera sencilla.

JWT (JSON Web Tokens): Estándar abierto que define de manera sencilla la transmisión de información de manera segura.

Log: Registros automático de sucesos, errores y eventos generados por una aplicación, útil para auditoría, monitoreo y detección de fallos.

MessageMapping: Anotación de *Spring* utilizada para mapear destinos de mensajes *STOMP* a métodos del *backend*.

Microservicios: Arquitectura de software que separa los servicios en aplicaciones independientes.

Monolito: Arquitectura que todas las funcionalidades se implementan en una sola aplicación

MongoDB: Base de datos documental orientada a documentos en formato *BSON* que permite almacenar, indexar y consultar grandes volúmenes de datos no estructurados o semiestructurados.

MVC: Patrón de diseño el cual separa la aplicación en tres componentes principales.

Mock: Objeto simulado usado en pruebas de software para replicar el comportamiento de componente reales y validar la interacción entre ellos.

MockMVC: Herramienta de spring que permite simular peticiones *HTTP* en entornos de prueba sin la necesidad de exponer los *endpoints* en un entorno de producción.

Mockito: Framework de pruebas de código para Java que permite crear datos de prueba que se almacenan en memoria.

ORM (Object-Relational-Mapping): Técnica que permite mapear las clases como tablas en la base de datos.

PageRequest: Clase de *Spring Data* que representa una solicitud de página para paginar resultados de consultas en base de datos.

Paginated (*Paginación*): Técnica que divide resultados en grupos o páginas para optimizar la visualización y manejo de grandes volúmenes de datos comúnmente implementado en *APIs*.

Patrón N+1: Problema de rendimiento de base de datos con *ORM*, donde cada entidad principal consulta *n* veces datos relacionados, generando muchas consultas a la base de datos. Esto debe eliminarse mediante consultas optimizadas.

Pipeline: Secuencia automatizada de pasos que define el flujo de compilación, prueba, integración y despliegue de software, facilitando el *CI/CD* en proyectos de software.

PostgreSQL: sistema gestor de bases de datos.

PostGIS: Extensión de la base de datos *PostgreSQL* que habilita el almacenamiento, consulta y análisis de datos espaciales y geográficos.

Principal: Objeto de seguridad en Spring que representa la identidad del usuario autenticado en una sesión o contexto de seguridad.

Principios ACID: Conjunto de propiedades que garantizan que las transacciones en bases de datos sean confiables y mantengan la integridad de la información.

Pull request: Mecanismo del control de versiones que permite proponer cambios al código fuente de la aplicación, los cuales pueden ser discutidos y aprobados para unificarlos en la *Branch principal*.

RabbitAdmin: Clase de *SpringBoot* que permite la administración y configuración automática de colas, *exchanges* y *bindings* en *RabbitMQ*.

RabbitMQ: Sistema de mensajería que gestiona colas y enrutamiento de mensajes entre servicios de forma asíncrona.

RabbitTemplate: Componente de *SpringBoot* que facilita la publicación y el consumo de mensajes en *RabbitMQ*, simplificando la interacción con este *broker*.

Rate Limiting: Técnica de control que limita el número de peticiones o eventos que un usuario puede ejecutar en un periodo determinado, previniendo sobrecargas al servidor.

React: Librería de *JavaScript* para desarrollar interfaces de usuario de forma eficiente y mantenible por medio de componente reutilizables.

Redis: Base de datos en memoria de tipo clave-valor usado principalmente como sistema de caché y sesiones en aplicaciones de alta concurrencia, optimizando la rapidez de acceso a la información de un sistema.

Repository: Capa del *framework* de *SpringBoot* que se encarga de manejar las consultas a la base de datos.

RFC (Request for comments): Documento que define y describe protocolos, métodos y conceptos de internet. Gestionado por el *IETF*.

SecurityFilterChain: Cadena de filtros que gestionan la autenticación, autorización y manejo de seguridad en aplicaciones web basadas en *SpringBoot*. Configurado mediante la librería *Spring Security* para validar usuarios y proteger *endpoints*.

Sentry: Herramienta de monitoreo de errores y trazabilidad en tiempo real para aplicaciones que permite registrar fallos, excepciones y recibir alertas automáticas.

Service: Capa del *framework* de *SpringBoot* que contiene la lógica de negocio de la aplicación.

Spring-boot-starter-websocket: Librería de *SpringBoot* que permite la implementación del protocolo *WebSocket* y *STOMP* para aplicaciones del ambiente *Spring*.

Spring Security: Framework de autenticación y autorización altamente personalizable para aplicaciones basadas en la JVM.

SpringBoot Tests: Framework de pruebas del *framework* de *SpringBoot* que ofrece utilidades, anotaciones y contexto fácil de configurar para escribir tests unitarios y de integración.

Stateless: Arquitectura en la que cada petición contiene toda la información necesaria para ser procesada, sin la necesidad de depender de sesiones almacenadas en el servidor.

STOMP (Simple Text Orientated Messaging Protocol): Protocolo de mensajería asíncrona.

Stomp.js: Librería de implementación de *JavaScript* del protocolo *STOMP*, utilizada para enviar y recibir mensajes sobre *WebSocket* en aplicaciones web en tiempo real.

Stubbing: Técnica de pruebas de software que consiste en reemplazar la implementación real de un método de un servicio por una simulada o predefinida.

TCP (Transmission Control Protocol): Protocolo de red que permite el intercambio de datos de manera confiable.

Token: Cadena de caracteres que representa la identidad o permisos de un usuario y se utiliza para autenticar o autorizar solicitudes dentro de un sistema.

UAT (User Acceptance Testing): pruebas de aceptación de usuarios para verificar si un producto o servicio cumple con los requisitos específicos de los usuarios finales.

WebSocket: Protocolo de comunicación bidireccional entre el cliente y servidor a través de una sola conexión TCP.

INDICE DE CONTENIDOS

ABSTRACT	4
INDICE DE FIGURAS	12
INDICE DE TABLAS	15
INTRODUCCIÓN	16
I. CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA.....	17
II. CAPÍTULO 2: MARCO CONCEPTUAL	20
III. CAPÍTULO 3: PROPUESTA DE SOLUCION	53
IV. CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	67
V. CAPÍTULO 5: CONCLUSIONES	78
REFERENCIAS BIBLIOGRÁFICAS	91

INDICE DE FIGURAS

Figura 1: Funcionamiento del protocolo WebSocket vs peticiones HTTP. Fuente: maybe.works. ...	25
Figura 2: Funcionamiento de una API Fuente: Outvio.....	32
Figura 3: Funcionamiento de un webhook vs una API tradicional Fuente: Zapier.	36
Figura 4: Funcionamiento del ORM en una aplicación hecha en Java Fuente: Ramesh Fadatare.	39
Figura 5: Funcionamiento de redis en una aplicación Fuente: Recuperado de Backendless.	41
Figura 6: Diferencia entre la autenticación y autorización Fuente: Recuperado de okta.	43
Figura 7: Estructura de JWT	45
Fuente: recuperado de Dev tools, hecho por Daniel DeGroff.....	45
Figura 8: Funcionamiento de la cadena de filtros de SpringSecurity Fuente: Elaborada por Kasun Dissanayake	47
Figura 9: Modelo original de Mike Cohn de La Pirámide de Pruebas Ágiles. Fuente: Elaborada por Mike Cohn.....	50
Figura 10: Arquitectura general del proyecto.....	53
Fuente: Elaborado por Dismar Quiroz	53
Figura 11: Modelo de datos del módulo de chat de BeatMap	55
Fuente: Elaboración propia.....	55
Figura 12: Diagrama de secuencia del módulo de chat de BeatMap Fuente: Elaboración propia... ..	58
Figura 13: Diagrama de secuencia del funcionamiento de la seguridad del módulo de mensajería	62
Fuente: Elaboración propia.....	62
Figura 14: Reporte de las pruebas realizadas para el MessageServiceImpl.	70
Fuente: Elaboración propia.....	70
Figura 15: Reporte de las pruebas realizadas para el ChatServiceImpl.	74
Fuente: Elaboración propia.....	74
Figura 16: Reporte de las pruebas realizadas para el servicio que gestiona la integración con RabbitMQ.....	76
Fuente: Elaboración propia.....	76
Figura 17: Panel de administración para la validación del funcionamiento del envío de mensajes.	78
Fuente: Elaboración propia.....	78

Figura 18: Logs de la interacción entre el backend y el servicio de RabbitMQ para el envío de mensajes.....	78
Fuente: Elaboración propia.....	78
Figura 19: Registros de la tabla Messages de la base de datos con el mensaje de prueba enviado.	79
Fuente: Elaboración propia.....	79
Figura 20: Vista de administración para validar las funcionalidades a la obtención del historial de mensajes.....	80
Fuente: Elaboración propia.....	80
Figura 21: Vista de administración para validar la obtención del historial de mensajes paginados.	81
Fuente: Elaboración propia.....	81
Figura 21: Registros de la tabla Message_history con el total de mensajes enviados.	81
Fuente: Elaboración propia.....	81
Figura 22: Vista de administración para validar el estado de lectura de los mensajes enviados.	83
Fuente: Elaboración propia.....	83
Figura 23: Formulario para marcar un mensaje como leído.	83
Fuente: Elaboración propia.....	83
Figura 24: Registros en la base de datos para la tabla <i>Message_read_status</i> para validar el estado de lectura de los mensajes enviados por los usuarios.	84
Fuente: Elaboración propia.....	84
Figura 25: Vista de validación de los <i>endpoints</i> del servicio de mensajería.....	85
Fuente: Elaboración propia.....	85
Figura 25: Registros de la base de datos de la tabla <i>Messages</i> para validar la persistencia de los mensajes.....	85
Fuente: Elaboración propia.....	85
Figura 26: Vista de validación de la edición de los mensajes enviados.	86
Fuente: elaboración propia	86
Figura 27: Registros de la base de datos de la tabla Messages para validar la persistencia de los mensajes editados por el usuario.	86
Fuente: Elaboración propia.....	86
Figura 28: Registros iniciales en la tabla Messages.....	87

Fuente: Elaboración propia.....	87
Figura 29: Confirmación visual del borrado del mensaje mediante la vista.....	87
Fuente: Elaboración propia.....	87
Figura 30: Registros de la tabla <i>Messages</i> después de la eliminación del mensaje con <i>id</i> 3.	88
Fuente: elaboración propia.	88

INDICE DE TABLAS

Tabla 1: Tabla de objetivos específicos. Fuente: Elaboración Propia.	19
Tabla 2: Tabla de ventajas y desventajas de Ruby on Rails. Fuente: Elaboración Propia.	21
Tabla 3: Tabla de ventajas y desventajas de SpringBoot. Fuente: Elaboración Propia.	22
Tabla 4: Ventajas y desventajas del framework de Grails.....	23
Fuente: Elaboración propia.....	23
Tabla 5. Tabla comparativa para la elección del framework más adecuado al proyecto según los criterios técnicos previamente establecidos.	24
Fuente: Elaboración propia.....	24
Tabla 6. Tabla comparativa para la elección del broker de mensajería adecuado.	28
Fuente: Amazon (AWS, 2023).	28
Tabla 7: Tabla comparativa de sistemas gestores de base de datos. Fuente: recuperada de datdata.	30
Tabla 8: Ventajas y desventajas de utilizar una API RESTful. Fuente: recuperada de medium.	33
Tabla 9: Ventajas y desventajas de utilizar una API WebSocket. Fuente: Elaboración propia	34
Tabla 10: Ventajas y desventajas de utilizar una API basada en Webhooks. Fuente: elaboración propia	35
Tabla 11: Comparativa entre una API RESTful, WebSocket y Webhooks Fuente: elaboración propia.....	37
Tabla 12: Ventajas y desventajas del uso de un ORM en SpringBoot Fuente: elaboración propia.	40
Tabla 13: Diferencias entre autenticación y autorización Fuente: Recuperada de okta.....	44
Tabla 14: Ventajas y desventajas de JWT. Fuente: Elaboración propia	46
Tabla 15: función y uso en BeatMap de Spring Security y JWT Fuente: Elaboración propia	48
Tabla 16: Ventajas y desventajas de las pruebas de software. Fuente: Elaboración propia	51
Tabla 17: Ventajas y desventajas de las pruebas las distintas librerías de testing. Fuente: Elaboración propia	52
Tabla 18: Listado de endpoints del módulo de mensajería. Fuente: Elaboración propia	59

INTRODUCCIÓN

La presente memoria aborda el diseño, implementación y validación de un módulo de mensajería instantánea para plataforma digital *BeatMap*, orientada a la industria musical chilena. En este contexto, el desarrollo de servicios de comunicación robustos y seguros se ha vuelto una necesidad fundamental para facilitar la colaboración entre músicos y organizadores de eventos, permitiéndoles gestionar proyectos, coordinar instancias y compartir información en tiempo real de manera eficiente.

El área de trabajo se enmarca en el desarrollo backend de aplicaciones web modernas, donde la comunicación en tiempo real, la escalabilidad y la seguridad de los datos son exigencias crecientes frente a los desafíos de plataformas sociales colaborativas. El problema particular abordado radica en la ausencia de soluciones integrales que permitan centralizar la gestión de eventos, la comunicación directa y la coordinación profesional en el ecosistema musical, superando las limitaciones de aplicaciones tradicionales como WhatsApp e Instagram, que carecen de trazabilidad, integración y herramientas especializadas para este dominio.

Para responder a este desafío, se plantea como solución la implementación de un módulo de mensajería bajo una arquitectura escalable, desacoplada y segura, utilizando tecnologías y patrones actuales como *Spring Boot*, *WebSocket*, *STOMP* y *RabbitMQ*, apoyados por bases de datos relacionales y de caché (*PostgreSQL* y *Redis*). La propuesta integra mecanismos de autenticación robustos mediante *JWT* y *Spring Security*, junto con la adopción de estándares y buenas prácticas en el diseño de *APIs*, manejo de persistencia, paginación y validación de acceso, anticipando una futura migración hacia microservicios y servicios en la nube (*AWS S3*) para almacenamiento multimedia.

La metodología empleada para desarrollar la memoria incluyó un riguroso proceso de análisis y levantamiento de requisitos, selección y justificación de la arquitectura y tecnologías empleadas, la implementación iterativa del *backend*, y una validación exhaustiva basada en pruebas unitarias, de integración y funcionales por medio de procesos de *UAT*. Para garantizar la calidad y robustez de la solución, se aplicó la pirámide de pruebas de software de Cohn, asegurando la cobertura de lógica de negocio, persistencia de datos, seguridad y rendimiento. Esta memoria ofrece una visión integral del proceso técnico, las decisiones tomadas y los aprendizajes obtenidos en la creación de un sistema de mensajería instantánea preparado para los desafíos actuales y la evolución futura de *BeatMap* como plataforma social colaborativa.

I. CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA

1.1 Contextualización del proyecto.

La industria musical emergente enfrenta múltiples barreras para poder acceder de manera equitativa a oportunidades que brinden visibilidad y permitan profesionalizar su trabajo. En Chile, existen al menos 35 mil músicos activos, según cifras de la Sociedad Chilena de Autores e Intérpretes Musicales (SDC). De los cuales, la gran mayoría de estos músicos se encuentran en una constante búsqueda de espacios y oportunidades para presentarse en eventos que les brinden difusión de sus proyectos musicales. Y, sobre todo contrataciones que les permitan profesionalizar su talento para poder vivir de su pasión, la música. Si bien, actualmente existen orientadas a los músicos como Vampr, una red social estadounidense lanzada en el año 2016. Con un concepto similar a la popular aplicación de citas Tinder, pero con un enfoque diseñado para músicos. Donde Vampr permite que sus usuarios creen perfiles en los que mencionan los instrumentos que tocan, con una breve biografía y fotografías del usuario. De esta forma, se genera un feed con las posibles conexiones que se pueden generar, basadas en la ubicación del usuario. No obstante, esta aplicación no contempla ni incluye a todos los actores clave de la industria musical, tales como productores, sellos discográficos o incluso a los mismos oyentes. Además, tampoco cuenta con opciones de contratación directa de servicios o visualizaciones de eventos locales. Dado estas limitantes y al bajo impacto que ha tenido la aplicación en la última década, los músicos recurren mayormente a redes sociales como Tiktok, Instagram, Facebook o WhatsApp para difundir su trabajo y buscar oportunidades. Sin embargo, estas aplicaciones ofrecen un alcance muy limitado, baja trazabilidad y la ausencia de filtros y de herramientas profesionales diseñadas para impulsar el crecimiento de sus carreras musicales.

Frente a esta necesidad de los músicos chilenos y latinoamericanos, surge BeatMap. Una plataforma digital integral que busca revolucionar la industria musical local. BeatMap conecta a todos aquellos. Quienes crean, producen, impulsan y celebran la música. Con un mapa interactivo, una red profesional y con funcionalidades clave diseñadas para potenciar el ecosistema musical. Donde la comunicación es uno de los puntos clave para los distintos actores de la industria musical es un elemento clave que, para el networking en el mundo de la música, por lo que centralizar las interacciones en una sola plataforma es clave. Para evitar que los usuarios no dependan de herramientas externas para gestionar su carrera musical. Dado esto, el presente trabajo se centra en el proceso de diseño, implementación y pruebas de un servicio de mensajería en BeatMap, utilizando el protocolo *WebSocket* para la comunicación. Este módulo de chat en la plataforma beneficiará a los usuarios en el proceso de coordinación de eventos, la organización de nuevos proyectos musicales y la comunicación interna para contratar y gestionar servicios musicales. Actualmente, no existe una plataforma que integre de manera centralizada la gestión de eventos, la postulación de artistas musicales a eventos y la comunicación directa entre los distintos actores de la industria musical. Las soluciones tecnológicas existentes se enfocan en aspectos aislados del ecosistema musical. Como la difusión de contenido o la búsqueda activa de colaboradores, pero estas plataformas no ofrecen un entorno que permita coordinar todas

las etapas del proceso musical. En este caso el módulo de mensajería de *BeatMap* surge como un componente clave que aportará valor estratégico a la plataforma, al ofrecer un canal de comunicación interno, seguro y en tiempo real. Que facilitará la coordinación entre músicos, productores y organizadores de eventos. Este módulo busca centralizar las interacciones y fortalecer el networking profesional, eliminando la dependencia de herramientas externas como WhatsApp o Instagram. Consolidando a *BeatMap* como una solución integral para el desarrollo de la industria musical chilena.

1.2 Objetivo General del proyecto.

Diseñar e implementar un módulo de mensajería instantánea mediante el protocolo de comunicación *WebSocket*, utilizando el *framework Spring Boot*. Este módulo tiene como propósito facilitar el networking entre los usuarios, aportando una herramienta de valor a la comunicación dentro de la plataforma. Garantizando su escalabilidad, mantenibilidad a largo plazo y alta disponibilidad del servicio mediante la adopción de un estándar de calidad de software basado en la pirámide de Cohn para realizar pruebas unitarias, de integración de sistema, así como procesos *UAT* de aceptación de usuarios finales.

1.3 Tabla de objetivos específicos.

Objetivo	Marco conceptual	Propuesta de solución
Diseñar una arquitectura escalable para el módulo de chat de la aplicación definiendo las herramientas, tecnologías y protocolos necesarios para esta implementación.	Conceptos de arquitecturas de software (<i>Monolito</i> , <i>Backend-Frontend</i> , <i>Microservicios</i>) para el diseño de una arquitectura escalable a largo plazo. Junto con un modelado relacional de la base de datos normalizada para la persistencia de los mensajes enviados mediante el protocolo <i>WebSocket</i> .	Diseñar una arquitectura <i>Backend-Frontend</i> , con una estructura de carpetas mediante una modularización por casos de uso. Junto con un modelo de datos normalizado para el módulo de chats usando el framework de <i>Java</i> , <i>SpringBoot</i> y el protocolo <i>WebSocket</i> para el envío de mensajes para los usuarios.
Implementar la <i>API WebSocket</i> junto con la persistencia de mensajes a la base de datos.	Fundamentos del protocolo <i>WebSocket</i> (RFC-6455) y <i>STOMP</i> para el enrutamiento de mensajes y el <i>ORM</i> de <i>Spring Boot</i> como capa de persistencia principal y <i>Redis</i> como	Desarrollar los endpoints para la mensajería con protocolo <i>WebSocket</i> usando <i>Spring Boot</i> y <i>PostgreSQL</i> para la persistencia de los mensajes. Junto con la base

	base de datos en caché para optimizar el rendimiento de las consultas a la base de datos.	de datos Redis para la persistencia en caché para optimizar la performance de la base de datos.
Integrar protocolos de autenticación y autorización para garantizar conexiones seguras entre los miembros de los chats.	Principios de autenticación basado en tokens mediante <i>JWT</i> y Spring Security para autenticar y autorizar peticiones realizadas desde el Frontend.	Configurar los filtros de seguridad usando Spring Security para la creación y validación de tokens mediante <i>JWT</i> para permitir el uso de los endpoints disponibles desde el <i>backend</i> .
Adoptar un estándar de calidad de software mediante pruebas unitarias, de integración, sistema y de aceptación de usuario. Para validar el funcionamiento de los servicios implementados.	Enfoque en establecer un estándar de calidad de software basado en la pirámide de Cohn para realizar pruebas con diferentes casuísticas que validen todas las funcionalidades del módulo de chat.	Implementar tests unitarios usando la librería <i>Junit</i> y <i>Mockito</i> para probar los servicios de los endpoints del módulo de chat. Además de configurar tests de integración mediante Spring Boot tests.

Tabla 1: Tabla de objetivos específicos.

Fuente: Elaboración Propia.

II. CAPÍTULO 2: MARCO CONCEPTUAL

Es menester contar con una fundamentación técnica sólida que nos permita tomar las decisiones adecuadas, evitando problemas futuros como la deuda técnica, uno de los mayores desafíos que se enfrentan los desarrolladores de software en su día a día. La cual se refiere a todas aquellas soluciones “parche” que en un principio resuelven un problema en específico pero que, en un plazo de dos a tres años generarán complicaciones adicionales que aumentarán la complejidad y mantenibilidad del sistema, limitando la rapidez de implementar nuevas funcionalidades, dado decisiones técnicas sin buenos fundamentos de fondo ni evidencia empírica que valide la solución propuesta. Teniendo en cuenta estos puntos, este apartado se presentarán las herramientas y tecnologías analizadas rigurosamente que buscan sentar las bases tecnológicas que compondrán el módulo de chats de BeatMap.

2 Decisión técnica y diseño de la arquitectura del módulo de chat

La selección de las herramientas y tecnologías que se utilizaron en el desarrollo del módulo de chat y del backend completo de BeatMap debieron considerar los siguientes criterios técnicos:

- El framework debe tener soporte nativo para la arquitectura de *microservicios*. Facilitando la escalabilidad horizontal del proyecto con una migración transitoria a esta arquitectura.
- Disponibilidad de librerías robustas para el protocolo *WebSocket* y *STOMP*, evitando la dependencia de librerías externas con baja escalabilidad y mantenibilidad.
- Un motor de base de datos altamente escalable, robusto y eficiente para el almacenamiento de grandes volúmenes de datos.

2.1 Elección del framework para el proyecto.

En ingeniería de software, un *framework* consta de una serie de componentes y herramientas de softwares reutilizables que entregan una estructura base a un proyecto de software. Lo cual, agiliza en gran medida el proceso de desarrollo de nuevas aplicaciones de su inicio, hasta su despliegue en un ambiente de producción en un servidor. Uno de los mayores beneficios del uso de *frameworks* dentro de los equipos de desarrollo es que, gracias a que los frameworks incluyen una gran cantidad de *API's*, librerías de código de todo tipo, depuradores de código y compiladores. Los desarrolladores únicamente se deben de centrar en el desarrollo de la lógica de negocio del proyecto sin preocupaciones innecesarias de tener que configurar herramientas propias para desarrollar una aplicación.

2.2 ¿Qué es Ruby on Rails?

Ruby on Rails es un *framework open source* de desarrollo web para el lenguaje de ruby que implementa el patrón de diseño modelo vista controlador (MVC) (Ruby on rails, s.f.). Este *framework* sigue la filosofía de “convención sobre configuración”. La cual establece valores por defecto para clases, nombres de archivos y rutas, con la finalidad de acelerar el proceso de desarrollo sin tener que implementar configuraciones explícitas que restan tiempo al proceso de desarrollo de lógica de negocios (Microsoft, 2015)

Ventajas y desventajas de utilizar Ruby on Rails

Ventajas	Desventajas
Ruby tiene un potencial de escalabilidad gigante, con aplicaciones como <i>Buk</i> y <i>GitHub</i> que están creadas con Ruby on Rails.	Con aplicaciones muy grandes y con mucho tráfico de usuarios, las aplicaciones pueden tener problemas de rendimiento.
Tiene una gran comunidad que activamente mejora día a día el framework y sus librerías (gemas).	Algunos plugins como <i>Sorbet</i> para la validación de tipado estático dentro del código fuente genera altos problemas de rendimiento para los desarrolladores
Incluye integración nativa de librerías de testing para realizar procesos de <i>TDD</i> más cómodos.	Al existir una gran cantidad de tests dentro del proyecto, los procesos de <i>CI/CD</i> se vuelven más lentos.

Tabla 2: Tabla de ventajas y desventajas de Ruby on Rails.

Fuente: Elaboración Propia.

2.3 ¿Qué es SpringBoot?

SpringBoot, es un *framework open source* para Java lanzado en el 2014, este *framework* está diseñado para acelerar el proceso de desarrollo de aplicaciones para ser desplegadas en un ambiente de producción (Reddit, 2023). A diferencia de su predecesor *spring*. *SpringBoot* al igual que Ruby on Rails, sigue la filosofía de convención sobre configuración, gracias a su amplia gama de módulos que no requieren de configuraciones manuales (Geeks for geeks , 2025). Sino que, al igual que el conocido *framework* de Ruby, Ruby on Rails. Este sigue la misma filosofía de “convención sobre configuración”. Con una gran serie de elementos preconfigurados, un servidor web *tomcat* embebido que beneficia directamente a los desarrolladores para crear aplicaciones totalmente nuevas sin mayores preocupaciones y centrándose únicamente en la lógica del negocio y con las librerías de seguridad, gestión de base de datos y del protocolo WebSocket de manera nativa. Esto hace de SpringBoot un *framework* muy completo para el desarrollo de proyectos con arquitecturas complejas como microservicios o como una modelo vista controlador tradicional (Spring Boot, 2025).

Ventajas y desventajas de utilizar SpringBoot

Ventaja	Desventaja
Permite la integración nativa de todas las librerías pertenecientes al framework de spring. Lo que facilita agregar funcionalidades sin dependencias ni plugins externos.	Aunque <i>SpringBoot</i> es un framework que simplifica el desarrollo de software, este tiene una alta curva de aprendizaje muy alta para los nuevos desarrolladores.
Tiene una comunidad muy grande, con muchos artículos, videos y cursos que facilitan el aprendizaje de este <i>framework</i> .	La migración entre versiones del framework en muchas ocasiones resulta en la inhabilitación de métodos funcionalidades personalizadas basado en las librerías de spring que quedan marcadas como deprecadas.
Es un <i>framework</i> que permite el soporte nativo para la arquitectura de microservicios gracias a sus potentes librerías (Ganji, 2025)	Se crean muchos archivos de dependencias que no se utilizan, lo que resulta en un proyecto más pesado para levantarlo en un servidor

Tabla 3: Tabla de ventajas y desventajas de SpringBoot.

Fuente: Elaboración Propia.

2.4 ¿Qué es Grails?

Grails es un *framework open source* de desarrollo web que corre sobre la *JVM* con el lenguaje *Groovy*. Inspirado en Ruby on Rails, este framework fue lanzado en el año 2005 con el objetivo de simplificar el trabajo de desarrollar de aplicaciones en Java. Para esto Grails reutiliza todas las librerías de spring de manera nativa, sin necesidad de tener que implementarlas manualmente por medio de dependencia (Obregon, 2023). Grails funciona mediante el patrón modelo vista controlador (MVC) y al igual que los otros frameworks mencionados, también adopta la filosofía de convención sobre configuración para reducir en gran medida la complejidad de introducir *boilerplate* para realizar configuraciones manualmente. Este framework también implementa su propio *ORM*, el *GORM* (*Grails Object Relational Mapping*) para la persistencia de los datos. Y, al estar basado en SpringBoot grails aprovecha las funcionalidades que tiene SpringBoot como la inyección de dependencias (Rocher, Ledbrook, Palmer, & Brown, s.f.) o la creación de API's de manera rápida y sencilla para aquellos desarrolladores con experiencia en el framework de SpringBoot.

Ventaja	Desventaja
La curva de aprendizaje de este framework es no es muy compleja. Por lo que con la documentación oficial y conocimientos básicos sobre el patrón <i>MVC</i> es muy sencillo entender del funcionamiento de Grails.	En aplicaciones muy grandes el framework de Grails tiene muchos problemas de rendimiento tanto como en ambientes locales como <i>QA</i> , de producción y en flujos de <i>CI/CD</i> .
Grails tiene un <i>framework de testing</i> muy completo llamado <i>Spock</i> . El cual es muy útil para hacer testing más expresivo y fácil de leer por cualquiera.	Las migraciones entre versiones que se realizan suelen ser catastróficas, causando varios dolores de cabeza entre los desarrolladores.
El manejo de consultas a la base de datos mediante su <i>ORM</i> es muy fácil de aprender para nuevos desarrolladores.	Ocasionalmente al iniciar la aplicación, el framework no reconoce la clase principal para iniciar. Lo que hace que los desarrolladores deban de volver a compilar la aplicación para que funcione correctamente.

Tabla 4: Ventajas y desventajas del framework de Grails

Fuente: Elaboración propia.

Las ventajas y desventajas expuestas en esta tabla se basan en la experiencia diaria del autor como ingeniero de software en *Buk*. Una empresa que desarrolla un software integral de

recursos humanos y utiliza *Grails* como *framework* principal para su módulo de control de asistencia. Dado que la información pública sobre ciertos aspectos internos de *Grails* es limitada, estas observaciones reflejan de primera fuente los retos y beneficios reales que el autor ha encontrado al implementar nuevas funcionalidades.

2.5 Elección del framework adecuado.

La elección del *framework* adecuado para el desarrollo del *backend* y, sobre todo, del módulo de chats constituye un punto de inflexión para establecer las bases técnicas de esta implementación. En donde no solo se debe considerar la familiaridad que tienen los desarrolladores con un *framework* en específico (Parabavidez, 2024). Sino que también es primordial evaluar los requisitos propios del proyecto, la curva de aprendizaje asociada a cada tecnología, la calidad y existencia de documentación técnica, la disponibilidad de librerías afines que simplifiquen el desarrollo y que no existan instancias de realizar refactorizaciones innecesarias para adaptarlas las necesidades cambiantes del proyecto. Y, que sigan el principio de convención sobre configuración. Para esto, se hizo un análisis de las tecnologías candidatas y una tabla comparativa para la elección del *framework* más adecuado.

Criterio	Ruby on Rails	Grails	SpringBoot
Documentación técnica	Abundante, excelente y muy detallada.	Documentación reducida, pero es detallada.	Documentación muy completa, detallada y extensa.
Curva de aprendizaje	Media-Alta	Media.	Alta.
Comunidad y soporte activo	Comunidad enorme, muy activa y con una gran cantidad de recursos online.	Comunidad muy reducida, <i>framework</i> con bajo soporte y con menos actividad	Comunidad muy grande, activa y constituida. Soporte muy alto a largo plazo
Soporte para microservicios	Diseñado para MVC y API REST.	Diseñado idealmente para MVC y API REST.	soporte nativo para microservicios.

Tabla 5. Tabla comparativa para la elección del framework más adecuado al proyecto según los criterios técnicos previamente establecidos.

Fuente: Elaboración propia.

De acuerdo con los criterios previamente establecidos y evaluados, el *framework* más adecuado para los requerimientos del proyecto es *SpringBoot*. Gracias a su excelente documentación, librerías nativas muy completas, soporte nativo para microservicios asegurará una implementación escalable, robusta y mantenible a largo plazo.

2.6 Soporte nativo para el protocolo *WebSocket* y *STOMP* y elección de un *broker*.

Dado que *SpringBoot* ha sido elegido cómo el *framework* adecuado para esta implementación. Gracias a su excelente y detallada documentación, librerías nativas muy completas y soporte nativo para microservicios. En esta sección se hablará sobre las librerías que permiten la mensajería en tiempo real, muy necesaria para el módulo de chat que *BeatMap* necesita. Para esto, se abordarán las bases conceptuales y técnicas del protocolo *WebSocket* y del protocolo *STOMP* mediante la librería “*spring-boot-starter-websocket*” que permite la integración nativa de estos protocolos dentro del proyecto, sin la necesidad de depender de librerías y dependencias externas. además de la comparación entre dos intermediarios de mensajes o *brokers* para gestionar el envío de mensajes en *BeatMap*.

2.7 ¿Qué es el protocolo WebSocket?

El *WebSocket* (*RFC 6455*), es un protocolo de red que permite el intercambio de mensajes entre los clientes y el servidor por medio de una conexión *TCP*. En donde inicialmente se establece la conexión entre el cliente y el servidor por medio de un *handshake* por medio de una petición *HTTP*. A lo cual, una vez realizado el *handshake* se mantiene abierta la conexión entre el cliente y el servidor para realizar el envío de mensajes en formato *full-dúplex* (Melnikov & Fette, 2011).

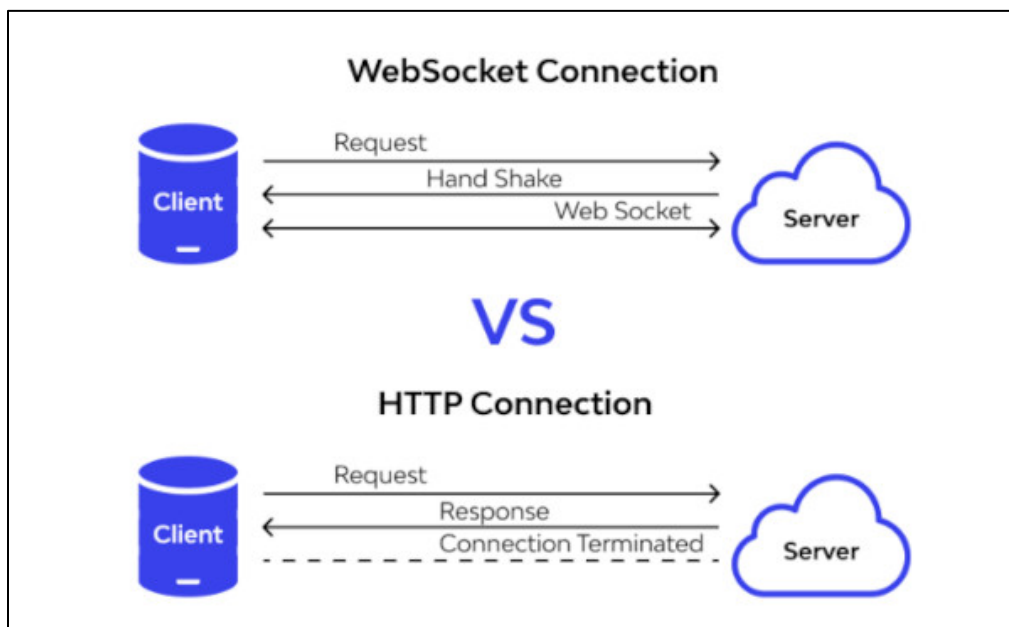


Figura 1: Funcionamiento del protocolo WebSocket vs peticiones HTTP.

Fuente: maybe.works.

Alguna de las principales ventajas del protocolo WebSocket:

- **Conexión persistente:** Una establecida la conexión entre el cliente y el servidor fue realizada correctamente, esta conexión permanece abierta. Por lo que se elimina la repetitividad de peticiones HTTP.
- **Comunicación full-duplex:** Gracias a que el protocolo WebSocket permite una comunicación de tipo *full-duplex* para el envío de datos de manera simultánea, se convierte en el mayor aliado para el módulo de chat de BeatMap.
- **Compatibilidad:** Al ser un protocolo estándar de internet, el *WebSocket* posee una gran compatibilidad entre navegadores web, lo cual es un punto alto para BeatMap para garantizar la compatibilidad entre dispositivos.

2.8 ¿Qué es el protocolo STOMP?

STOMP, por sus siglas en inglés. *Simple Text Orientated Messaging Protocol* es un protocolo de mensajería diseñado originalmente para que lenguajes de programación como Ruby, Python o Perl se pudieran comunicar con *brokers* de mensajería de empresas (IBM, 2021). *STOMP*, funciona en base a una conexión *TCP* para aprovechar la conectividad *full-duplex* de WebSocket .

2.9 ¿Qué es un intermediario de mensajes?

Un *message brokers* o intermediario de mensajes en español. Funciona como un software que permite a las aplicaciones, servicios y sistemas comunicarse e intercambiar información entre sí. Inclusive si los sistemas están escritos en tecnologías diferentes entre sí, gracias a la traducción de mensajes entre los protocolos de mensajería formales. Los intermediarios de mensajes tienen la capacidad de enrutar, almacenar, validar y entregar mensajes a los destinatarios correspondientes. Gracias a esto, los remitentes de los mensajes pueden recibir los mensajes sin saber la ubicación del emisor, si están activos o la cantidad de usuarios existentes.

Para garantizar que la entrega de los mensajes se haga de una manera confiable se complementa la integración de los intermediarios de mensajes con colas de mensajes, las cuales almacenan y ordenan los mensajes de manera que el servicio encargado del envío de mensajes pueda procesarlos y enviarlos a su destinatario (IBM, 2021).

2.10 Tipos de intermediarios de mensajes.

Dos de los intermediarios más utilizados son RabbitMQ y Apache Kafka. Los cuales son sistemas de colas de mensajería que sirven para procesar secuencias de grandes volúmenes de datos, continuos e incrementales los cuales requieren de una alta velocidad de procesamiento (AWS, 2023).

RabbitMQ:

es un software orientado a la entrega de mensajes en aplicaciones mediante el protocolo *Advanced Message Queuing Protocol*. El cual es un protocolo de mensajería que permite que las aplicaciones se puedan comunicar entre de sí de manera asíncrona, lo hace que no sea necesario que los usuarios de un chat no tengan que estar simultáneamente conectados para enviarse mensajes (Casero, 2024).

Apache Kafka:

Es otro software que sirve para la transmisión de datos en tiempo real. El cual permite, almacenar, publicar y procesar diferentes flujos de eventos, a los que los usuarios pueden suscribirse sin problema alguno. *Kafka* garantiza una tasa de latencia extremadamente baja, alta tolerancia a fallos gracias a que replica las particiones de los tópicos entre diferentes nodos. Diseñado y desarrollado originalmente por LinkedIn como una herramienta que fuera capaz de poder gestionar sus más de 1.4 billones de mensajes diarios (Red Hat, 2022).

2.11 Diferencias entre RabbitMQ y Apache Kafka y elección del intermediario de mensajería

Luego de adentrarnos en los fundamentos de los protocolos de red necesarios para el envío y recepción de mensajes, para BeatMap es ideal integrar un broker de mensajería dedicado a la distribución de mensajes, sin la necesidad de tener que crear manualmente un servicio tan complejo como RabbitMQ o Kafka. Para esto, a nivel de *backend* puede enviar los mensajes de los usuarios hacia el broker el cuál gestionará la entrega de estos hacia a los remitentes. Sin importar su estado de conexión, aprovechando los fundamentos de envío asíncrono de mensajes mediante el protocolo *Advanced Message Queuing Protocol*.

A continuación, se presenta una tabla comparativa entre RabbitMQ y Apache Kafka basada en cuatro criterios definidos por Amazon AWS, los cuales serán de gran ayuda para determinar el *broker* adecuado para BeatMap.

Criterio	Apache Kafka	RabbitMQ
Arquitectura	Kafka utiliza un diseño basado en particiones para procesar flujos de alto rendimiento y en tiempo real. Utiliza el modelo <i>pull</i> . Los productores publican mensajes en temas y particiones a los que los consumidores están suscritos.	La arquitectura de RabbitMQ está diseñada para el enrutamiento de mensajes complejos. Utiliza el modelo <i>push</i> . Los productores envían mensajes a los consumidores con reglas diferentes.
Gestión de mensajes	Los consumidores realizan un seguimiento de la recuperación de los mensajes con un rastreador de desplazamiento. Kafka conserva los mensajes de acuerdo con la política de retención. No hay prioridad en los mensajes.	Los agentes de RabbitMQ supervisan el consumo de mensajes. RabbitMQ elimina los mensajes después de consumirlos. Es compatible con la prioridad en mensajes.
Rendimiento	Kafka transmite en tiempo real hasta millones de mensajes por segundo.	RabbitMQ tiene una latencia baja. Además, envía miles de mensajes por segundo.
Lenguaje, protocolo de programación	Kafka cuenta con opciones limitadas de lenguajes de programación. Utiliza un protocolo binario sobre TCP para la transmisión de datos.	RabbitMQ admite una amplia gama de lenguajes y protocolos antiguos.

Tabla 6. Tabla comparativa para la elección del broker de mensajería adecuado.

Fuente: Amazon (AWS, 2023).

Finalmente, y de acuerdo con los criterios y requerimientos técnicos del módulo de chats, el software intermediario de mensajería más adecuado para el módulo de chat de BeatMap es RabbitMQ, el cual combinado con la librería nativa de spring “*spring-boot-starter-websocket*” para usar el protocolo *WebSocket* y *STOMP*. Hará del módulo de chat un servicio altamente robusto, escalable y fácil de mantener a largo plazo, garantizando la rapidez y alta disponibilidad para el envío y entrega de mensajes.

2.12 Elección del sistema gestor de base de datos

La elección del sistema gestor de base de datos es otra parte crucial para el desarrollo de este proyecto, dado que la base de datos es el lugar donde se almacenará toda la información de los usuarios de BeatMap. Para esto, para garantizar que la capa de persistencia cumpla con los criterios técnicos definidos (Open source, soporte a largo plazo, rendimiento). Se tienen tres sistemas gestores de bases de datos que son los candidatos los líderes a nivel mundial por su popularidad y rendimiento frente a grandes aplicaciones (Redgate Software, 2025). Los cuáles serán comparados entre sí para elegir el sistema gestor de base de datos que más se alineé con los requerimientos del proyecto.

2.13 ¿Qué es un sistema gestor de base de datos?

Por definición técnica, un sistema gestor de base de datos es un software que está compuesto por múltiples programas que permiten al usuario crear, gestionar y administrar toda la información en una base de datos. Así mismo un sistema gestor de base de datos provee a los usuarios una amplia gama de herramientas para almacenar enormes volúmenes de datos de forma ordenada, así como manteniendo la integridad de los datos mediante protocolos de seguridad mediante roles y la capacidad de realizar backups de seguridad y *rollbacks* en casos críticos (Universidad Europea, 2022).

2.14 Gestores de base más usados y candidatos principales.

- **PostgreSQL:** Sistema gestor de base de datos open-source relacional desarrollado por Michael Stonebraker en 1987 (PostgreSQL, s.f.). El cual es conocido a nivel mundial por su alta robustez y escalabilidad, soportando completamente el estándar SQL y destacando sobre todo por su excelente manejo de transacciones complejas con altos niveles de consistencia.
- **MySQL:** Sistema gestor de base de datos igualmente open-source. Muy sencillo de utilizar y con un gran desempeño para el manejo de lectura de datos. Además de contar con una comunidad muy grande y activa, hace de MySQL una opción muy viable para aplicaciones que requieren muchas consultas sencillas (datdata, 2025).
- **Microsoft SQL Server:** Sistema gestor de base de datos desarrollado por Microsoft en el año de 1989, es uno de los sistemas gestores de base de datos más utilizados en los sectores empresariales y gubernamentales, gracias su fuerte integración con toda la suite de Microsoft (Excel, Azure, .NET y Power BI) (datdata, 2025). A diferencia de PostgreSQL y MySQL requiere de una licencia comercial para implementar este sistema gestor en proyectos de software en producción.

2.15 Elección del sistema gestor de base de datos adecuado.

A continuación, se presenta una tabla comparativa entre PostgreSQL, Microsoft SQL Server y MySQL como candidatos creada por datdata, la cual será de ayuda para elegir el sistema gestor de base de datos que mejor se adecue a los criterios técnicos y requerimientos del proyecto para ser la herramienta que almacenará los datos y realizará todas las transacciones para BeatMap.

Criterio	SQL Server	MySQL	PostgreSQL
¿Es open-source?	No, requiere licencia de Microsoft para su uso	Si	Si
Soporte	Soporte muy activo por parte de la comunidad	Soporte comercial por parte de Microsoft.	Soporte muy activo por parte de la comunidad
Escalabilidad	Alta, existen versiones disponibles en la nube en <i>Azure</i>	Mediana, ideal para proyectos medianos	Alta, excelente para aplicaciones con una gran base de usuarios
Rendimiento	Alto, optimizado para entornos empresariales.	Rápido para la lectura de datos.	Óptimo para realizar una gran cantidad de transacciones.

Tabla 7: Tabla comparativa de sistemas gestores de base de datos.

Fuente: recuperada de datdata.

De acuerdo con los criterios técnicos establecidos previamente el sistema gestor de base de datos que más se adecua frente a los requerimientos de la aplicación es *PostgreSQL* es la opción más viable para BeatMap. Gracias a su alto rendimiento frente a un gran volumen de datos y transacciones en un corto periodo de tiempo, licenciamiento gratuito y soporte a largo plazo.

2.16 Elección de la arquitectura adecuada para la implementación del proyecto

En base a los criterios técnicos previamente (escalabilidad, mantenibilidad, alta disponibilidad y tolerancia a fallos). Dado estos criterios, elegir una arquitectura monolítica usando el patrón MVC no es una opción viable para la ejecución de este proyecto. Dado que este patrón acopla tanto como la lógica de negocio y las vistas en un solo directorio. Esto lleva a aumentar la complejidad de realizar refactorizaciones al código o al introducir nuevas funcionalidades más complejas a nivel de lógica. Lo cual, a largo plazo va a quedar con una deuda técnica muy grande y compleja de saldar en caso de que BeatMap comience con una expansión de usuarios a nivel latinoamericano. Y, para garantizar que la aplicación pueda soportar una carga de usuarios y peticiones tan grande es requerido implementar una arquitectura para sentar unos cimientos sólidos en el *backend*. Para eso se decidió unánimemente implementar una *arquitectura desacoplada*, separando tanto el *backend* como el *frontend* en aplicaciones independientes. Utilizando el *framework* de *java*, *SpringBoot* para el backend. Y para el *frontend* se implementará mediante el *framework* de *React*. Esta decisión temprana con una visión a futuro sobre la potencial escalabilidad del proyecto beneficiará a todo el equipo en tres aspectos claves:

1. **Separación de responsabilidades:** Gracias a que tanto como el *backend* como el *frontend* trabajan como aplicaciones independientes. El *backend* gestionará la *lógica del negocio* y el *Frontend* se centrará en la *UX/UI*.
2. **Preparación para migrar a microservicios:** El *backend* se organizará por medio de módulos independientes, la migración a microservicios se podrá realizar más rápido, gracias a que cada uno de los módulos (Chat, Usuarios, Eventos, etc.) está separado por su correspondiente caso de uso, con sus propias capas (*entity*, *controller*, *service*, *repository*, *mapper*, *DTO* y *configuration*) la migración.
3. **Flujo de trabajo más fluido:** Al separar la lógica de negocio en módulos independientes, los desarrolladores del proyecto puedan trabajar en paralelo en sus desarrollos sin tener *conflictos de merge* a la hacer un *pull request* para *merge* sus cambios.

2.17 APIS de mensajería y su capa de persistencia.

En el contexto del desarrollo de aplicaciones de mensajería, la elección del tipo de API constituye una decisión crítica, dado que de ella depende la eficiencia en la comunicación entre cliente y servidor. Existen diferentes enfoques para resolver este problema, entre los que destacan REST, WebSocket y Webhooks, cada uno con ventajas, limitaciones y escenarios de aplicación específicos. Por lo tanto, resulta pertinente describirlos y compararlos en función de los requerimientos técnicos del sistema. Para fundamentar esta decisión, es necesario establecer criterios técnicos como:

1. **Latencia:** La capacidad de la API para entregar los mensajes en el menor tiempo posible es fundamental para una aplicación que ofrece mensajería instantánea.
2. **Bidireccionalidad:** Se debe brindar la posibilidad de que el cliente y el servidor puedan intercambiar información en ambas direcciones y de manera simultánea.
3. **Escalabilidad:** El servicio de mensajería debe de soportar un aumento en la cantidad de usuarios a la aplicación sin degradar su rendimiento.

Considerando estos criterios, en los siguientes apartados se analizan las tres principales alternativas: REST, WebSocket y Webhooks, con el fin de determinar cuál es la más adecuada para el módulo de mensajería de BeatMap.

2.4.2 APIS de mensajería: REST vs WebSocket vs Webhooks

Una *API* o interfaz de programación de aplicaciones es un conjunto de reglas, protocolos y definiciones que permiten a un software comunicarse con otros sistemas, independiente de la tecnología sobre la cual estén contruidos. En la ingeniería de software, las *API's* representan una forma de reutilización y estandarización, evitando tener que “reinventar la rueda” y permitiendo que los desarrolladores accedan a funcionalidades ya hechas mediante *endpoints* estandarizados por los mantenedores de estas *API's* (Cloudflare, s.f.).

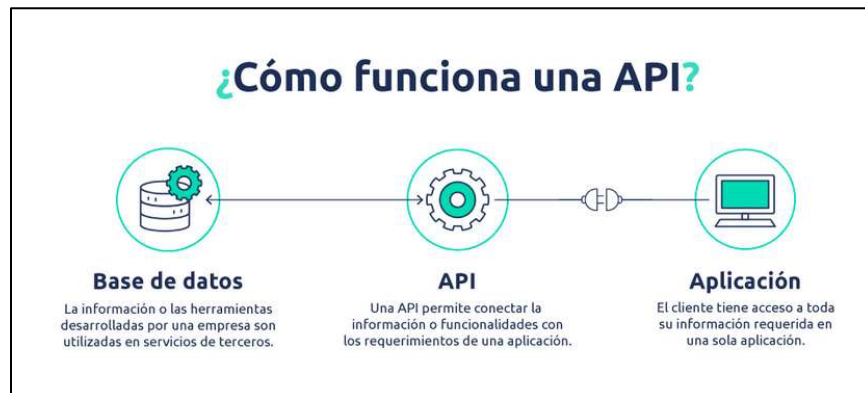


Figura 2: Funcionamiento de una API
Fuente: Outvio.

2.18 ¿Qué es una API RESTful?

Dentro de este contexto, una API RESTful se basa en la arquitectura de software REST, la cual establece condiciones como la separación cliente-servidor, la ausencia de estado. Donde la posibilidad de almacenamiento en caché y la interfaz uniforme. Estas restricciones buscan que la comunicación mediante peticiones HTTP sea escalable, flexible y fácil de mantener (Fielding, 2000). Hoy en día, tanto APIs propias como de terceros suelen implementarse siguiendo este estilo, ya sea en arquitecturas monolíticas, desacopladas o basadas en microservicios.

Ventajas y desventajas de utilizar una API RESTful

Ventajas	Desventajas
Permiten desarrollar <i>endpoints</i> para realizar <i>CRUD's</i> de manera rápida y escalable.	De realizar nuevas implementaciones en algún <i>Endpoint</i> de una <i>API</i> , puede significar en problemas de compatibilidad en aplicaciones que dependan de ella.
Una <i>API</i> debe de utilizar HTTP o HTTPS como capa de transporte. Por lo cual estandariza el uso de los verbos HTTP para el envío de información.	La falta de soporte en caso de utilizar <i>API's</i> de terceros puede significar en dificultades en implementaciones propias hasta que los mantenedores de la <i>API</i> lo solucionen.
Las <i>API'S RESTful</i> son muy versátiles de usar, dado que son admisibles en diversas aplicaciones y permiten incorporar nuevas funcionalidades son la necesidad de desarrollarlas desde cero.	Al usar peticiones HTTP en cada petición, significa un problema de rendimiento en aplicaciones con muchos usuarios y peticiones diarias.
Tiene una curva de aprendizaje baja, por lo cual hace que los nuevos desarrolladores puedan aprender más rápido su funcionamiento.	No es tan útil cuando se quiere implementar en escenarios de uso en tiempo real. Dado que no está diseñado para una comunicación bidireccional.

Tabla 8: Ventajas y desventajas de utilizar una API RESTful.

Fuente: recuperada de medium.

El uso de *APIs RESTful* dentro de la arquitectura de BeatMap, permite realizar una rápida implementación de una *API Gateway* que se encargue de centralizar todas las peticiones que se realizan desde el frontend. Esto facilitará la seguridad, el balanceo de la carga de las peticiones y el monitoreo de los servicios. además de preparar la aplicación para una migración a una arquitectura de microservicios en el futuro, según como vaya evolucionando la aplicación. Donde cada módulo (usuarios, eventos, chat) sean microservicios independientes que puedan escalar de manera independiente.

2.19 ¿Qué es una API WebSocket?

Una API WebSocket es una interfaz que permite la comunicación bidireccional y en tiempo real entre cliente y servidor a través de una única conexión TCP persistente. A diferencia de las API REST, donde cada solicitud requiere abrir y cerrar una conexión HTTP, con WebSocket se establece un *handshake* inicial y posteriormente se mantiene un canal abierto que soporta el envío y recepción de mensajes de manera simultánea. Esto la hace especialmente adecuada para aplicaciones que requieren baja latencia, como chats, videojuegos en línea o sistemas financieros (Sheldon, 2024)

Ventajas	Desventajas
Permite una comunicación bidireccional y persistente, permitiendo que el cliente y el servidor envíen datos sin la necesidad de realizar nuevas peticiones (Melnikov & Fette, 2011)	Tiene una mayor complejidad operativa, dado que requiere gestionar múltiples conexiones concurrentes y políticas de reconexión. (Sheldon, 2024)
Tiene una latencia muy baja, lo cual lo hace adecuado para aplicaciones en tiempo real como un chat o videojuegos.	Requiere de un mayor monitoreo, lo que hace necesario disponer de herramientas y métricas específicas para conexiones persistentes.
Permite integrar modelos de tipo publish/subscribe mediante el protocolo <i>STOMP</i> , simplificando en gran medida la gestión de salas y temas.	Tiene una escalabilidad compleja, lo que hace necesario disponer de balanceadores especializados o brokers de mensajería para distribuir la carga.
Tiene un menor overhead por mensaje, ya que reutiliza la misma conexión <i>TCP</i> .	Tiene una menor compatibilidad con sistemas de caché tradicionales que está diseñados para <i>HTTP</i>

Tabla 9: Ventajas y desventajas de utilizar una API WebSocket.

Fuente: Elaboración propia

Dentro de BeatMap, el protocolo *WebSocket* constituye como el candidato ideal para el módulo de chats. Ya que este protocolo proporciona una baja latencia y una comunicación persistente para los usuarios, garantizando una experiencia de uso fluida en tiempo real. Además de su integración con *STOMP* permite gestionar un gran volumen de chats mediante el procedimiento de subscripciones a tópicos.

2.20 ¿Qué son los Webhooks?

Los Webhooks, son un mecanismo de comunicación basada en eventos que permiten a una aplicación enviar datos de manera automática por medio de peticiones *HTTP* cuando ocurre una acción específica. A diferencia de una *API* basada en *REST*, *WebSocket*. Con los Webhooks no es necesario que el cliente tenga que consultar constantemente al servidor para poder obtener información, ya que este es el último que envía la información hacia su destino en el momento en que se gatilla el evento (Github, 2025) Un caso típico es la sincronización de datos en plataformas de gestión de personal, donde los Webhooks permiten replicar información sobre vacaciones o permisos entre sistemas internos y aplicaciones externas de asistencia.

Ventajas	Desventajas
Al usar <i>HTTP</i> de manera estándar, solamente es necesario exponer la URL de destino para recibir los datos.	La comunicación es unidireccional: el servidor envía los datos, pero no existe un canal de respuesta inmediato para el cliente.
Permite realizar sincronizaciones con diferentes aplicaciones externas como pasarelas de pago o notificaciones push.	Es necesario validar las solicitudes que llegan mediante <i>tokens</i> para evitar posibles fraudes o intentos de ciberataques.
No requieren de un <i>polling</i> constante, puesto que los datos serán enviados al cliente únicamente cuando exista un evento asociado.	Los Webhooks no son los más adecuados para aplicaciones que requieren de una interacción bidireccional e instantánea como un chat de mensajería.
Permiten reducir el acoplamiento del código base de la aplicación, dado que por su naturaleza permiten realizar integraciones con terceros sin la necesidad de modificar las <i>API</i> 's principales.	Si el receptor de los datos no está disponible, el evento puede perderse si es que no existen reintentos automáticos configurados.

Tabla 10: Ventajas y desventajas de utilizar una API basada en Webhooks.

Fuente: elaboración propia

Finalmente, dado la naturaleza del diseño de los Webhook, estos no son los más adecuados para el módulo de chats de BeatMap, dado que estos funcionan de manera unidireccional frente entre el cliente y el servidor. No obstante, los Webhooks en la aplicación pueden funcionar como mecanismo complementario para realizar integraciones con aplicaciones externas, por ejemplo, para notificar a un organizador de eventos cuando un artista confirma su participación para un evento o también, para notificar a un artista cuando un oyente se quiere contactar con él para una posible contratación.

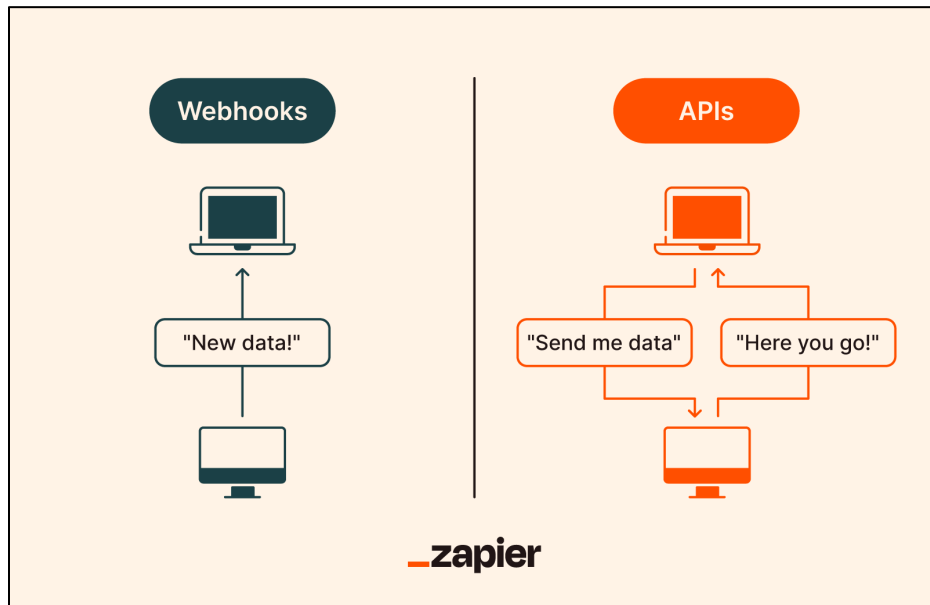


Figura 3: Funcionamiento de un webhook vs una API tradicional
Fuente: Zapier.

2.21 Selección de la API correcta para el servicio de mensajería.

Una vez revisadas las bases técnicas de las diferentes alternativas a la solución adecuada para el módulo de chats de BeatMap. Es necesario compararlas en función de criterios tales como, la comunicación, baja latencia, bidireccionalidad y adecuación al contexto de la necesidad de implementar el módulo de chats en la plataforma BeatMap. Bajo ese contexto de necesidad tecnológica, en la presente tabla se realiza la comparación para determinar el tipo de *API* correcta para el módulo de chats.

Criterio	API REST	API WebSocket	Webhooks
Latencia	Alta, por la apertura de múltiples solicitudes	Muy baja, ideal para aplicaciones en tiempo real.	Baja, pero depende de cuándo se dispare el evento.
Bidireccionalidad	No soportada nativamente.	Sí, cliente y servidor pueden enviar y recibir datos de forma simultánea.	No soportada: solo el servidor emite datos hacia el cliente.
Escalabilidad	Alta, es adecuada para arquitecturas desacopladas y microservicios.	Media-alta, requiere gestionar conexiones concurrentes y balanceo especializado (ej. brokers).	Alta en escenarios de integración con sistemas externos, pero limitada en aplicaciones críticas de tiempo real.
Caso de uso	CRUD de usuarios, gestión de recursos, APIs públicas y servicios web.	Chats, videojuegos en línea, trading financiero, notificaciones en tiempo real.	Integraciones externas, notificaciones de pago, automatización de flujos entre plataformas.
Modelo de comunicación	Basado en request/response sobre HTTP. Cada interacción abre y cierra una solicitud.	Comunicación bidireccional y persistente sobre una única conexión TCP (full-duplex).	Comunicación unidireccional basada en eventos : el servidor envía datos mediante HTTP cuando ocurre un evento.

Tabla 11: Comparativa entre una API RESTful, WebSocket y Webhooks

Fuente: elaboración propia.

A partir del análisis y la comparación realizada, resulta crucial definir correctamente la *API* para el módulo de chat, con el fin de facilitar el desarrollo técnico de la solución y garantizar la mantenibilidad a futuro. Una vez evaluadas las alternativas, se concluye que la opción más adecuada para la arquitectura del módulo de chat de BeatMap es una *API* basada en *WebSocket*, complementada con el protocolo *STOMP* para el enrutamiento de mensajes. Las principales razones de esta decisión se fundamentan en la capacidad de *WebSocket* para trabajar con baja latencia y establecer una comunicación bidireccional persistente, lo que brinda a los usuarios una experiencia de uso fluida en tiempo real, aspecto clave en aplicaciones de mensajería moderna. No obstante, una *API RESTful* continúa siendo el centro de la solución y mantiene un rol esencial en operaciones *CRUD* y gestión de la información, mientras que los *Webhooks* representan un complemento valioso para futuras integraciones con aplicaciones y sistemas de terceros. Sin embargo, ninguna de estas dos alternativas satisface por sí sola los requerimientos de instantaneidad y bidireccionalidad que demanda el chat de BeatMap. Por este motivo, se determina que la implementación de una *API WebSocket* constituye la solución más adecuada para este módulo.

Una vez establecida la elección de implementar una *API* basada en *WebSocket* como base tecnológica del módulo de chats, es necesario ir profundizando aún más en lo que es el mecanismo que permitirá enrutar, organizar y distribuir los mensajes de manera eficiente los mensajes a los usuarios. Bajo este contexto, el protocolo *STOMP* se presenta como la solución más adecuada para realizar la gestión del enrutamiento de los mensajes trabajando con el protocolo *WebSocket*. De esta forma, se reduce la complejidad de implementar manualmente configuraciones innecesarias y extensas desde el backend para poder enrutarlos. Además, el uso de *STOMP* permite integraciones con gestores de mensajería externos, como *RabbitMQ*, lo cual facilita la escalabilidad horizontal de la plataforma y prepararla para manejar mayores volúmenes de usuarios y mensajes sin comprometer el rendimiento.

2.22 Capa de persistencia de datos de mensajería

Para el desarrollo del módulo de mensajería de BeatMap, la persistencia de datos corresponde al conjunto de técnicas y herramientas que permiten que la información almacenada en el sistema permanezca disponible de forma duradera y pueda ser recuperada en cualquier momento, incluso en escenarios de fallos o reinicios del servidor. En un servicio de chat, esta característica es esencial, ya que asegura que los mensajes, usuarios, grupos y archivos multimedia queden registrados y puedan consultarse posteriormente, manteniendo la coherencia y continuidad de la comunicación. En este contexto, surge la necesidad de contar con una solución que facilite la interacción entre la aplicación y la base de datos relacional. Aquí es donde aparece el concepto de Object Relational Mapping (ORM), una técnica que permite mapear las representaciones de entidades de las bases de datos y las utilizadas en la programación orientada a objetos. En términos simples, un *ORM* funciona como un traductor que permite traducir dos técnicas para trabajar que de manera natural no son compatibles entre sí (Ellingwood, 2023). Dentro del framework Spring Boot, el ORM más utilizado es Hibernate, que implementa la especificación JPA (Java Persistence API). Esta combinación se ha consolidado como un estándar en el ecosistema Java gracias a su robustez y a la integración con librerías del propio framework. El uso de JPA/Hibernate permite:

1. Definir entidades Java como tablas de base de datos, manteniendo un modelo de objetos coherente con el modelo relacional.
2. Gestionar transacciones y relaciones entre tablas (uno a muchos, muchos a muchos) de forma sencilla y segura.
3. Implementar consultas estandarizadas mediante Criteria API o JPQL, lo que facilita operaciones complejas de recuperación de datos sin necesidad de escribir SQL nativo.

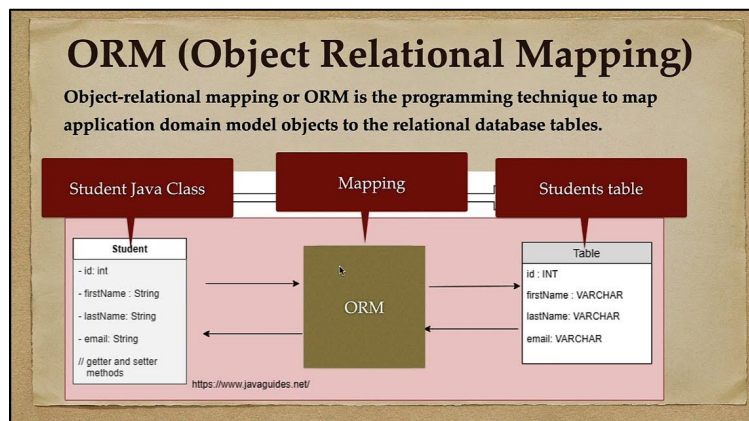


Figura 4: Funcionamiento del ORM en una aplicación hecha en Java
Fuente: Ramesh Fadattare.

Ventajas	Desventajas
Simplifica el acceso a los datos por medio de operaciones de alto nivel, omitiendo tener que realizar consultas <i>SQL</i> complejas de manera manual.	Puede realizar problemas de rendimiento en caso de realizar consultas muy complejas y con grandes volúmenes de datos.
Reduce la repetitividad del código al automatizar consultas comunes por simples métodos.	Se tiene un menor control de las consultas que se realizan, a diferencia de usar una integración con <i>JDBC</i> .
Posee una integración nativa con SpringBoot y una gran variedad de librerías.	Su curva de aprendizaje es medianamente alta, lo que requiere un alta practica para evitar realizar consultas ineficientes (por ejemplo, el problema <i>N+1</i>)

Tabla 12: Ventajas y desventajas del uso de un ORM en SpringBoot

Fuente: elaboración propia.

En BeatMap, el ORM basado en Spring Boot con JPA/Hibernate cumple un rol central en la gestión de la información persistente del sistema. Gracias a este enfoque, se pueden:

- Persistir mensajes de los usuarios en el historial de cada sala.
- Registrar usuarios y salas como entidades con sus respectivas relaciones.
- Realizar consultas eficientes como la recuperación de los últimos mensajes enviados, la búsqueda por usuario o el filtrado por fecha.
- Garantizar la consistencia transaccional, asegurando que los mensajes no se pierdan, aunque se produzca un fallo durante la comunicación.

El uso de un ORM en Spring Boot basado en JPA/Hibernate resulta ser la alternativa más adecuada para el módulo de mensajería de BeatMap, ya que permite equilibrar productividad y mantenibilidad sin sacrificar la consistencia de los datos. A través de este enfoque, se logra abstraer la complejidad del acceso a la base de datos, reduciendo errores derivados de la manipulación manual de consultas *SQL* y asegurando que las transacciones se ejecuten de manera confiable. Si bien presenta ciertas limitaciones en cuanto a rendimiento en consultas complejas o gran volumen de datos, estas pueden mitigarse con un diseño adecuado del modelo y el apoyo de mecanismos complementarios como el uso de bases de datos en caché. De esta forma, la elección de JPA/Hibernate no solo garantiza una gestión eficiente de la información persistente, sino que también prepara la aplicación para evolucionar hacia arquitecturas más escalables en el futuro.

Redis, es un motor de base de datos en memoria de código abierto. Diseñado para realizar manejos de datos de forma rápida y eficiente. Comparado con una base de datos relacional tradicional, *Redis* mantiene los datos en memoria, lo que permite poder acceder a ellos en cuestión de milisegundos, logrando un rendimiento altamente superior al de las consultas directas a la base de datos que dependen enteramente de la capacidad de lectura del disco duro. (IBM, 2024)

Su funcionalidad más común dentro de las aplicaciones modernas es de capa caché, de manera que funcione como un intermediario entre el cliente y la base de datos relacional. De manera que *redis* almacene en memoria los datos que son frecuentemente retornados desde unas consultas SQL tradicionales, reduciendo en gran medida la necesidad de acceder continuamente a la base de datos principal de la aplicación. Esto significa en gran medida una reducción de la latencia y la carga del sistema cuando múltiples usuarios consultan información con frecuencia. (Fanchi, 2022)

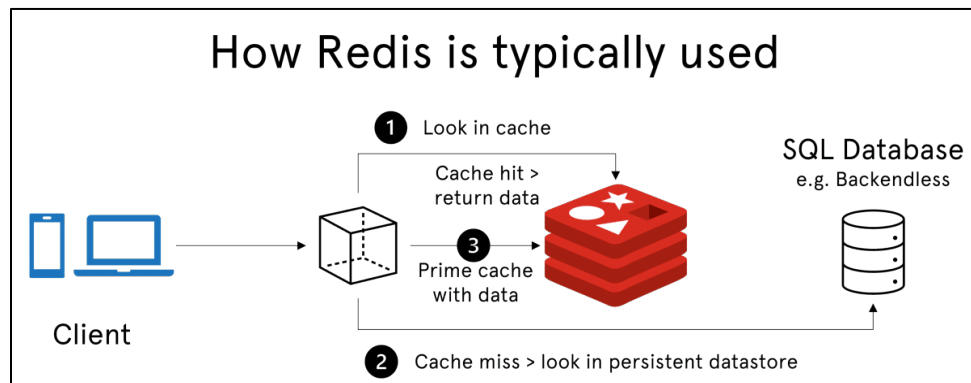


Figura 5: Funcionamiento de redis en una aplicación

Fuente: Recuperado de Backendless.

Una vez analizado el panorama general de los requerimientos del módulo de chat de BeatMap, se establece como base técnica la utilización de una API WebSocket, complementada con el protocolo STOMP para el enrutamiento de mensajes y con la posibilidad de integrarse a futuro con un message broker como RabbitMQ. Esta combinación permite una comunicación bidireccional y persistente, ofreciendo baja latencia y organizando la distribución de mensajes mediante un modelo de tópicos y suscripciones, lo que facilita tanto los mensajes privados como los intercambios en grupos.

En cuanto a la capa de persistencia, se adopta el uso del ORM de Spring Boot (JPA/Hibernate) como elemento central de la arquitectura. Gracias a este enfoque, se abstrae la complejidad del acceso a la base de datos PostgreSQL, garantizando la consistencia transaccional en el almacenamiento de mensajes, usuarios y demás entidades. Además, se incrementa la productividad y mantenibilidad del sistema a largo plazo, sin renunciar a consultas SQL nativas cuando son necesarias, y manteniendo un modelo de datos claro para operaciones típicas del chat como paginación, recuperación de historial y filtros.

Finalmente, se incorpora Redis como una capa de caché y optimización, destinada a mejorar el rendimiento de las consultas repetitivas hacia la base de datos. Bajo el patrón cache-

así, Redis permite consultar de manera inmediata los datos más demandados en la plataforma, reduciendo la carga sobre PostgreSQL, minimizando la latencia y entregando una experiencia de uso mucho más fluida al usuario final.

En conjunto, estas decisiones conforman una arquitectura de mensajería coherente y escalable, donde:

- *WebSocket*, *STOMP* y *RabbitMQ* resuelven la interacción en tiempo real y el enrutamiento de mensajes.
- *JPA/Hibernate* garantiza la persistencia fiable y una evolución mantenible del modelo de datos.
- *Redis* optimiza las lecturas y refuerza la percepción de inmediatez en el chat.

De esta manera, el marco conceptual establece los fundamentos técnicos y justifica las decisiones de diseño que guiarán la implementación en el capítulo siguiente. La elección de WebSocket con STOMP, Hibernate como ORM y Redis como caché responde directamente a los requerimientos de latencia, escalabilidad y consistencia planteados por el módulo de chat de BeatMap.

2.23 Seguridad en el módulo de chats

En el diseño y desarrollo de aplicaciones de mensajería, la seguridad constituye un componente esencial, ya que en este tipo de sistemas se intercambia de manera frecuente información sensible como datos de usuarios, archivos multimedia y conversaciones privadas. Garantizar que solo los usuarios autorizados puedan acceder a los recursos del sistema no solo protege la integridad y confidencialidad de la información, sino que también refuerza la confianza de los usuarios en la plataforma. En este contexto, los mecanismos de autenticación y autorización se vuelven fundamentales para validar la identidad de los usuarios y determinar los permisos de acceso a cada recurso del sistema. Por ello, en el diseño del módulo de chat de BeatMap resulta indispensable implementar estrategias que aseguren la comunicación sin comprometer la escalabilidad de la plataforma ni la experiencia de uso. De esta forma, se establece la necesidad de contar con un modelo de seguridad que se alinee con los principios técnicos de la solución.

2.24 Fundamentos de la autenticación y autorización.

En el ámbito de la seguridad informática, dos conceptos clave permiten controlar el acceso a los sistemas: autenticación y autorización. Aunque suelen confundirse, cumplen funciones distintas y complementarias en el diseño de aplicaciones seguras.

Autenticación: Es el proceso de verificar la identidad de un usuario o de una identidad que intenta acceder a un sistema. Este proceso puede realizarse de diferentes maneras y utilizando distintos mecanismos, tales como contraseñas, tokens de acceso, certificados digitales o autenticación por doble factor (*MFA*). En el caso de las aplicaciones modernas, la autenticación basada en token por medio de *JWT* se ha convertido en el estándar dado su facilidad de integración con diversas arquitecturas de software. (auth0, s.f.)

Autorización: La autorización define los privilegios que tiene un usuario dentro del sistema una vez autenticado. Es decir, determina a que recursos puede acceder y que tipo de operaciones puede realizar. Donde, este control suele realizarse por medio de la gestión de roles (por ejemplo: administrador, organizador de eventos, artista y oyente) y por permisos específicos asignados a cada rol. De esta manera se evita que un usuario con privilegios limitados pueda acceder a datos sensibles o modificar recursos que no le corresponden por su rol. Esto permite establecer políticas de control de acceso a los recursos. (okta, 2024)

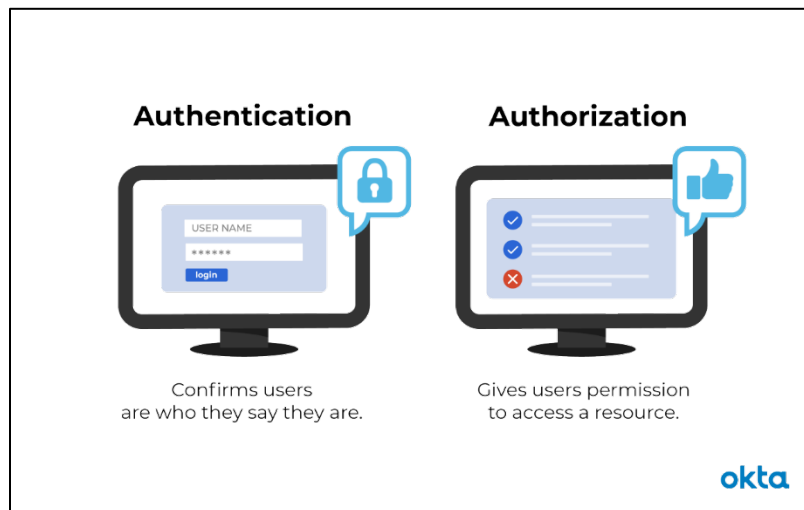


Figura 6: Diferencia entre la autenticación y autorización
Fuente: Recuperado de okta.

Para comprender mejor este concepto de la seguridad informática, se puede utilizar una analogía de una llave y una casa.

1. La **autenticación** es la llave que permite abrir la puerta de la casa. Solo la persona que posee la llave o las credenciales correctas puede ingresar al sistema.
2. La **autorización** es el equivalente a los permisos dentro de la casa. Una persona invitada puede entrar y servirse un café en la cocina, pero no necesariamente tiene permiso para acceder al dormitorio principal, el cual está reservado únicamente para el propietario.

De esta forma, la autenticación asegura que el usuario es quien dice ser, mientras que la autorización define qué recursos y acciones puede realizar una vez dentro del sistema.

Aspecto	Autenticación	Autorización
Función	Verifica las credenciales de acceso al sistema	Concede o niega acciones en base a los permisos del usuario.
Mecanismos	Uso por medio de contraseñas, tokens de acceso	Definidos por los equipos de seguridad y de desarrollo
Visible para el usuario	Sí	No, es de manera interna
Control del usuario	Parcial (Cambiar la contraseña del usuario)	No, depende de las políticas definidas para el sistema
Movimiento de datos	Tokens de identidad	Tokens de acceso

Tabla 13: Diferencias entre autenticación y autorización
Fuente: Recuperada de okta.

2.25 Autenticación basada en token JWT.

En el desarrollo de aplicaciones modernas, uno de los mecanismos de autenticación más utilizados es el JSON Web Token (JWT), un estándar abierto diseñado para transmitir información de forma segura entre dos partes como un objeto JSON firmado digitalmente. A diferencia de las sesiones tradicionales basadas en cookies, JWT no requiere mantener el estado en el servidor, lo que lo convierte en un enfoque más ligero, escalable y adecuado para arquitecturas distribuidas y microservicios (Auth0, s.f.).

2.26 ¿Cómo está compuesto un token de JWT?

Un token de JWT está compuesto por tres piezas fundamentales:

1. **Header:** Especifica el tipo de token y el algoritmo que utiliza para encriptar la información.
2. **Payload:** Contiene los *claims* o datos del usuario, id, fecha de expiración del token, entre otros
3. **Signature:** Es el resultado de firmar el Header con el Payload mediante un par de claves públicas y privadas secretas. Por lo cual, si una de las dos claves se altera la firma será considerada inválida y el token no es válido.

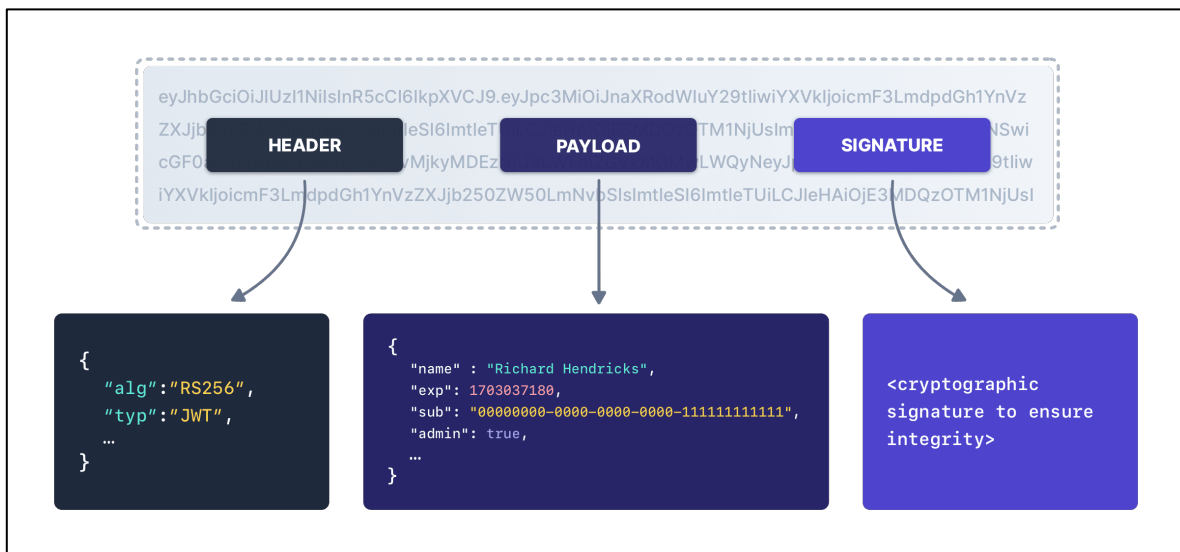


Figura 7: Estructura de JWT

Fuente: recuperado de Dev tools, hecho por Daniel DeGroff.

2.27 Ventajas y desventajas de utilizar JWT.

Si bien JWT ofrece múltiples beneficios en términos de escalabilidad y seguridad, también presenta ciertos desafíos que deben ser considerados en su implementación. A continuación, se muestran las principales ventajas y desventajas de este mecanismo de autenticación:

Ventaja	Desventaja
El token incluye toda la información necesaria, por lo que mejora la escalabilidad y reduce la carga de solicitudes constantes al servidor.	En un sistema con múltiples solicitudes, JWT tiende a ser más grande que un id de sesión normal, lo cual puede afectar al rendimiento de la aplicación.
Al ser un estándar abierto, es ampliamente soportado en diferentes lenguajes de programación y frameworks para su capa de seguridad.	Una vez emitido un token, este es válido hasta su tiempo de expiración, por lo cual es necesario diseñar un mecanismo como listas negras de tokens en caso de querer invalidar token de atacantes.
Al usar llaves públicas y privadas se garantiza que el contenido del token no ha sido alterado.	Si un token se filtra, es importante tener un manejo de tiempo y de revocación de tokens en caso de caer en las manos equivocadas.

Tabla 14: Ventajas y desventajas de JWT.

Fuente: Elaboración propia

En el módulo de chats de BeatMap, la autenticación basada en JWT se integra con Spring Security para garantizar que solo los usuarios autorizados puedan enviar y recibir mensajes dentro de la aplicación. El flujo funciona de manera que al iniciar sesión correctamente, el backend genera un token JWT firmado digitalmente que incluye los datos esenciales del usuario, como su id, roles (administrador, organizador, artista, oyente) y el tiempo de expiración del token. Posteriormente, este token es almacenado en el frontend y adjuntado en cada petición que realice el cliente, tanto para enviar mensajes a través de la API WebSocket como para acceder al historial de conversaciones o archivos multimedia asociados. El servidor valida la firma y vigencia del token antes de permitir la acción, asegurando que únicamente los usuarios con credenciales válidas puedan participar en el chat o acceder a los mensajes almacenados. De esta manera, JWT no solo garantiza la seguridad en la comunicación entre cliente y servidor, sino que también aporta escalabilidad, ya que elimina la necesidad de mantener sesiones activas en memoria del servidor, optimizando así el rendimiento del sistema de mensajería en tiempo real.

2.28 El rol de SpringSecurity con JWT.

Si bien la autenticación mediante JSON Web Tokens (JWT) constituye un mecanismo eficaz para validar la identidad de los usuarios en aplicaciones modernas, en un sistema de mensajería en tiempo real resulta indispensable incorporar un enfoque más amplio de seguridad. En este punto entra en juego Spring Security, un framework altamente extensible que complementa la autenticación basada en JWT con una capa de autorización, control de acceso y protección de endpoints.

Spring Security funciona a través de una cadena de filtros (Security Filter Chain) que intercepta cada petición HTTP o suscripción WebSocket antes de que llegue a la lógica de negocio. Esto desacopla la seguridad de la aplicación del resto de la arquitectura, permitiendo que la validación se realice de manera centralizada y consistente. De este modo, se garantiza que tanto las solicitudes de creación de chats internos o el envío de archivos como las acciones de suscripción o envío de mensajes en tiempo real (WebSocket) sean verificadas antes de ejecutarse.

Los filtros de Spring Security permiten:

- Validar credenciales de acceso: comprobando tokens JWT en cada petición.
- Verificar permisos asociados al rol del usuario: asegurando que solo los usuarios autorizados accedan a recursos sensibles o endpoints administrativos.
- Aplicar políticas de seguridad adicionales: tales como protección contra ataques CSRF, validación de cabeceras o control de sesiones.

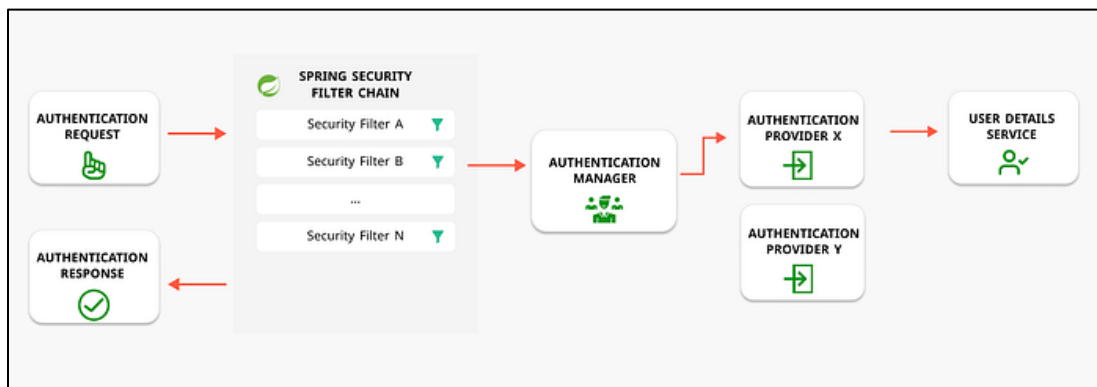


Figura 8: Funcionamiento de la cadena de filtros de SpringSecurity

Fuente: Elaborada por Kasun Dissanayake

Gracias a este esquema, cada petición que atraviesa la aplicación es verificada y controlada sin necesidad de modificar directamente los controladores del sistema, manteniendo una separación clara entre la lógica de seguridad y la lógica de negocio.

2.29 Integración de JWT con Spring Security en BeatMap.

Finalmente, en BeatMap la seguridad no se limita únicamente a verificar la identidad de los usuarios, sino que también contempla la definición de las acciones y vistas que cada rol está autorizado a utilizar dentro de la plataforma. Para ello, Spring Security complementa la autenticación basada en JWT con un sistema de autorización definido por roles y permisos. De esta forma, cada usuario autenticado recibe un rol específico que determina el alcance de sus operaciones dentro del sistema, habilitando o restringiendo tanto el acceso a secciones de la interfaz como la posibilidad de ejecutar determinadas acciones:

- **Organizador de eventos:** puede crear, editar y eliminar sus propios eventos, además de publicar postulaciones para que artistas se inscriban. También dispone de chats internos con los artistas aceptados para coordinar la organización de cada evento.
- **Artista:** tiene permisos para postularse a eventos publicados, enviar mensajes y compartir archivos multimedia en los chats habilitados por los organizadores de los eventos en los que participa.
- **Oyente:** cuenta con acceso restringido al *tracker* de eventos, pudiendo visualizar las publicaciones realizadas por los organizadores, pero sin privilegios de interacción avanzada ni acceso a información sensible.
- **Administrador:** posee acceso completo a la plataforma, incluyendo la supervisión de usuarios, la gestión de roles y el monitoreo del correcto funcionamiento del sistema.

De esta manera, aunque los usuarios cuenten con un token JWT válido, sus acciones dentro de la aplicación estarán reguladas por las reglas de autorización definidas en Spring Security.

Mecanismo	Función Principal	Uso en BeatMap
JWT (Autenticación)	Se encarga de verificar la identidad de los usuarios	Genera un token con un id asociado, rol y expiración
Spring Security (Autorización)	Define los procesos y controla los accesos.	Restringe uso de endpoints, vistas o canales WebSocket.

Tabla 15: función y uso en BeatMap de Spring Security y JWT

Fuente: Elaboración propia

La combinación de JWT y Spring Security constituye un enfoque integral de seguridad para el módulo de chat de BeatMap, garantizando tanto la verificación de la identidad de los usuarios como la delimitación precisa de sus permisos. Este modelo resulta fundamental en sistemas de mensajería en tiempo real, donde se gestionan datos sensibles como conversaciones privadas y archivos multimedia. En consecuencia, asegurar de forma conjunta la autenticación y la autorización incrementa la confianza de los usuarios en la plataforma, al tiempo que fortalece la robustez y escalabilidad de la solución propuesta.

2.30 ¿Qué son las pruebas de software?

En ingeniería de software, las pruebas de software son el proceso de evaluar y verificar que las funcionalidades de una aplicación de software funcionan de manera eficiente y segura, de acuerdo con los requisitos predefinidos. Su propósito no solo se limita a la detección temprana de errores, sino que también a prevenir defectos, casos bordes y garantizar la calidad del software desarrollado al cliente. (Susnjara & Smalley, 2025) Hoy en día, el testing está integrado en las prácticas modernas de desarrollo de software, incluyendo las metodologías ágiles, la filosofía DevOps y los procesos de CI/CD. Dejando de ser solamente el fin del ciclo de vida del software, sino que evolucionó a ser un proceso continuo que se considera desde el diseño de una funcionalidad, hasta el despliegue a producción. Este enfoque hoy en día es conocido como shift-left, el cual permite a los equipos de desarrollo identificar problemas en etapas tempranas de desarrollo.

A grandes rasgos existen dos tipos generales de pruebas de software:

1. **Pruebas manuales:** Son pruebas realizadas por personas que validan el funcionamiento de las funcionalidades siguiendo casos de pruebas establecidos y probando como usuarios finales. También conocidas como procesos de UAT.
2. **Pruebas automatizadas:** Son pruebas implementadas a nivel de código mediante librerías y frameworks de testing que permiten validar de manera repetitiva funcionalidades del software.

2.31 La pirámide de Cohn en las pruebas de software.

La pirámide de pruebas de Cohn, propuesta en el año 2009 por el ingeniero de software Mike Cohn en su libro *Succeeding with Agile*, se ha consolidado como un estándar en la industria tecnológica para estructurar las estrategias de testing en proyectos de software. Este modelo establece una jerarquía de niveles de prueba:

1. **Pruebas Unitarias:** Validan de manera aislada el correcto funcionamiento de componentes individuales de un software, como servicios o métodos de una clase.
2. **Pruebas de Integración:** Comprueban que los servicios o módulos de una aplicación interactúen correctamente entre sí.
3. **Pruebas de interfaz gráfica:** Evalúan el funcionamiento del sistema desde la perspectiva de un usuario final, asegurando que la aplicación cumple con los requerimientos funcionales y casos de uso.

El modelo enfatiza principalmente las pruebas unitarias, seguidas de las pruebas de integración, y en menor proporción las pruebas de interfaz gráfica. La lógica detrás de esta disposición es que las pruebas más cercanas al código son menos costosas y rápidas de ejecutar, mientras que las pruebas en niveles superiores, aunque necesarias, son más lentas y costosas. En consecuencia, la pirámide de Cohn no solo busca garantizar la cobertura y calidad del software, sino también optimizar los recursos y reducir riesgos en entornos de desarrollo ágiles y de integración continua. (Lgarciacantillano, 2022)

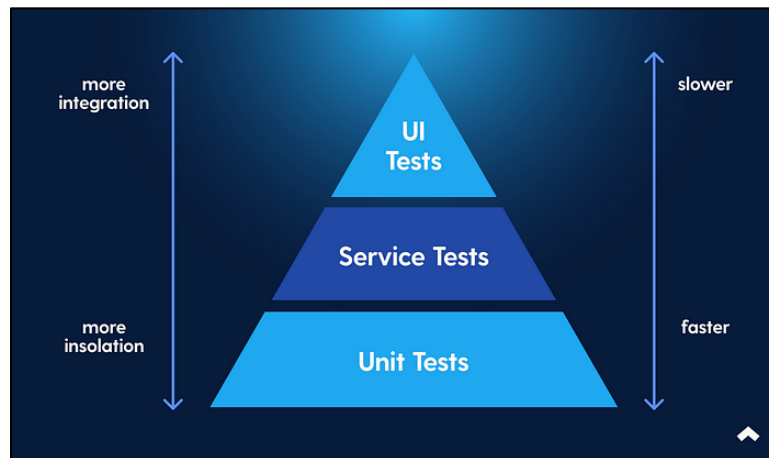


Figura 9: Modelo original de Mike Cohn de La Pirámide de Pruebas Ágiles.
Fuente: Elaborada por Mike Cohn.

2.32 Ventajas y desventajas del testing para el desarrollo de software.

El testing de software constituye una de las prácticas más relevantes dentro de la ingeniería de software, ya que permite garantizar la calidad, confiabilidad y escalabilidad de los sistemas antes de ser liberados al usuario final. Sin embargo, implementar un plan de pruebas también implica desafíos asociados a la inversión de tiempo, recursos y a la necesidad de mantener una estrategia clara de priorización.

En el caso de *BeatMap*, donde el módulo de mensajería instantánea debe asegurar comunicaciones en tiempo real y manejo de datos sensibles, resulta indispensable ponderar los beneficios y limitaciones del testing para diseñar un estándar acorde al proyecto.

Aspecto	Ventajas	Desventajas
Calidad del software	Permite la detección temprana de errores y de defectos en producción.	Requiere una planificación exhaustiva para desarrollar un buen plan de pruebas.
Mantenibilidad	Favorece la escalabilidad del sistema a largo plazo.	Mantener pruebas desactualizadas puede provocar problemas de deuda técnica
Integración continua	Permite la entrega continua y seguras, reduciendo el riesgo de lanzamiento de nuevas funcionalidades previamente testeadas.	Puede ralentizar el ciclo de desarrollo de una funcionalidad si no se prioriza adecuadamente los tipos de pruebas a realizar.
Experiencia del usuario	Aumenta la confianza del usuario final al validar que las funcionalidades cumplen con los requerimientos planteados.	Los procesos de pruebas de interfaz gráfica o de UAT pueden ser lentos y costosos en comparación a las pruebas unitarias.

Tabla 16: Ventajas y desventajas de las pruebas de software.

Fuente: Elaboración propia

El testing ofrece grandes ventajas en términos de robustez y fiabilidad del sistema a largo plazo. No obstante, su efectividad depende directamente de la calidad de los casos de prueba definidos para las funcionalidades.

2.33 Librerías de Java recomendadas para testing.

En el ecosistema de desarrollo de aplicaciones basadas en *Java*, y más aún en aplicaciones que utilizan el *framework* de *Spring Boot*, existen varias librerías y *frameworks* que permiten implementar distintas clases de pruebas (unitarias, de integración, aceptación). A continuación, se presentan algunas de las más utilizadas y se explica el criterio de selección para *BeatMap*, considerando como base de las pruebas la pirámide de Cohn.

Librería	Ventajas	Desventajas
JUnit	Librería estándar en Java, permite integración nativa con SpringBoot.	Por sí solo no puede simular servicios o entidades, requiere de librerías complementarias.
Mockito	Facilita la simulación de servicios y entidades para escribir tests de manera sencilla.	No reemplaza pruebas de integración, es más un complemento para escribir tests.
Spock	Sintaxis muy expresiva, integra el proceso de hacer <i>Mocks</i> sin depender de librerías externas.	Los desarrolladores deben tener conocimientos de Groovy, por lo cual la curva de aprendizaje para los equipos puede ser alta.
Cucumber	Permite escribir tests con lenguaje natural, lo que facilita la lectura de estos.	Si se aplica en exceso puede generar sobrecargas y problemas de rendimiento al servidor.
TestContainers	Levanta contenedores de Docker de diferentes servicios para realizar pruebas realistas.	Requiere conocimientos de Docker y el peso de las imágenes puede alargar el tiempo de ejecución de los tests.

Tabla 17: Ventajas y desventajas de las pruebas las distintas librerías de testing.

Fuente: Elaboración propia

La evaluación de ventajas y desventajas del testing y la comparación de las principales librerías de pruebas en *Java*, permite establecer un estándar adaptado a las necesidades de *BeatMap*. Se definió que cada nuevo desarrollo deberá incluir pruebas unitarias e integración implementadas con *JUnit* y *Mockito*, garantizando la validación técnica de los componentes del sistema. Además, como parte del flujo interno de desarrollo, todo *Pull Request* deberá ser revisado mediante una prueba manual de interfaz gráfica (*GUI*) para validar el correcto funcionamiento de los nuevos desarrollos.

III. CAPÍTULO 3: PROPUESTA DE SOLUCION

3.1 Diseño de la arquitectura del proyecto y del módulo de chat.

A continuación, se presentan dos diseños de arquitectura que se complementan entre sí para el diseño de *BeatMap*, teniendo su principal foco con el módulo de chat. Para esto, en primera instancia se presenta la interacción entre los componentes principales de la aplicación (*Back-end*, *Front-end*, base de datos) para enseñar la arquitectura global del proyecto, respetando las decisiones técnicas para la elección de las tecnologías que se implementarán por el lado del *backend*. Posteriormente, se detalla el diseño del módulo de chat de esta implementación.

3.2 Diseño de la arquitectura general de BeatMap.

Este diagrama muestra la interacción entre los contenedores de *Docker* que componen la aplicación de *BeatMap*. Siguiendo los lineamientos técnicos establecidos previamente, se definió una arquitectura desacoplada que separa completamente el *Backend* del *Frontend*, garantizando escalabilidad, mantenibilidad y una mejor gestión del ciclo de desarrollo mediante integración continua. El Frontend de *BeatMap* fue desarrollado utilizando *React*, un *framework* que permite la construcción de componentes reutilizables y la creación de interfaces dinámicas y ligeras para los usuarios. Por su parte, el Backend fue implementado con el *framework* de *Java Spring Boot*, el cual proporciona un entorno robusto, modular y preparado para la creación de *APIs RESTful* y servicios *WebSocket*. Para la capa de persistencia, se seleccionó el motor de base de datos *PostgreSQL* debido a su confiabilidad y soporte para transacciones complejas, complementado con *Redis* como base de datos en caché, con el fin de optimizar el rendimiento en consultas frecuentes. Finalmente, en el área de *DevOps*, se integró *Jenkins* para la automatización de despliegues y la ejecución de *pipelines* de integración continua.

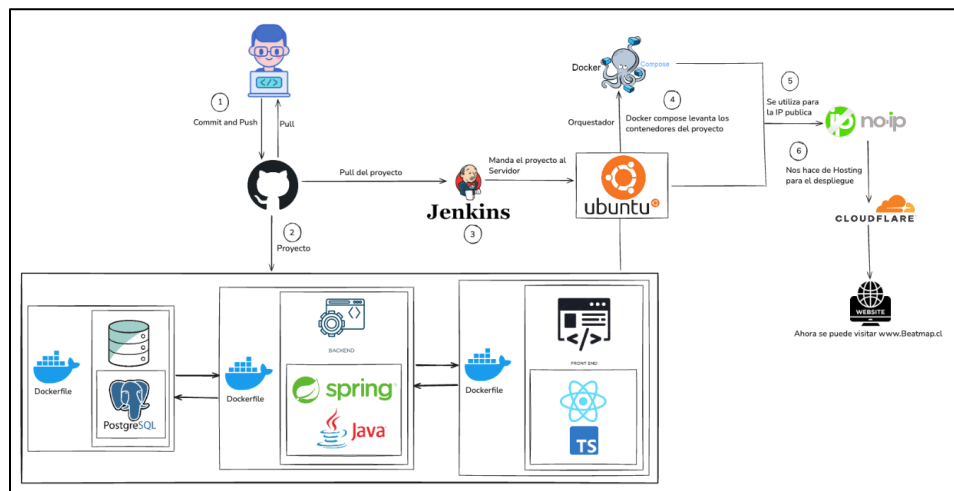


Figura 10: Arquitectura general del proyecto

Fuente: Elaborado por Dismar Quiroz

El enfoque de esta arquitectura no solo simplifica la administración y el mantenimiento de la plataforma, sino que también prepara a BeatMap para escalar progresivamente conforme la masa crítica de usuarios aumente con el tiempo. Una vez alcanzado dicho umbral, esta estructura permitirá una transición eficiente hacia una arquitectura del Backend basada en microservicios, sin comprometer la estabilidad ni la disponibilidad del actual de la plataforma.

3.3 Diseño de la arquitectura del módulo de chat de BeatMap.

El módulo de chat de BeatMap fue diseñado bajo una arquitectura basada en comunicación en tiempo real, implementando el protocolo *WebSocket* para el intercambio bidireccional de mensajes y complementando la solución con *STOMP* para el enrutamiento de mensajes. Esta elección tecnológica responde a la necesidad de mantener una conexión persistente, segura y de baja latencia entre los usuarios de la plataforma, los cuales son elementos fundamentales para garantizar la inmediatez y fluidez del servicio de mensajería. Estos factores constituyen aspectos claves para diseñar y construir una solución escalable y de alto rendimiento dentro de la plataforma.

Cada mensaje transmitido a través del protocolo *STOMP*, el cual se encarga de gestionar la suscripción a los distintos tópicos. Esto permite que múltiples usuarios puedan enviar mensajes dentro de un mismo canal sin interferir con otros flujos de comunicación de los usuarios. Además, para mejorar la escalabilidad de la plataforma se integró el gestor de colas *RabbitMQ*, el cual se encarga de enrutar los mensajes hacia el destinatario correspondiente y de manera asíncrona, garantizando una alta disponibilidad con una masa crítica de usuarios. En cuanto a la persistencia, cada mensaje enviado por los usuarios será almacenado en la base de datos *PostgreSQL*, por medio del *ORM* de *SpringBoot* que garantiza la persistencia de los datos y a recuperación del historial de las conversaciones de los usuarios. Y, finalmente para optimizar el rendimiento de la base de datos se integra Redis para la capa de caché de la aplicación, lo que permite acceder de manera inmediata a los mensajes más recientes luego de consultar a la base de datos por primera vez.

3.4 Implementación del servicio de mensajería: modelo de datos, endpoints y persistencia

El modelo entidad-relación del módulo de mensajería de BeatMap fue diseñado para soportar la comunicación en tiempo real entre los usuarios, gestionando un alto volumen de mensajes y garantizando la trazabilidad completa de las conversaciones. Para esto, la implementación de la base de datos se realizó sobre *PostgreSQL*, gracias a su alto soporte, robustez en soluciones complejas, alto rendimiento y cumplimiento de los principios *ACID*. Lo que garantiza la integridad, consistencia y persistencia de los datos. Además, *PostgreSQL* permite una integración nativa con *PostGIS*, lo que resulta ventajoso para BeatMap al poder posibilitar la gestión de información geoespacial vinculada la ubicación de los eventos publicados en el tracker de BeatMap. Este modelo entidad-relación define las relaciones fundamentales entre los usuarios, chats y mensajes. Incorporando componentes claves orientados para la trazabilidad, como el historial de mensajes enviados y los estados de

lectura de estos, lo que refuerza la confiabilidad y garantiza la escalabilidad a futuro del servicio de mensajería. A continuación, se presenta el modelo entidad-relación del módulo de mensajería, el cual constituye las bases de la capa de persistencia para orientar una solución escalable para el *Back-end*.

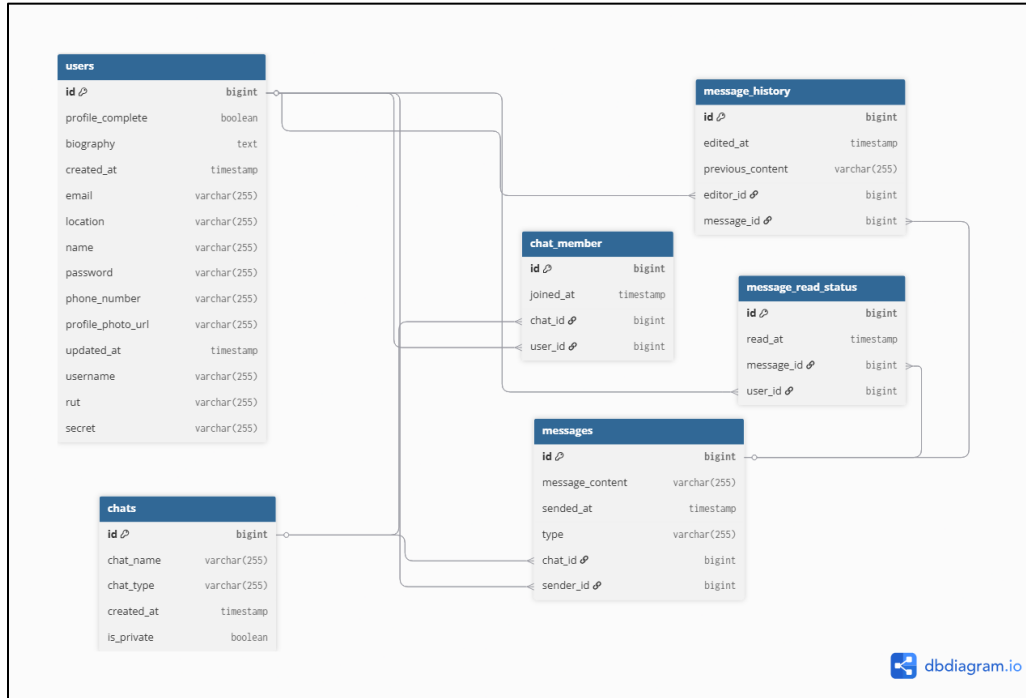


Figura 11: Modelo de datos del módulo de chat de BeatMap

Fuente: Elaboración propia

3.5 Descripción de las entidades del módulo de mensajería.

Actualmente, el modelo se compone de seis entidades principales que estructuran la base de datos del servicio de mensajería y las cuales definen las relaciones entre los usuarios, chats y mensajes.

1. **Users:** Almacena los datos generales de los usuarios registrados en la plataforma, incluyendo su información de autenticación correspondiente, perfil y datos relevantes como el email, Rut, username y biografía. Esta tabla está relacionada con los mensajes enviados, los estados de lectura de estos y la membresía a los chats con otros usuarios.
2. **Chats:** Representa las salas de mensajerías 1 a 1. En donde los chats pueden ser públicos o privados. Pero para la primera versión de la plataforma los chats por defecto son privados, esto es habilitado por medio del atributo *is_private*, el cual es un valor booleano que define la privacidad del chat.

3. **Chat_member:** Tabla intermedia entre *Users* y *chats* que registra la pertenencia de un usuario a un chat, junto con la fecha en la que el usuario se unió al chat por medio del atributo *joined_at*.
4. **Message:** Registra todos los mensajes enviados por los usuarios dentro de la plataforma, dejando un registro del contenido del mensaje, el tipo de mensaje, aunque en esta instancia solo está habilitado el envío de mensajes de texto solamente. Pero se deja preparado el modelo para la futura inclusión de archivos multimedia en la plataforma.
5. **Message_history:** Almacena las versiones previas de los mensajes editados por el usuario, junto con el id del usuario que editó dicho mensaje y la fecha en la cual editó el mensaje.
6. **Message_read_status:** Permite registrar el momento en el que un usuario leyó un mensaje que le fue enviado. Relacionando cada mensaje con su correspondiente lector mediante atributos *message_id*, *user_id* y fecha en que se leyó dicho mensaje. Gracias a esto, se puede tener un mayor control y permite mejorar la experiencia del usuario al ofrecer indicadores de lectura de los mensajes en tiempo real.

En conjunto, estas entidades conforman la capa de persistencia del módulo de mensajería de BeatMap, sobre la cual se construye la lógica de negocio y los endpoints del sistema. Esta estructura permite una comunicación eficiente con la capa de aplicación, garantizando escalabilidad, consistencia y trazabilidad en tiempo real mediante *Spring Boot* y *WebSocket*, que constituyen los pilares tecnológicos de esta solución.

3.6 Flujo de comunicación en tiempo real del servicio de mensajería

El servicio de mensajería de BeatMap fue diseñado bajo una arquitectura orientada a la comunicación en tiempo real con el objetivo de mantener una comunicación bidireccional, persistente y de baja latencia entre los usuarios. Para lograr esto, se implementó el protocolo *WebSocket* en conjunto de *STOMP* para el envío de mensajes de manera asíncrona, permitiendo la transmisión eficiente de mensajes en tiempo real y la gestión de canales temáticos mediante tópicos por medio de *RabbitMQ*. El flujo de comunicación sigue una secuencia lógica:

1. **Establecimiento de la conexión:** El cliente inicia una conexión *Websocket* con el servidor a través de la *API Gateway*. En el *Endpoint* "*ws://localhost:8000/dm*". Y por el lado de la autenticación esta se realiza mediante un token *JWT* enviado en la cabecera de la solicitud de la conexión
2. **Suscripción a Tópicos:** Una vez validada la identidad del usuario, la conexión persistente permite al cliente suscribirse a los tópicos de interés. Por lo que, para

recibir mensajes de un chat en específico, el cliente se suscribe al destino del *broker* cuya ruta es */topic/chat.{chatId}*, donde *{chatId}* es el identificador único del chat.

3. **Publicación del mensaje:** Para enviar un mensaje, el cliente lo publica en un destino del *backend* con el prefijo */app*, específicamente en la ruta *app/chat/{chatId}/send*. Un método anotado mediante la anotación *@MessageMapping* en el controlador del backend que recibe y procesa dicho mensaje.
4. **Enrutamiento en el Broker:** El *backend* tras procesar el mensaje, lo reenvía al broker de mensajería *RabbitMQ*. Este utiliza un *Exchange* de tipo *topic* para enrutar el mensaje hacia las colas correspondientes. Donde el *routing key* dinámico *chat.{chatId}* asegura que el mensaje se dirija exclusivamente a la cola *queue.chat.{chatId}*, a la que están vinculados los suscriptores de ese chat.
5. **Procesamiento y persistencia:** Los consumidores en el servicio del *backend*, suscritos a las colas de *RabbitMQ* procesan los mensajes entrantes. Cada mensaje es almacenado de forma persistente en la base de datos *PostgreSQL* para garantizar la persistencia y la recuperación del historial de los mensajes.
6. **Entrega a suscriptores:** Finalmente, *RabbitMQ* distribuye el mensaje a todos los clientes suscritos al tópico correspondiente por medio de la ruta *"/topic/chat.{chatId}"*. Estos reciben la actualización en tiempo real a través de su conexión *WebSocket*, incluyendo la información del remitente y el contenido del mensaje. Este mismo mecanismo se utiliza para notificar eventos como "usuario escribiendo" o confirmaciones de lectura de los mensajes enviados.

En este flujo, se presenta como un usuario inicia una conexión de *WebSocket* y *STOMP* a través de la *API Gateway*, validando su autenticación por medio de *JWT*. Donde, una vez establecida la conexión, el mensaje es emitido desde el lado del cliente en dirección al servicio de mensajería del Backend, el cual enruta hacia *RabbitMQ* para su procesamiento asíncrono. Posteriormente, el mensaje es persistido en la base de datos mediante *JPA* y almacenado temporalmente en *Redis* y, finalmente distribuido a todos los clientes suscritos al canal correspondiente.

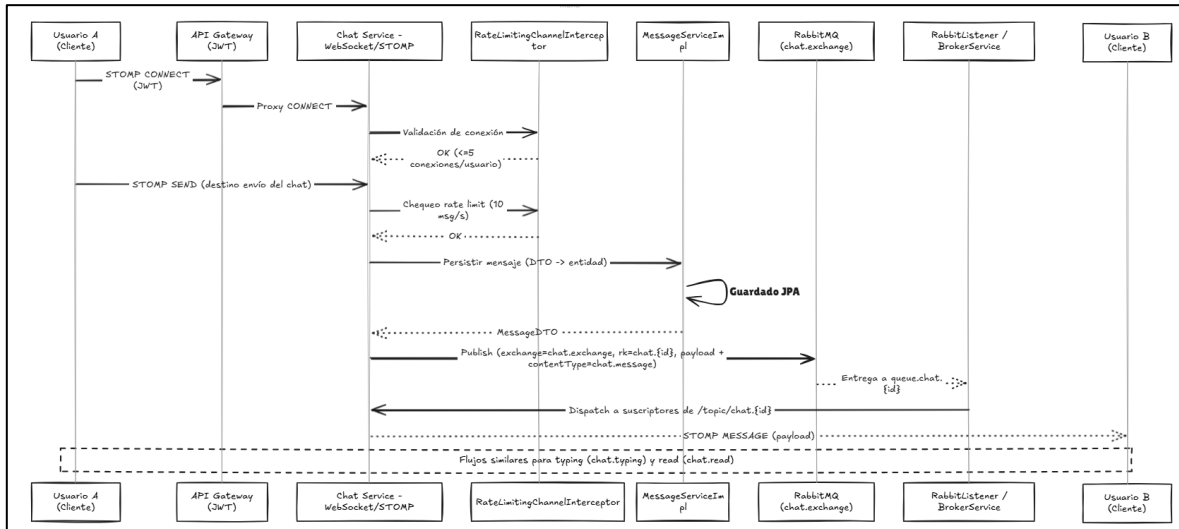


Figura 12: Diagrama de secuencia del módulo de chat de BeatMap

Fuente: Elaboración propia.

Este diagrama evidencia la interacción entre las capas del sistema y demuestra cómo el servicio de mensajería mantiene la consistencia, baja latencia y la tolerancia a fallos al combinar de manera asíncrona los componentes de *Spring Boot*, *RabbitMQ*, *Redis* y *PostgreSQL*. Además, refuerza el cumplimiento de los principios de escalabilidad horizontal y comunicación en tiempo real que sustentan la arquitectura de BeatMap.

3.7 Endpoints del servicio de mensajería.

El módulo de mensajería de BeatMap expone un conjunto de *endpoints HTTP* y *WebSocket* diseñados para gestionar la comunicación entre usuarios y garantizar la trazabilidad de los mensajes enviados, editados y leídos dentro de la plataforma. Estos *endpoints* forman la interfaz principal entre el cliente y el servidor, y le permiten interactuar con la capa de persistencia y los servicios de mensajería en tiempo real, implementados con *Spring Boot*, *STOMP*, *RabbitMQ* y *PostgreSQL*. Donde los *endpoints HTTP* se utilizan para la gestión estructurada de los recursos asociados a los chats y mensajes, permitiendo realizar operaciones *CRUD* de manera segura mediante autenticación con *JWT*. A continuación, se presentan los principales *endpoints* implementados en el servicio.

Método	Verbo HTTP	Path del Endpoint	Descripción del Endpoint
<i>getOrCreateDmChat</i>	GET	/chat/direct/{submitterUserId}	Obtiene o crea un chat directo entre dos usuarios.
<i>createPrivateChat</i>	POST	/chat	Crea una nueva sala de chat privado o grupal.
<i>getMessageHistory</i>	GET	/message/{chatId}/messages	Recupera el historial completo de mensajes de un chat.
<i>getUnreadMessages</i>	GET	/message/unread/{chatId}	Devuelve los mensajes no leídos por el usuario autenticado.
<i>enqueueMessage</i>	POST	/message	Encola un nuevo mensaje para ser procesado y distribuido por <i>WebSocket</i> .
<i>enqueueEdit</i>	PUT	/message/{id}	Edita el contenido de un mensaje y registra el cambio en la entidad <i>Message_history</i> .
<i>enqueueDelete</i>	DELETE	/message/{id}	Elimina un mensaje, manteniendo la trazabilidad del cambio en la base de datos.

Tabla 18: Listado de endpoints del módulo de mensajería.

Fuente: Elaboración propia

La implementación del servicio de mensajería de *BeatMap* integra de manera cohesiva entre las capas de persistencia, comunicación y distribución de datos, garantizando una experiencia de mensajería en tiempo real confiable y escalable. El uso de *Spring Boot* con *JPA/Hibernate* y *PostgreSQL* asegurando la integridad transaccional de los mensajes, mientras que la incorporación de *RabbitMQ* como intermediario asíncrono desacopla la capa de persistencia de la plataforma del flujo de entrega de los mensajes, lo que reduce la latencia y mejorando la tolerancia a fallos de la plataforma. Esta arquitectura implementada permite que cada mensaje enviado por los usuarios sea almacenado correctamente almacenado, procesado y distribuido de forma eficiente, sustentando la robustez y la mantenibilidad del módulo de mensajería dentro de la plataforma de *BeatMap*.

3.8 Configuración e implementación de la capa seguridad del servicio de mensajería.

La seguridad del servicio de mensajería de *BeatMap* se implementa bajo un enfoque de defensa en profundidad, donde cada capa de seguridad aporta un nivel adicional de protección frente a posibles amenazas. Este modelo considera que la seguridad no debe depender de un único componente, sino de una combinación de controles distribuidos que aseguran la autenticación, la autorización, la integridad y la disponibilidad del servicio. Para esto, la plataforma opera bajo un esquema stateless, por medio de *JWT* como mecanismo de autenticación. Esto permite que cada solicitud que viene desde el frontend contenga toda la información necesaria para identificar el usuario, eliminando la necesidad de mantener sesiones persistentes en el servidor.

Los principales pilares que sustentan esta arquitectura son:

- La autenticación: Se implementa mediante la validación de tokens firmados.
- La autorización: Para restringir el acceso a recursos específicos según el rol del usuario.
- Integridad: Asegura que las solicitudes realizadas no sean alteradas ni suplantadas.
- Prevención de abuso: Se integra mediante la limitación de la tasa de conexión y envío de mensajes.

Los principales componentes que materializan estas arquitecturas son el *API Gateway*, encargado de validar los tokens en el perímetro. *Spring Security*, responsable del control de acceso y la protección de *endpoints*. Y, finalmente la implementación de interceptores para las conexiones de *WebSocket* establecidas. Estos son los pilares encargados de mantener la seguridad en la comunicación en tiempo real de la plataforma.

3.9 Flujo de autenticación y validación de tokens con JWT.

La autenticación en *BeatMap* se basa en el uso de *JWT* bajo un modelo *stateless*, eliminando la necesidad de mantener sesiones persistentes en el servidor. El proceso se inicia cuando el cliente envía sus credenciales al endpoint público */api/auth/login*, donde el *backend* valía las credenciales del usuario y genera un token firmado con el algoritmo *HMAC-SHA256*, que contiene información del usuario y sus roles asociados. Este token se devuelve junto a un token de refresco y este se almacena localmente en el lado de cliente por medio de *local storage*. En cada petición realizada posteriormente, el token debe de ser incluido dentro de los *headers HTTP* en los apartados de *Bearer Authentication*. Permitiendo que la *API Gateway* valide la petición a través del servicio *JWTService* que abstrae la lógica de la seguridad. Para esto, existe el método *validateToken* que verifica la firma, expiración y los claims del token. Rechazando los tokens inválidos con un error *HTTP 401* que indica que la solicitud no se pudo completar porque el usuario carece de credenciales de autenticación válidas. En el caso de que las credenciales sean validas, el *Gateway* reenvía la petición al *backend*, donde *Spring Security* establece la identidad del usuario en el contexto de seguridad.

3.10 Flujo de autorización y control de acceso.

La autorización de los *endpoints* del *backend* se gestiona mediante *Spring Security* configurando una serie de reglas que determinan las rutas que requieren de autenticación y las rutas que son públicas. Para esto, las rutas fueron siguen un patrón para determinar fácilmente las rutas que van a requerir de una autorización previa para poder consumir los *Endpoint* asociados a dichas rutas. Las cuales están delimitadas de la siguiente forma.

- *Endpoints* públicos: */api/auth/***
- *Endpoints* protegidos: */api/chat/***, */api/message/***

Estas reglas están definidas dentro del *securityFilterChain* y garantizan que solo los usuarios con tokens válidos puedan acceder a los recursos disponibles del módulo de mensajería. Además, *Spring Security* establece un contexto de seguridad por petición, de manera que los controladores *REST* pueden obtener el usuario autenticado mediante las clases *Principal* o *Authentication*.

3.11 Flujo de seguridad en la capa WebSocket/STOMP.

En el servicio de mensajería en tiempo real, la autenticación y autorización también se aplican a las conexiones *WebSocket* mediante el protocolo *STOMP*. Durante el *handshake* inicial envía un frame de *CONNECT* que incluye el token *JWT* en el Header de *Authorization*. El *backend* intercepta este frame mediante el interceptor *ChatChannelInterceptor*, que valida el token consumiendo el mismo servicio de *JWTService*. Si la validación es exitosa, se asocia el usuario autenticado a la sesión de *WebSocket*. Antes de procesar un mensaje, el método anotado con un *@MessageMapping* realiza una validación adicional para confirmar que el usuario autenticado al chat de destino. En caso contrario, el mensaje enviado se descarta, impidiendo el acceso no autorizado a otros chats.

A continuación, se presenta el diagrama de secuencia que integra los flujos de autenticación, autorización y persistencia del módulo de mensajería de *BeatMap*.

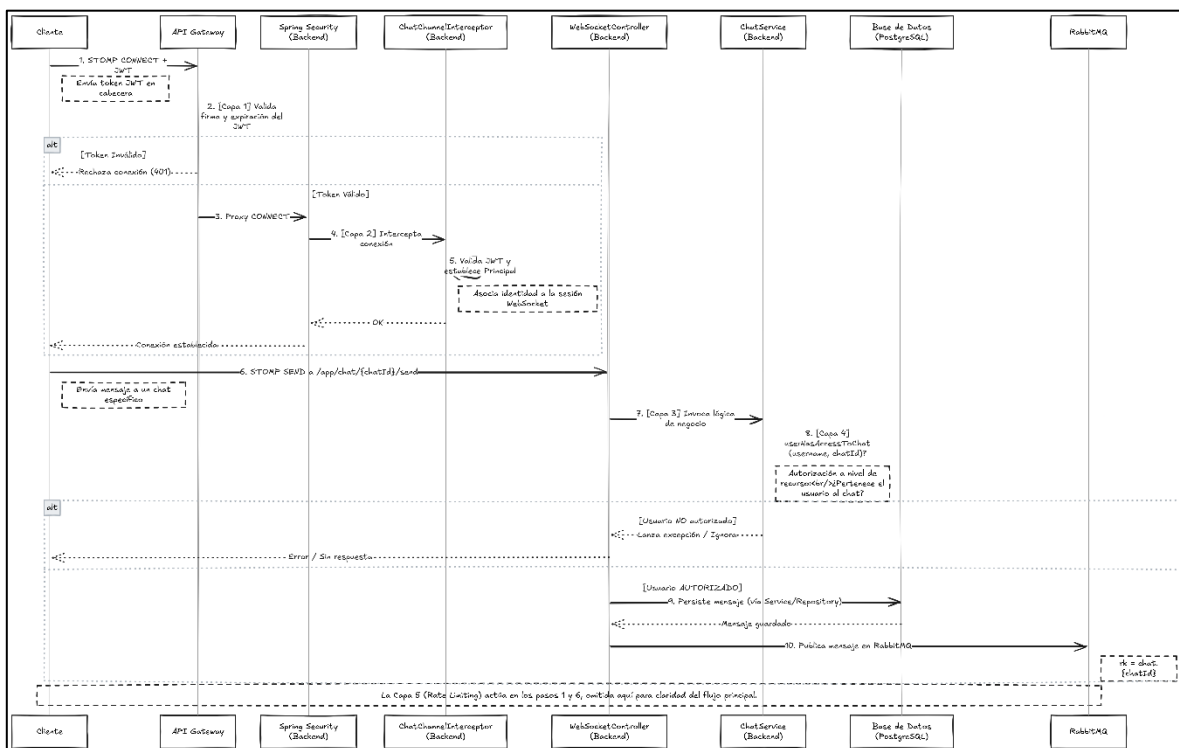


Figura 13: Diagrama de secuencia del funcionamiento de la seguridad del módulo de mensajería
Fuente: Elaboración propia

Este flujo refleja el modelo de defensa en profundidad, combinando validaciones en el perímetro y en la plataforma, junto con el control de acceso dinámico en los canales *WebSocket*. De esta forma, *BeatMap* garantiza la autenticación, autorización, integridad y la trazabilidad de los mensajes, manteniendo la alta disponibilidad del sistema y a la seguridad de los usuarios.

3.12 Plan de pruebas y validaciones del módulo de mensajería.

La validación del servicio de mensajería de BeatMap se estructuró en torno a una estrategia de pruebas orientada a garantizar la calidad, fiabilidad y mantenibilidad de la plataforma. Dado que el módulo de mensajería opera en tiempo real y procesa grandes volúmenes de mensajes de manera recurrente, las pruebas fueron un pilar esencial para asegurar la integridad de los datos y la correcta interacción entre los componentes de la plataforma. Para implementar un plan de pruebas sólido dentro de *BeatMap* se implementó un estándar de pruebas orientado a la pirámide de Cohn. Priorizando una amplia cobertura de pruebas unitarias sobre componentes individuales y complementando con pruebas de integración que validan el flujo completo entre las capas del *backend*. El principal objetivo de las pruebas realizadas fue validar el funcionamiento de la lógica de negocio, la consistencia del almacenamiento de la información en la base de datos y el correcto funcionamiento de los *endpoints REST* que interactúan con el servicio de mensajería.

Para esto, se utilizaron las siguientes librerías y herramientas.

1. *Junit 5: Framework* principal para la ejecución de pruebas de software.
2. *Mockito*: Librería de *mocking* para simular entidades y servicios.
3. *Spring Boot Test*: Entorno de integración que permite levantar el contexto completo de la aplicación para validar la interacción real entre los componentes de la aplicación.
4. *H2 Database* y *MockMVC*: Base de datos en memoria y simulador de peticiones *HTTP* que facilitan la verificación de flujos sin depender de servicios externos.

3.13 Pruebas unitarias: Aislamiento y validación de servicios.

Las pruebas unitarias constituyen la base de la estrategia del plan y estándar de pruebas de software dentro de la plataforma. Las cuales están centradas en validar el comportamiento esperado de los principales servicios sin la necesidad de acceder a la base de datos. Para solucionar el problema de acceder a la base de datos se implementó *Mockito* para crear *mocks* de los repositorios y simular datos de pruebas. Permitiendo centrar las pruebas únicamente en validar el funcionamiento de la lógica de negocio y no desgastar tiempo en realizar configuraciones redundantes e innecesarias.

El servicio *ChatServiceImpl* se centra en la creación, recuperación de salas de chat, así como las validaciones asociadas a los usuarios participantes. Donde los casos de prueba desarrollados fueron:

1. **Creación de un nuevo chat privado:** Se verificó que, al invocar el servicio con dos identificadores de usuario que no comparten un chat previo, el sistema crea correctamente una nueva instancia de la entidad Chat. Se comprobó que a esta nueva entidad se le asignan los dos usuarios como miembros y se persiste adecuadamente, simulando la interacción con el repositorio mediante un mock.

2. **Reutilización de un chat existente:** Para este caso de prueba, se validó el comportamiento idempotente del servicio. Se simuló la existencia de un chat previo entre dos usuarios. Al solicitar nuevamente la creación de un chat entre los mismo dos participantes, se comprobó que el método *getOrCreateDirectChat* retorna la instancia de un chat ya existente, evitando la creación de registros duplicados y optimizando así el uso de los recursos
3. **Validación de casos de errores en la creación de chats:** Se diseñó un test para validar casos bordes, en donde un usuario de prueba que intentó crear un chat consigo mismo. Con esta prueba, se verificó que el servicio arroja una excepción de tipo *IllegalArgumentException*, previniendo la creación inconsistente de chats. Además, se validó que si se intenta crear un chat con un id de un usuario que no existe en la base de datos, lo que provoca que se arroje una excepción de tipo *EntityNotFoundException*, garantizando un robusto manejo de errores.
4. **Verificación de acceso al chat:** Se implementó un caso de prueba para el método *userHasAccessToChat*. Se simuló un escenario donde un usuario es miembro de un chat y otro no. Se confirmó que el método retorna *true* para el miembro y *false* para el otro usuario, garantizando que la lógica de autorización a nivel de servicio funciona como se espera. Además, se probó el método *isUserInChat*, validando que el método retorna *true* si un usuario es miembro del chat y *false* en el caso contrario.
5. **Recuperación de todos los chats de un usuario:** Se probó el método *findAllByUser* que recupera el listado de chats de un usuario. En donde se simuló que el repositorio devuelve un listado de chats asociados a un identificador de usuario y se verificó que el servicio los mapea correctamente a una lista de objetos *ChatDTO*, asegurando que el cliente reciba los datos en el formato esperado.

El servicio *MessageServiceImpl* gestiona todo el ciclo de vida de un mensaje, desde su creación y persistencia hasta la gestión de su estado de lectura y la posterior distribución. Para validar la correcta implementación de estas funcionalidades críticas, se diseñó una serie de pruebas unitarias que cubren los siguientes escenarios:

1. **Creación y persistencia de mensajes:** Se probó el funcionamiento del método *saveMessage*. Verificando que una entidad *MessageDTO* sea correctamente convertida a una entidad *Message*. Asociando el mensaje al chat correspondiente y persistiéndolo en la base de datos. Además, se implementó otro caso de prueba para validar el caso de error que se intenta guardar un mensaje a un chat inexistente, comprobando que el servicio arroja una excepción de tipo *EntityNotFoundException* como se espera.

2. **Recuperación de mensajes:** Se probaron diferentes casos para la obtención de un mensaje por su id, incluyendo el caso de prueba de cuando un mensaje no se encuentra. También se verificó el funcionamiento de la recuperación del historial completo de un chat por el método *getMessagesByChatId*, sumado con la obtención de mensajes por medio de paginación con el método auxiliar *getRecentMessages*, asegurando que el servicio interactúa correctamente con los métodos del repositorio para finalizar el mapeo de los resultados a objetos *DTO*.
3. **Edición de los mensajes:** Se implementaron pruebas para el método *editMessage*, en donde se validó que el contenido de los mensajes enviados se actualiza correctamente.
4. **Eliminación de los mensajes:** Finalmente se implementaron pruebas para validar dos casos de pruebas para el método *deleteMessage*. En donde se comprobó tanto el caso exitoso, en donde los mensajes son borrados exitosamente en la base de datos. Y, por último, se probó un caso de error en el que se intentó eliminar un mensaje con un id inexistente. Confirmando que el servicio *MessageServiceImpl* tiene un robusto manejo de errores.
5. **Gestión del estado de lectura de los mensajes:** Se añadieron pruebas para los métodos de *markAsRead* y *getUnreadMessages*. Para *markAsRead* se validó que, al invocar a dicho método, el usuario receptor de prueba es añadido correctamente al mapa de usuarios lectores del mensaje enviado por el emisor, confirmando la lectura del mensaje. Igualmente se probaron casos de prueba para validar el manejo de errores. Como por ejemplo se intenta marcar un mensaje como leído cuando o el mensaje o el usuario no existe. Asegurando que el manejo de errores del método funciona correctamente. Para el método *getUnreadMessages* se validó que el servicio retorna el historial de mensajes no leídos por un usuario en un chat en específico, así como si el servicio retorne una lista vacía cuando no hay mensajes por leer.

Finalmente, para garantizar la correcta interacción con *RabbitMQ* se implementaron tests unitarios para el servicio *RabbitMQMessageBrokerService*. Estas pruebas se centraron en validar el funcionamiento de la lógica de negocio del servicio sin depender de una instancia real de *RabbitMQ*. Para esto se utilizaron *mocks* para simular el funcionamiento de *RabbitTemplate* y de *RabbitAdmin*. Los tests desarrollados para el servicio fueron:

1. **Envío de mensajes:** Se implementó una prueba para el método *sendMessage*. En donde se validó que, el servicio utilice correctamente el envío de mensajes por medio de *rabbitTemplate.convertAndSend()* para despachar el mensaje a su destinatario correspondiente. Mediante el uso de *ArgumentCaptor*, se verificó que la llamada al *mock* se realizara con los parámetros exactamente esperados.
2. **Suscripción a un tópico válido:** Se probó el método *subscribe*, en donde se simuló una suscripción a un destino con un formato valido por el servicio. En esta prueba se validó que el servicio llame correctamente a *rabbitAdmin.declareQueue* y *rabbitAdmin.declareBinding* para crear la cola y su enlace correspondiente. En donde se validó que los nombres de la cola, la *routing key* y el *Exchange* fueran los correctos, asegurando que la infraestructura de enrutamiento se configura dinámicamente y de forma de adecuada.
3. **Suscripción a un tópico inválido:** Para validar la robustez del servicio, se implementó un caso de prueba en donde se intenta realizar una suscripción a un destino con un formato invalido. En este caso de prueba se validó que el servicio no intenta crear ningún nuevo enlace o una nueva cola. Validando que el manejo de errores evita la creación de recursos inválidos para *RabbitMQ*.

IV. CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN

La validación de esta solución constituye un aspecto crítico dentro del desarrollo del módulo de mensajería de *BeatMap*, ya que permite comprobar de manera objetiva que las funcionalidades implementadas cumplen con los requerimientos técnicos y funcionales definidos durante la fase de diseño. En esta etapa se evalúa no solo la correcta operación del módulo, sino también su robustez, confiabilidad y coherencia frente a distintas casuísticas de uso dentro de la plataforma. El proceso de validación tiene como objetivo principal demostrar que la solución diseñada y planteada es técnicamente viable, escalable y segura. Garantizando la mantenibilidad a largo plazo y la capacidad de integración con el resto de los módulos de la plataforma sin afectar el correcto funcionamiento de esta misma. Para cumplir con este propósito, la validación de la solución se estructuró en dos etapas complementarias. La primera corresponde a la validación técnica de la solución, basado en la implementación de pruebas unitarias e integración por medio de las librerías de *JUnit 5*, *Mockito* y *Spring Boot Tests*, las cuales permiten verificar el correcto funcionamiento de los servicios involucrados y la interacción entre los diferentes componentes del *Backend*.

La segunda etapa aborda las validaciones funcionales, orientadas a los procesos *UAT*. Estas pruebas se realizaron sobre una vista *HTML* simplificada que permitió simular los flujos reales de mensajería y evaluar la respuesta del servidor ante diferentes casuísticas, garantizando que la solución cumple con los requerimientos planteados en el diseño de la solución. En conjunto esta estrategia de validación garantiza una cobertura completa del funcionamiento del módulo de mensajería, abarcando desde la lógica de negocio más específica hasta la validación del comportamiento observable por el lado de usuario final. De este modo, se garantiza que la solución implementada cumple con los requerimientos técnicos de calidad de software y con los requerimientos funcionales definidos para el módulo de mensajería de *BeatMap*.

4.1 Validación Técnica: Pruebas unitarias de los servicios

Las pruebas unitarias constituyen la base de la estrategia de validación técnica de la solución, ya que permiten evaluar el comportamiento de aislado de cada método del módulo de mensajería sin depender de servicios externos o de la base de datos de producción. Para esta implementación se utilizó *JUnit 5* y *Mockito*, para simular las dependencias y repositorios mediante la técnica de *Mocking* y *Stubbing*. Estas pruebas validan tanto casos de pruebas exitosos como casos de pruebas en los que el servicio falla. De manera que se pueda evaluar la mayor cantidad de casos de prueba posible, para evitar malfuncionamientos por casos borde que podrían haber sido validados durante el desarrollo de las pruebas de los servicios.

4.2 Validación del servicio de *MessageServiceImpl*

El servicio *MessageServiceImpl* constituye un componente central dentro del módulo de mensajería de *BeatMap*, ya que este servicio implementa toda la lógica asociada al ciclo de vida de los mensajes enviados entre usuarios. Su correcta operación resulta fundamental para garantizar la consistencia, persistencia, seguridad y comportamiento esperado del servicio de mensajería de la plataforma. Para asegurar su funcionamiento, se desarrolló un conjunto de 15 pruebas unitarias, evaluando tanto escenarios de prueba exitosos como casos en los que el servicio falla. A continuación, se describen los principales casos de prueba validados.

4.3 Creación y persistencia de los mensajes en la base de datos.

Este conjunto de pruebas validó el comportamiento del método *saveMessage*, encargado de persistir un nuevo mensaje en la base de datos. En el caso de pruebas exitoso se comprobó que:

1. El chat asociado exista previamente.
2. El mensaje recibido desde el *frontend* sea convertido correctamente en un *DTO* a una entidad.
3. La entidad resultante sea persistida en la base de datos a través del repositorio.
4. El mensaje almacenado es correctamente convertido a un *DTO* antes de retornar la respuesta.

Adicionalmente, se implementó un caso de error, validando que cuando se intenta guardar un mensaje asociado a un chat inexistente en la base de datos, el servicio arroja la excepción *EntityNotFoundException*, asegurando un manejo correcto del flujo de errores previstos.

4.4 Validación de la recuperación de mensajes almacenados en la base de datos.

Para el método *getMessagesById* se validaron dos escenarios principales del método. Los cuales son:

- Caso de prueba exitoso: El mensaje es recuperado correctamente desde la base de datos y finalmente la entidad es convertida correctamente a un *DTO* y devuelto al cliente.
- Caso de error: Si el mensaje no existe, el servicio arroja una excepción de tipo *EntityNotFoundException*, indicando que la operación no puede completarse.

Estos tests garantizan que el servicio mantiene un comportamiento robusto y coherente frente a búsquedas válidas e inválidas.

4.5 Recuperación del historial de mensajes enviados en un chat entre usuarios.

En este apartado se evaluaron los métodos *getMessagesByChatId* y *getRecentMessages*, ambos esenciales para la carga del historial de conversaciones.

- *getMessagesByChatId*: Valida que el servicio retorne el historial completo de mensajes ordenados cronológicamente, garantizando la consistencia en la visualización de los últimos mensajes enviados.
- *getRecentMessages*: Verifica el uso correcto de la paginación mediante *PageRequest*, permitiendo obtener solo los mensajes más recientes, reduciendo la carga sobre la base de datos.

Ambos casos de pruebas confirmaron que los mensajes son convertidos correctamente a *DTO* y el repositorio es invocado exactamente una vez, evitando problemas de rendimiento como la consulta repetida de datos, evitando el *patrón N+1*.

4.6 Validación de la eliminación de los mensajes enviados.

Esta prueba se encargó de validar el método *deleteMessage*. En este caso, se implementaron tests para validar tanto los casos exitosos como los casos fallidos.

- Validación exitosa de los mensajes: El servicio busca el mensaje en la base de datos por *id* y en caso de existir. Se inicia una transacción para actualizar su estado y este se elimina de forma permanente de la base de datos.
- En caso de que un mensaje no exista: Se valida que el servicio arroje una excepción *EntityNotFoundException* cuando se intenta eliminar un mensaje que no existe.

Estas pruebas aseguran la consistencia del estado de los mensajes y el correcto manejo de errores.

4.7 Validación de la edición de los mensajes.

Las pruebas asociadas al método *editMessage* confirmaron que el mensaje enviado se recupera correctamente desde la base de datos. Una vez recuperado se actualiza su contenido correctamente y posterior a esto se convierte nuevamente a un *DTO* para ser enviado al frontend con el contenido actualizado. Este conjunto de pruebas garantiza que la funcionalidad de edición funciona correctamente y que los cambios se reflejan de manera inmediata por el lado del cliente.

4.8 Validación de la gestión de lectura de los mensajes enviados.

Este fue uno de los aspectos más relevantes a validar, debido a su impacto en la experiencia de mensajería en tiempo real. Para esto los tests cubrieron los siguientes casos de prueba:

- Caso exitoso para el método *markAsRead*: El usuario que lee el mensaje queda registrado correctamente en la colección *readByUsers* de la tabla *Message*.
- Casos de prueba que retorne un error del servicio: Si el mensaje no buscado no existe. Como resultado el servicio retorna una excepción de tipo *EntityNotFoundException*, indicando que el mensaje no existe. Aplicando el mismo caso para cuando se quiere dejar marcado como leído un mensaje para un usuario que no existe, el servicio arroja la misma excepción
- Caso de prueba que valida que no existan duplicados: Se validó que un usuario no pueda marcar como leído un mensaje que ya figura como leído, evitando datos duplicados y manteniendo la integridad de estos.

Además, se probó el método *getUnreadMessages*, validando que el servicio retorne únicamente los mensajes no leídos y no la totalidad de los mensajes enviados por el usuario en el chat. Además de validar que el servicio retorne una lista vacía si no existen mensajes pendientes. Con esto se asegura un manejo correcto del estado de lectura dentro del ciclo de vida del mensaje.

Al finalizar las validaciones, se obtuvo un reporte automático generado por el entorno de pruebas utilizando. Que evidencia la correcta ejecución de los 15 tests correspondientes al servicio *MessageServiceImpl*. La siguiente imagen muestra el resultado obtenido.

MessageServiceImplTest: 15 total, 15 passed		2.17 s
		Collapse Expand
<code>markAsRead_messageNotFound_throws()</code>	passed	1.99 s
<code>getRecentMessages_happyPath()</code>	passed	71 ms
<code>getUnreadMessages_noUnreadMessages_returnsEmptyList()</code>	passed	9 ms
<code>getMessageById_happyPath()</code>	passed	10 ms
<code>markAsRead_userNotFound_throws()</code>	passed	6 ms
<code>validate_markAsRead_happyPath()</code>	passed	18 ms
<code>saveMessage_chatNotFound_throws()</code>	passed	7 ms
<code>markAsRead_alreadyRead_doesNotDuplicate()</code>	passed	5 ms
<code>deleteMessage_notExists_throws()</code>	passed	7 ms
<code>getMessageByChatId_happyPath()</code>	passed	7 ms
<code>saveMessage_happyPath()</code>	passed	17 ms
<code>getMessageById_notFound_throws()</code>	passed	6 ms
<code>editMessage_happyPath()</code>	passed	4 ms
<code>deleteMessage_exists_deletes()</code>	passed	6 ms
<code>getUnreadMessages_happyPath()</code>	passed	13 ms

Figura 14: Reporte de las pruebas realizadas para el MessageServiceImpl.

Fuente: Elaboración propia

Este reporte confirma que la totalidad de las pruebas implementadas se ejecutaron satisfactoriamente, validando el correcto funcionamiento del servicio bajo todos los escenarios definidos.

4.9 Validación del servicio de *ChatServiceImpl*.

El servicio *ChatServiceImpl* es otro de los componentes esenciales dentro del módulo de mensajería de *BeatMap*, ya que implementa toda la lógica asociada a la gestión de chats privados entre los usuarios. Ofreciendo la gestión completa de las conversaciones, desde la creación de chats, validación de los miembros de un chat, control de acceso, recuperación de chats asociados y la consistencia del modelo de membresía a un chat para la tabla *ChatMember*. Dada la importancia del flujo de mensajería en tiempo real, se desarrolló un conjunto de 13 pruebas unitarias orientadas a validar el comportamiento del servicio frente a escenarios exitosos como los casos de error.

4.10 Validación de la creación de chats privados.

Las pruebas implementadas al método *createPrivateChat* se centraron en validar que la lógica de negocio del servicio sea capaz de crear correctamente un chat de tipo *ChatType.PRIVATE* para los dos usuarios. Una vez creado el chat entre los dos usuarios, el *DTO* generado por el servicio contiene el id correspondiente al chat y el listado de usuarios miembros es este. Además de quedar de registrar cada miembro del chat en la tabla de *ChatMember*. Y finalmente, la capa del repositorio será invocada una única vez para registrar estos nuevos registros en la base de datos.

4.11 Idempotencia en la creación de chats privados.

Uno de los principios claves del módulo de mensajería es evitar la duplicidad de chats entre usuarios. Para esto, se implementó una prueba que se encargó de validar el caso en el que, si el método *createPrivateChat* es llamado más de una vez por los mismos usuarios, el servicio debe de retornar el mismo chat creado anteriormente. Evitando casos de chats duplicados en la plataforma. La idempotencia se cumple incluso cuando se invierte el orden de los usuarios miembros del chat. Ya que un chat entre dos usuarios debe ser único.

4.12 Validación del acceso de usuarios a un chat

Una de las pruebas implementadas para el servicio fue para el método *userHasAccessToChat* se centraron en validar distintos escenarios de pruebas que validaron que:

- Cuando un usuario pertenece al chat este tiene un acceso legítimo al mismo.
- Un usuario externo que no tiene permisos para acceder al chat, el servicio retornará un valor *booleano* de *false*, indicando que el usuario externo no tiene permisos para acceder al chat.
- Finalmente, el servicio permite validar el acceso tanto a partir del nombre de usuario como del *id* del chat, otorgando flexibilidad en los mecanismos de validación.

Esta serie de pruebas confirma que el servicio es lo suficientemente robusto para implementar reglas de seguridad para impedir que usuarios externos accedan al chat.

4.13 Validación de la coherencia entre los mensajes del chat y la lectura de estos.

En esta prueba de integración entre el *ChatServiceImpl* y el servicio *MessageServiceImpl*, se verificó correctamente que se creara un chat entre dos usuarios. Permitiendo el envío de mensajes para poder obtener un listado de mensajes no leídos por el receptor y, con el listado de mensajes no leídos se realiza una llamada al servicio *markAsRead*, desapareciendo correctamente de la lista de mensajes no leídos y este mensaje leído queda correctamente en la tabla *Message_read_status*. Esta prueba valida que tanto la lógica de lectura de mensajes como la lógica de envío y obtención de mensajes funcionan correctamente cuando se utilizan en conjunto los servicios de mensajería.

4.14 Integridad de los miembros de un chat.

Las pruebas implementadas para validar la consistencia de la membresía de los usuarios se encargaron de validar que los usuarios en realidad pertenecen al chat recientemente creado, donde ambos usuarios tienen registros asociados a la columna *joined_at* para la tabla *ChatMember* y el repositorio al ser llamado, este retorna correctamente a los dos miembros del chat, sin tener inconsistencias de los datos. Confirmando que el modelo de miembros es construido de manera completa y coherente con lo diseñado en el modelo de datos.

4.15 Manejo de errores para la creación de chats.

El servicio al tener un manejo de errores implementado para garantizar la robustez del servicio ante posibles errores en ambientes productivos, se implementaron pruebas que se encargaron de validar que el servicio arroje correctamente la excepción de tipo *IllegalArgumentException* para cuando un usuario intente crear un chat consigo mismo, rompiendo la lógica de negocio establecida. Además, se implementó otra prueba para validar el caso en el que se intente crear un chat con un usuario inexistente, para esto. El servicio arroja una excepción de tipo *EntityNotFoundException*, gracias al método *validateUserExists*. Estas validaciones implementadas en el servicio confirman que la gestión de peticiones entrantes desde el *frontend* hacia el *backend* está correctamente implementada e impiden que se generen inconsistencias lógicas dentro de la plataforma.

4.16 Validación del método `getOrCreateDirectChat`.

Las pruebas implementadas al método `getOrCreateDirectChat` validaron el caso de prueba en el que si un artista es aceptado en un evento por un organizador de eventos. El servicio consulta en el repositorio si ya existe un chat entre ambos usuarios. En el caso de que ya exista el chat entre estos dos usuarios, el servicio retorna directamente el chat sin la necesidad de crear uno nuevo. En el caso contrario que no exista un chat entre los usuarios, se creará un nuevo chat con sus correspondientes miembros. En estos casos, las pruebas implementadas garantizaron que cuando se crea un chat:

1. El método del servicio `chatService.save()` es invocado correctamente una sola vez.
2. El método del servicio `chatMemberRepository.save()` es llamado correctamente una vez por usuario

Estas pruebas implementadas validan que el método cumple con su propósito de crear un chat cuando no existe o retornar el chat cuando este ya existe.

4.17 Obtención de todos los chats asociados a un usuario.

El método `findAllByUser` fue verificado bajo dos casos de prueba, para probar la mayoría de los casos de prueba posible para evitar casos bordes que puedan provocar inconsistencias en ambientes productivos. Para esto se implementó el caso de prueba en el que el usuario tiene chats registrados con su *id*. Y, el servicio retorna en ese caso retorna un listado con los chats que el usuario posee. Caso contrario en el que usuario no posee chats asociados a su *id*. En ese caso, el servicio es capaz de retornar una lista vacía, reflejando que puede manejar casos en los que existe la información consultada o no. Estas pruebas desarrolladas aseguran la correcta recuperación del historial de conversaciones del usuario.

4.18 Validación del método `isUserInChat`.

Las pruebas verificaron dos escenarios fundamentales para validar la robustez y la seguridad de la plataforma al validar previamente si un usuario es miembro de este chat. Para esto las pruebas desarrolladas validaron los casos en el que un usuario autorizado, miembro del chat quiere acceder a este, el servicio le retorna un valor *booleano* de *true*, indicando que si es miembro de este. Y, finalmente se implementó el caso en el que un usuario no es miembro del chat, el servicio reconoce que no es miembro. Retornando un *booleano* en *false*. El método `chatMemberRepository.existsByChatIdAndUserId()` fue implementado en las pruebas por medio de Mocks y Stubbing para reflejar estos dos estados, validando que el servicio es capaz de agregar capas extras de seguridad al realizar validaciones previas para acceder a otros recursos.

El reporte generado automáticamente por el entorno integrado de desarrollo o IDE. Confirma que la totalidad de las pruebas desarrolladas para el servicio `ChatServiceImpl` fueron ejecutadas satisfactoriamente, validando que el servicio fue correctamente diseñado, implementado y probado todos los escenarios definidos previamente.

ChatServiceTest: 13 total, 13 passed			307 ms
			Collapse Expand
<code>testUserHasAccessToChat()</code>	passed	117 ms	
<code>getOrCreateDirectChat_whenChatExists_returnsExistingChat()</code>	passed	49 ms	
<code>testChatMembershipConsistency()</code>	passed	17 ms	
<code>createPrivateChat_withSameUser_throwsIllegalArgumentException()</code>	passed	18 ms	
<code>validateIfMessageReadStatus()</code>	passed	15 ms	
<code>testCreatePrivateChatIdempotency()</code>	passed	14 ms	
<code>testCreatePrivateChat()</code>	passed	13 ms	
<code>createPrivateChat_withNonExistentUser_throwsEntityNotFoundException()</code>	passed	12 ms	
<code>getOrCreateDirectChat_whenChatDoesNotExist_createsNewChat()</code>	passed	15 ms	
<code>isUserInChat_whenUserIsMember_returnsTrue()</code>	passed	10 ms	
<code>findAllByUser_whenUserHasNoChats_returnsEmptyList()</code>	passed	11 ms	
<code>isUserInChat_whenUserIsNotMember_returnsFalse()</code>	passed	13 ms	
<code>findAllByUser_whenUserHasChats_returnsChatList()</code>	passed	12 ms	

Generated by IntelliJ IDEA on 20/11/25, 0:12

Figura 15: Reporte de las pruebas realizadas para el `ChatServiceImpl`.
Fuente: Elaboración propia.

4.19 Validación del servicio *RabbitMQMessageBrokerService*.

El servicio *RabbitMQMessageBrokerService* es el componente encargado de la comunicación en tiempo real entre los usuarios por medio de *RabbitMQ*, actuando como message *broker* del módulo de mensajería. Su principal función consiste en publicar mensajes hacia los canales correspondientes y gestionar las suscripciones dinámicas a tópicos por medio de colas y *bindings* que se crean en tiempo de ejecución. Dado que *RabbitMQ* opera como un servicio externo al *backend*. La validación fue realizada mediante pruebas unitarias que utilizan *Mockito* para simular el comportamiento del *RabbitTemplate* y de *RabbitAdmin*. Esto permite verificar la interacción con el *broker* sin requerir una instancia real de *RabbitMQ* en permanente ejecución, lo cual es crítico para asegurar el aislamiento, reproducibilidad y velocidad en las pruebas. Dado esto, las pruebas se centraron en validar cuatro aspectos fundamentales del servicio.

1. Validar el envío correcto de los mensajes al exchange.
2. Validar la creación dinámica de colas y de bindings.
3. Validar el correcto manejo de destinos inválidos.
4. Validar la propagación de excepciones de *RabbitMQ*.

A continuación, se detalla cada caso de prueba validado dentro de los tests unitarios.

4.20 Validación del método *sendMessage* para el envío de mensajes.

El método *sendMessage* es responsable de enviar un mensaje hacia el exchange definido en la aplicación, utilizando una *routing key* derivada del destino del canal asociado al chat. La prueba *sendMessage_sendsMessageToCorrectDestination* validó que:

1. El mensaje es enviado al *exchange* correcto (*chat.exchange*).
2. La *routing key* utilizada coincide exactamente con el destino solicitado (*chat.123*).
3. El objeto enviado al *broker* es el mismo *MessageDTO* recibido por el servicio.

Para esta validación se utilizó *ArgumentCaptor*, lo cual permitió capturar los argumentos reales enviados por *convertAndSend()* y compararlos con los valores esperados. Esta prueba confirma que el servicio respeta el contrato de publicación y genera el enrutamiento correcto para el sistema de mensajería en tiempo real.

4.21 Validación de la suscripción a canales y creación de colas dinámicas.

El método *subscribe* permite que diferentes instancias del sistema como los *listeners* de *WebSocket* se suscriban dinámicamente a un tópico en específico. Este método es esencial para el funcionamiento del chat en tiempo real. Dado que *RabbitMQ* requiere crear una cola exclusiva para cada destino de suscripción, como también necesita crear un *binding* que conecta la cola con el *exchange* por medio de la *routing key* correspondiente. Para esto se implementó una prueba que validara el caso exitoso del método, en el cual se validó que se realizara una suscripción válida a un tópico válido con el nombre */topic/chat.123*. Teniendo en cuenta la suscripción válida, el servicio creará una cola válida con el nombre *queue.chat.123* y su correspondiente *routing key* correspondiente. Realizando la asociación correspondiente al *exchange* para poder derivar la cola y su *key* correctamente al destino del mensaje enviado por medio de *STOMP* para entregar asíncronamente el mensaje.

Además, el método *subscribe* incorpora validaciones internas para evitar la creación de colas o de *bindings* con formatos incorrectos. Lo cual es importante para prevenir basura en el *broker* y mantener la alta disponibilidad e integridad de la plataforma. La prueba válida los casos mencionados anteriormente se centraron en verificar que, si el destino no cumple con el formato esperado por el servicio, no se creará ninguna cola ni *binding*. Esto evita la creación de recursos innecesarios o inconsistentes en *RabbitMQ*, demostrando la robustez del servicio ante casos de entradas inválidas.

4.22 Validación del manejo de errores del servicio

La estabilidad del módulo de mensajería requiere que los errores provenientes del servicio de *RabbitMQ* sean detectados de manera temprana y gestionados adecuadamente en ambientes productivos. La prueba implementada validó que cuando el servicio de *RabbitMQ* lanza una excepción de tipo *AmqpException*. Este propagará la excepción sin suprimirla ni ignorarla. Permitiendo que la lógica de negocios pueda manejar el error adecuadamente. Este manejo de errores permite mejorar la trazabilidad de los errores, manteniendo una arquitectura alineada con el principio de *fail fast*.

En conjunto, las pruebas implementadas validan que el servicio envía mensajes correctamente al *exchange* de *RabbitMQ*. Junto con la correcta suscripción a los chats que generan colas y *bindings* válidos, además de invalidar los destinos que el servicio determine inválidos para descartarlos por seguridad. Con esto se garantiza que el módulo de mensajería cuenta con un servicio de comunicación en tiempo real, estable, seguro y coherente con los requerimientos de la plataforma.

RabbitMQMessageBrokerServiceTest: 4 total, 4 passed		2.00 s
		Collapse Expand
subscribe_withInvalidDestination_doesNotCreateQueue()	passed	1.96 s
subscribe_withValidDestination_createsQueueAndBinding()	passed	17 ms
sendMessage_whenRabbitThrowsException_propagatesException()	passed	22 ms
sendMessage_sendsMessageToCorrectDestination()	passed	4 ms
Generated by IntelliJ IDEA on 18/11/25, 23:48		

Figura 16: Reporte de las pruebas realizadas para el servicio que gestiona la integración con RabbitMQ.
Fuente: Elaboración propia.

4.23 Validaciones funcionales mediante interfaz web siguiendo procesos UAT.

Con el objetivo de validar funcionalmente el módulo de mensajería desde la perspectiva del usuario final, se desarrollaron cuatro interfaces web independientes implementadas con *HTML*, *CSS* y *JavaScript*. Estas vistas permiten interactuar directamente con los endpoints *REST* y *WebSocket* del *backend*, evaluando el comportamiento completo de la plataforma sin la necesidad de depender aún de un *frontend* oficial o de una aplicación móvil. Las vistas cumplen con el propósito esencial de validar la funcionalidad real del *backend* bajo casos de uso típicos, tales como el envío de mensajes, recuperar el historial de mensajes u obtener los mensajes no leídos por el usuario remitente. Con la finalidad de validar el funcionamiento de la comunicación en tiempo real. Para la comunicación mediante el protocolo *WebSocket* y *STOMP* se utilizó la librería *SockJS* y *stomp.js*, lo que permitió emular el comportamiento de un cliente en producción. Cada vista fue diseñada como un módulo de validación aislado, permitiendo evaluar por separado un subconjunto en específico de las funcionalidades del módulo de mensajería. A continuación, se describe la función de cada una de las vistas implementadas para analizar los resultados obtenidos.

4.24 Validación del flujo *WebSocket* para el envío en tiempo real de mensajes.

La primera interfaz desarrollada corresponde al validar más completo dentro de las cuatro vistas diseñadas para la validación funcional del módulo de mensajería. Esta vista permite simular de manera íntegra el flujo de mensajería en tiempo del *BeatMap*, verificando tanto el envío como la recepción de mensajes a través del canal *WebSocket*. El propósito central de esta interfaz es validar empíricamente que la comunicación bidireccional entre cliente y servidor *WebSocket* y *RabbitMQ* como *broker* intermediario que opera de forma correcta, consistente y sin pérdida de información. Gracias a la implementación de *SockJS* y *de stomp.js*, la vista replica fielmente el comportamiento que tendría un cliente real de la plataforma que utilice el módulo de mensajería. Permitiendo validar tanto la estructura del mensaje enviado, las rutas *WebSocket* utilizadas, el enrutamiento hacia el *exchange* y la correcta notificación hacia los suscriptores del chat. La vista se organiza por medio de módulos que representan cada etapa del flujo en tiempo real del envío de mensajes.

1. Conexión al *Websocket*: Permite establecer una sesión *STOMP* con el servidor e inspeccionar los logs generados durante la conexión con el servidor, lo que facilita validar que el *handshake WebSocket* se realizó correctamente.
2. Suscripción a un canal con la ruta *topic/chat/{chatId}*: Esta sección permite validar que el cliente queda correctamente suscrito al tópico correspondiente, quedando preparado para recibir mensajes publicados en ese chat.
3. Envío de mensajes en tiempo real: Una vez suscrito el cliente al tópico, la interfaz permite enviar un mensaje con un objeto *MessageDTO* completo al *endpoint STOMP /app/chat/{chatId}/send*, validando que el *backend* reciba y procese correctamente el mensaje enviado.
4. Recepción visual de los mensajes entrantes: Finalmente, mediante la sección de *logs*, se muestra cada mensaje que llega al canal suscrito, permitiendo comparar su estructura con el mensaje originalmente enviado y confirmando que la comunicación en tiempo real funciona correctamente.

Con la finalidad de evidenciar visualmente que el módulo de mensajería opera correctamente de principio a fin, se realizó una prueba enviando un mensaje real mediante la vista *websocket-test*. Donde a continuación se presentan los resultados obtenidos durante la prueba, incluyendo el mensaje enviado por el cliente, los logs generados por la vista y el registro final almacenado en la base de datos. Durante el proceso de validación, se utilizó la vista en un entorno de pruebas local, ingresando un *chatId* y el id del remitente que va a recibir el mensaje con un contenido genérico, el cual es “¡Hola! Esta es una prueba funcional para validar el funcionamiento de la comunicación en tiempo real del servicio de mensajería de *BeatMap*.”

BeatMap WebSocket Test

Conexión WebSocket

URL del servidor:

Conectado

Suscripción a Topic

ID del Chat:

Envío de Mensajes

ID del Chat:

ID del Remitente:

Mensaje:

Figura 17: Panel de administración para la validación del funcionamiento del envío de mensajes
Fuente: Elaboración propia.

Los logs de la vista ofrecen una trazabilidad del proceso completo, desde la serialización de los mensajes, el envío del mensaje hacia el endpoint *WebSocket*, entregando la confirmación del envío y la recepción de los mensajes.

Logs

```

22:17:48 - Enviando mensaje a /app/chat/4/send: {"chatId":4,"content":"Hola! Esta es una prueba funcional para validar el funcionamiento de la comunicación en tiempo real del servicio de mensajería de BeatMap","sender":{"id":1},"type":"TEXT","sendedAt":"2025-11-22T01:17:48.064Z"}
22:17:48 - >>> SEND destination:/app/chat/4/send content-length:234
{"chatId":4,"content":"Hola! Esta es una prueba funcional para validar el funcionamiento de la comunicación en tiempo real del servicio de mensajería de BeatMap","sender":{"id":1},"type":"TEXT","sendedAt":"2025-11-22T01:17:48.064Z"}
22:17:48 - Mensaje enviado
22:17:52 - <<< PONG
  
```

Figura 18: Logs de la interacción entre el backend y el servicio de RabbitMQ para el envío de mensajes.
Fuente: Elaboración propia.

Estos logs demuestran que el mensaje fue construido correctamente con el contenido escrito anteriormente. Una vez enviado el mensaje correctamente al *endpoint* */app/chat/4/send* con el mensaje enviado, el *broker* lo aceptó correcta y finalmente, la conexión por medio de *WebSocket* se mantuvo activa correctamente durante todo el proceso.

Posterior al envío del mensaje por medio del *endpoint*, se validó en la tabla *messages* de la base de datos para confirmar que el mensaje ha sido correctamente persistido en la base de datos, confirmando que durante todo el proceso de envío de mensajes no existió ninguna alteración ni pérdida de datos y la lógica de persistencia del módulo funcionó correctamente.



id	message_content	sent_at	type	chat_id	sender_id
1	Hola como estás?	2025-11-17 02:38:13.873000	TEXT	3	2
2	Bien y tu?	2025-11-17 02:44:40.228000	TEXT	3	1
3	wena javier choque	2025-11-17 23:18:38.077000	TEXT	3	2
4	Hola como estás?	2025-11-21 01:57:06.298000	TEXT	3	2
5	Hola como estás? esto es una prueba del estado de lectura	2025-11-21 02:43:18.626000	TEXT	3	1
6	Hola! Esta es una prueba funcional para validar el funcionamiento de la com...	2025-11-22 01:17:48.139000	TEXT	4	1

Figura 19: Registros de la tabla Messages de la base de datos con el mensaje de prueba enviado.

Fuente: Elaboración propia.

La prueba realizada mediante la vista de administración de *WebSocket* permitió validar el flujo completo de la comunicación en tiempo real del módulo de mensajería de *BeatMap*. Validando el flujo completo de conexión, suscripción, envío en tiempo real de los mensajes con su correspondiente recepción y persistencia de los mensajes en la base de datos. Las evidencias obtenidas ayudaron a confirmar que el *backend*:

1. Procesa correctamente los mensajes enviados por los usuarios.
2. El mensaje enviado se transmite sin pérdida ni modificaciones de terceros hacia el broker de *RabbitMQ*.
3. El cliente suscrito al canal recibe la notificación en tiempo real del mensaje enviado.
4. Y, finalmente la base de datos registra correctamente la información transmitida.

Con estas pruebas se demuestra que el servicio de mensajería cumple con los requerimientos funcionales establecidos y está preparado para la futura integración con el frontend oficial de *BeatMap*.

4.25 Validación del historial de mensajes del chat.

Para validar el funcionamiento de los *endpoints* responsables de recuperar el historial de mensajes asociados a un chat. Para esto se desarrolló una vista denominada *Chat History Test* que permite consumir todos los servicios *REST* del módulo de mensajería para visualizar la información tal como la recibiría un cliente real, pero en formato *JSON* para facilitar la integración con el *frontend*, proporcionando así una validación funcional completa por medio de pruebas de tipo *UAT*. Con esto, el propósito general de esta vista es validar que:

1. El *backend* retorna correctamente el historial completo de mensajes de un chat.
2. La paginación implementada en los endpoints funciona correctamente.
3. El servicio retorna los mensajes de manera ordenada y coincide con los registros almacenados en la base de datos.

La vista se estructura en tres módulos principales, los cuales se centrarán en validar la obtención del historial completo de los mensajes enviados por los usuarios en el chat. Se validará la obtención del historial de mensajes paginado para optimizar el rendimiento de la plataforma frente a un enorme volumen de mensajes y finalmente, se validará la visualización formateada correctamente de los mensajes recuperados. El primer módulo permite ingresar un *id* de un chat para consumir el *endpoint* con la ruta */chat/{chatId}/messages*. La vista muestra el historial completo de mensajes enviados a un chat. Durante el proceso de validación, se consultó al historial completo de mensajes del chat con un *id* 4. Y, una vez consumido el *endpoint* el servicio retornó correctamente hacia la vista todos los mensajes registrados en la base de datos, mostrando información relevante como el *id* del remitente, el contenido del mensaje enviado y la fecha en la que se envió el mensaje al remitente. Esta prueba confirma que el *endpoint* funciona correctamente y retorna la estructura definida en los *DTOs* del *backend*.

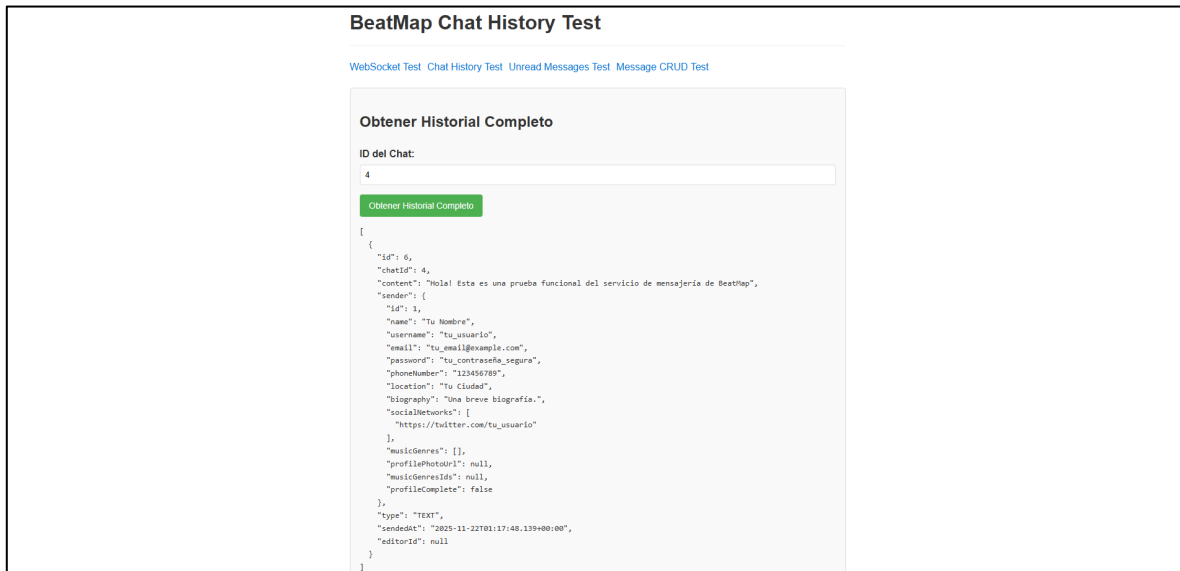


Figura 20: Vista de administración para validar las funcionalidades a la obtención del historial de mensajes.

Fuente: Elaboración propia.

4.26 Validación del historial de mensajes del chat paginado.

El segundo módulo de la vista permite evaluar el endpoint `/message/{chatId}/messages?page=X&size=Y&sortBy=Z&direction=W`. Este endpoint es fundamental para validar el rendimiento de la plataforma en chats con grandes volúmenes de mensajes. La vista permite configurar el número de página a buscar, el tamaño de esta, el formato de ordenamiento (por *id* o por la fecha de envío) y el ordenamiento. La vista al igual genera dinámicamente los botones de la paginación, lo que facilita la navegación de las páginas de los registros, manteniendo una carga incremental de los mensajes sin afectar al rendimiento del módulo de mensajería.

Obtener Historial Paginado

ID del Chat: 4

Página: 0

Tamaño de página: 10

Ordenar por: Fecha de envío

Dirección: Descendente (más reciente primero)

Obtener Historial Paginado

```
{
  "content": [
    {
      "id": 6,
      "chatId": 4,
      "content": "¡Hola! Esta es una prueba funcional del servicio de mensajería de Beasmap",
      "sender": {
        "id": 1,
        "name": "Tu Nombre",
        "avatar": "https://i.pravatar.cc/300"
      }
    }
  ]
}
```

Anterior 1 Siguiente

Figura 21: Vista de administración para validar la obtención del historial de mensajes paginados.

Fuente: Elaboración propia.

Durante las pruebas realizadas a este servicio se validó que los mensajes retornados correspondían únicamente al rango solicitado, el ordenamiento de los mensajes obtenidos en el orden solicitado en el formulario de la vista y el total de mensajes retornados a la vista coincide con los datos almacenados en la base de datos. Esto demuestra que la lógica de paginación del servicio opera de manera correcta, estable y coherente con lo almacenado en la base de datos.

id	edited_at	previous_content	previous_type	editor_id	message_id
1	2025-11-22 03:32:02.476000	¡Hola! Esta es una prueba funcional para validar el funcionamiento de la comu...	TEXT	1	6

Figura 21: Registros de la tabla Message_history con el total de mensajes enviados.

Fuente: Elaboración propia.

Los mensajes retornados por la vista fueron comparados con los registros reales de la tabla *Messages* en la base de datos confirmando que los mensajes, recuperados por la *API* corresponden a los mensajes almacenados en la base de datos. Finalmente, la validación mediante la vista *chat History test* permitieron concluir que el servicio de recuperación funciona correctamente y los mensajes enviados se obtienen de forma íntegra, ordenada y sin inconsistencias. Con esto, se puede evidenciar empíricamente que la obtención del historial de mensajes cumple con los requerimientos establecidos y está listo para su integración con el *frontend*.

4.27 Validación de mensajes no leídos y estados de lectura de los mensajes.

Para validar la funcionalidad asociada al estado de lectura de los mensajes, se desarrolló una vista denominada como *Unread Message Test*, la cual permite probar dos operaciones fundamentales del módulo de mensajería.

1. Obtención de mensajes no leídos por un usuario por medio de un endpoint *REST*.
2. Realizar el marcado de mensajes como leídos, por medio de un evento con *WebSocket* directo al *backend*.

Esta vista sirve para simular el flujo completo de lectura de un cliente real, lo que permite verificar visual y empíricamente que el servicio identifica correctamente que mensajes no han sido leídos por el receptor de los mensajes.

4.28 Validación de la obtención de mensajes no leídos.

El primer módulo de la vista permite ingresar el nombre de usuario y el *id* del chat para poder consumir el endpoint */message/unread/{chatId}*. Este endpoint forma parte del servicio *MessageServiceImpl* y su objetivo es filtrar los mensajes cuyo estado de lectura no incluye al usuario que realizó la consulta. Y, una vez realizada la petición la vista muestra

1. El *JSON* con los datos asociados a los mensajes sin leer por parte del receptor de los mensajes.
2. Construye una representación visual por medio de tarjetas del listado de mensajes no leídos con el *id* del remitente, el mensaje que envió, la hora y fecha en la que se envió el mensaje y finalmente una etiqueta que indica que el mensaje no ha sido leído.

Esto permite validar visualmente que los mensajes recuperados efectivamente no han sido leídos y su estructura coincide con los mensajes almacenados en la base de datos.

Figura 22: Vista de administración para validar el estado de lectura de los mensajes enviados.
Fuente: Elaboración propia.

4.29 Marcar un mensaje como leído por medio de una conexión con *WebSocket*.

El segundo de la vista corresponde a la validación de la operación de lectura de mensajes por medio de comunicación en tiempo real por medio de *WebSocket*. Para probar este *endpoint*. El usuario debe ingresar el *id* del chat, el *id* del mensaje que va a ser leído y el *id* del usuario que está leyendo el mensaje. Y, Al presionar el botón de “Marcar como leído”, la vista internamente realiza una conexión persistente por medio de *WebSocket* que envía un evento al *endpoint* `/app/chat/{chatId}/read/{messageId}` permite validar que la notificación llega correctamente al *backend*, siendo procesada sin errores y queda registrado el mensaje en la columna *readByUsers* de la tabla *Message*.

Figura 23: Formulario para marcar un mensaje como leído.
Fuente: Elaboración propia.

Y, finalmente comparamos la acción realizada en el lado del cliente con lo registrado en la base de datos, podemos confirmar que el mensaje marcado como leído queda registrado

correctamente en la base de datos en la tabla *Message_read_status*. Confirmando que las funcionalidades implementadas para el manejo del estado de lectura de los mensajes funcionan correctamente y es una funcionalidad clave para mejorar la experiencia de usuario para los usuarios finales.

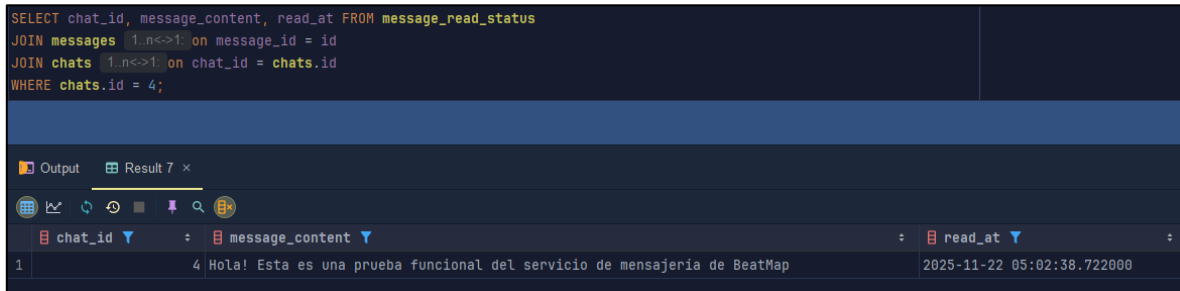


Figura 24: Registros en la base de datos para la tabla *Message_read_status* para validar el estado de lectura de los mensajes enviados por los usuarios.

Fuente: Elaboración propia

4.4.1 Validación de las operaciones *CRUD* del servicio *MessageServiceImpl*.

Con el objetivo de validar el correcto funcionamiento de los *endpoints* responsables del ciclo de vida completo de los mensajes, se desarrolló una vista denominada *Message CRUD Test*, la cual permite ejecutar las operaciones de creación, edición y eliminación de mensajes mediante una interfaz web sencilla. Esta vista funciona como un cliente experimental que permite evaluar las funcionalidades del *backend* sin depender aún del *frontend* oficial de *BeatMap* que esté listo para validar estas funcionalidades. La estructura modular de la vista facilita validar de manera aislada cada una de las operaciones del servicio, permitiendo evidenciar visualmente las respuestas del *backend*, junto con los códigos de estado *HTTP*, el contenido enviado y los resultados almacenados en la base de datos. A continuación, se presenta el desglose de cada módulo y las validaciones realizadas.

4.4.2 Validación de la creación de mensajes.

El primer módulo de la vista corresponde a la operación de creación de mensajes. Este panel permite ingresar el *id* del chat, el *id* del remitente y el contenido del mensaje. Al presionar el botón de crear mensaje, la vista genera el objeto *MessageDTO* y lo envía al *backend* utilizando *WebSocket* si la conexión está activa. Durante la validación se verifica que, una vez enviada la solicitud, la vista muestre:

- El código de estado de respuesta *HTTP* 200.
- El contenido enviado hacia el *backend*.
- La confirmación visual de que el mensaje fue procesado exitosamente.

Esta validación permitió confirmar que el *backend* recibe correctamente el *DTO* enviado desde el cliente, la estructura del mensaje creado coincide con la definida en el servicio sin existir errores de validación o de autenticación y finalmente el mensaje creado queda correctamente persistido en la base de datos en la tabla *Message*.

BeatMap Message CRUD Test

[WebSocket Test](#) [Chat History Test](#) [Unread Messages Test](#) [Message CRUD Test](#)

Crear Mensaje (POST)

ID del Chat:
4

ID del Remitente:
2

Contenido:
Esta es una validación del envío de mensajes del CRUD de mensajes

Crear Mensaje

Mensaje enviado exitosamente vía WebSocket.

```
Payload enviado: {
  "chatId": 4,
  "senderId": 2,
  "content": "Esta es una validación del envío de mensajes del CRUD de mensajes",
  "sender": {
    "id": 2
  },
  "type": "TEXT"
}
```

Editar Mensaje (PUT)

Figura 25: Vista de validación de los *endpoints* del servicio de mensajería.

Fuente: Elaboración propia

Y, para asegurar la consistencia, se comparó el mensaje enviado desde la vista con el registro almacenado en la base de datos, confirmando que el mensaje persistido corresponde al contenido enviado desde el cliente.

id	message_content	sented_at	type	chat_id	sender_id
1	Hola como estás?	2025-11-17 02:38:13.873000	TEXT	3	2
2	Bien y tu?	2025-11-17 02:44:40.228000	TEXT	3	1
3	wena javier choque	2025-11-17 23:18:38.077000	TEXT	3	2
4	Hola como estás? esto es una prueba del estado de lectura	2025-11-21 02:43:18.626000	TEXT	3	1
5	Hola! Esta es una prueba funcional del servicio de mensajería de BeatMap	2025-11-22 01:17:48.139000	TEXT	4	1
6	Esta es una validación del envío de mensajes del CRUD de mensajes	2025-11-23 01:57:10.698000	TEXT	4	2

Figura 25: Registros de la base de datos de la tabla *Messages* para validar la persistencia de los mensajes.

Fuente: Elaboración propia

4.4.3 Validación de la edición de mensajes.

El segundo módulo de la vista corresponde a la operación de editar mensajes, permitiendo validar que el servicio actualiza correctamente el contenido de los mensajes existentes. Para esto el usuario debe de ingresar en el formulario el id del mensaje a editar, el id del usuario editor del mensaje y el nuevo contenido del mensaje. Una vez realizada la petición, el backend procesa la actualización de la información y retorna el mensaje modificado, donde la vista muestra:

- El código de respuesta *HTTP* con un código 200.
- El mensaje actualizado retornado por el servicio.
- El contenido enviado a la *API* para la edición del mensaje.

Esta validación permitió comprobar que el backend reconoce correctamente el mensaje que debe ser modificado. La lógica de edición de mensajes respeta las validaciones del servicio. Además, que el campo *editedAt* de la tabla *Message_history* es generado correctamente por el *backend*, registrándolo en la base de datos.

```

Mensaje actualizado: {
  "id": 7,
  "chatId": 4,
  "content": "Esta es una validación de la edición de mensajes del CRUD de mensajes",
  "sender": {
    "id": 2,
    "name": "Tu Nombre",
  }
}

```

Figura 26: Vista de validación de la edición de los mensajes enviados.

Fuente: elaboración propia

La consistencia de la actualización de los mensajes editados se confirmó revisando directamente en la base de datos que el mensaje editado reflejara los cambios realizados a través de la vista.

	id	message_content	sent_at	type	chat_id	sender_id
1	1	Hola como estás?	2025-11-17 02:38:13.873000	TEXT	3	2
2	2	Bien y tu?	2025-11-17 02:44:40.228000	TEXT	3	1
3	3	wena javier choque	2025-11-17 23:18:38.077000	TEXT	3	2
4	5	Hola como estás? esto es una prueba del estado de lectura	2025-11-21 02:43:18.626000	TEXT	3	1
5	6	Hola! Esta es una prueba funcional del servicio de mensajería de BeatMap	2025-11-22 01:17:48.139000	TEXT	4	1
6	7	Esta es una validación de la edición de mensajes del CRUD de mensajes	2025-11-23 01:57:10.098000	TEXT	4	2

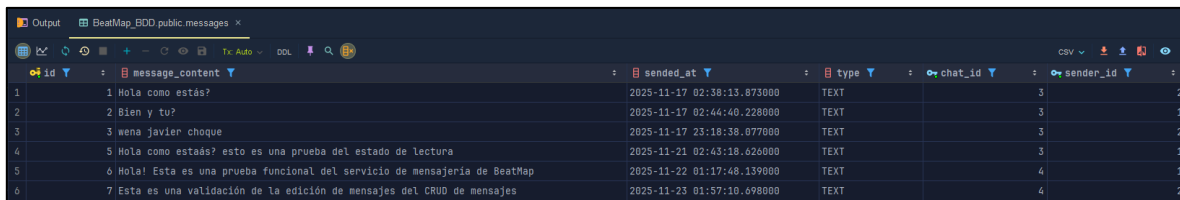
Figura 27: Registros de la base de datos de la tabla Messages para validar la persistencia de los mensajes editados por el usuario.

Fuente: Elaboración propia.

4.4.4 Validación de la eliminación de mensajes.

El tercer módulo de la vista permite validar la operación de eliminación de mensajes mediante el *endpoint* `/message/{messageId}/delete`. El objetivo principal de esta prueba es verificar que el servicio *MessageServiceImpl* elimine de forma correcta y definitivamente un mensaje registrado en la base de datos, garantizando la integridad del historial de conversación y un comportamiento coherente desde la perspectiva del usuario final. Para realizar esta validación, se procedió a seguir un flujo completo y totalmente observable mediante la interfaz web y los registros persistidos en la base de datos.

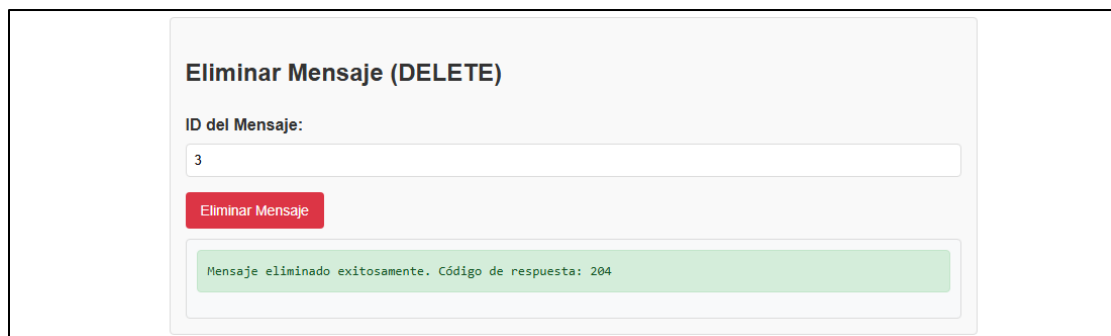
Previo a la eliminación de un mensaje de pruebas, se inspeccionó la tabla *Messages* en la base de datos para seleccionar un mensaje persistente a eliminar. En este caso se seleccionó el mensaje con el *id* 3, que contiene el mensaje “*wena Javier choque*” asociado al chat 3. Esto permite establecer un punto de control inicial para validar el cambio que debería reflejarse en la base de datos después de consumir el *endpoint*.



id	message_content	send_at	type	chat_id	sender_id
1	Hola como estás?	2025-11-17 02:38:13.873000	TEXT	3	2
2	Bien y tu?	2025-11-17 02:44:40.228000	TEXT	3	1
3	wena javier choque	2025-11-17 23:18:38.077000	TEXT	3	2
4	Hola como estás? esto es una prueba del estado de lectura	2025-11-21 02:43:18.626000	TEXT	3	1
5	Hola! Esta es una prueba funcional del servicio de mensajería de BeatMap	2025-11-22 01:17:48.139000	TEXT	4	1
6	Esta es una validación de la edición de mensajes del CRUD de mensajes	2025-11-23 01:57:10.698000	TEXT	4	2

Figura 28: Registros iniciales en la tabla Messages
Fuente: Elaboración propia

Posteriormente, desde la vista *Message CRUD Test*, se ingresó el id del mensaje que se desea eliminar. Al realizar la petición al *endpoint* `/message/3/delete` del *backend* y se muestra la respuesta *HTTP* entregada por el servicio. En este caso, el *backend* retornó correctamente un código de estado *HTTP* 204, indicando que la operación fue ejecutada con éxito y un mensaje confirmando que el mensaje se ha eliminado correctamente. Esta respuesta confirma que el *backend* aceptó y procesó la petición sin errores.



Eliminar Mensaje (DELETE)

ID del Mensaje:

Eliminar Mensaje

Mensaje eliminado exitosamente. Código de respuesta: 204

Figura 29: Confirmación visual del borrado del mensaje mediante la vista
Fuente: Elaboración propia

Finalmente, se volvió a consultar a la tabla *Messages* en la base de datos, verificando que el mensaje con el *id* 3 ya no aparece dentro del listado de mensajes persistidos en la base de datos. Con este flujo completo se validó empíricamente que la operación *DELETE* del módulo de mensajería funciona correctamente desde el punto de vista técnico y funciona, cumpliendo con los requerimientos establecidos para el manejo del ciclo de vida de los mensajes dentro de *BeatMap*.



id	message_content	sent_at	type	chat_id	sender_id
1	Hola como estás?	2025-11-17 02:38:13.873000	TEXT	3	2
2	Bien y tu?	2025-11-17 02:44:40.228000	TEXT	3	1
3	Hola como estás? esto es una prueba del estado de lectura	2025-11-21 02:43:18.626000	TEXT	3	1
4	Hola! Esta es una prueba funcional del servicio de mensajería de BeatMap	2025-11-22 01:17:48.139000	TEXT	4	1
5	Esta es una validación de la edición de mensajes del CRUD de mensajes	2025-11-23 01:57:10.698000	TEXT	4	2

Figura 30: Registros de la tabla *Messages* después de la eliminación del mensaje con *id* 3.

Fuente: elaboración propia.

Las validaciones funcionales realizadas mediante las cuatro vistas *HTML* desarrolladas permitieron confirmar de manera simple, completa, visual y empírica el comportamiento integral del módulo de mensajería. Cada vista fue diseñada para un ambiente de pruebas *UAT*, capaz de simular casos de uso reales que un usuario final ejecutaría dentro de la plataforma. A lo largo de estas validaciones se comprobó que:

1. El flujo de *WebSocket* completo funciona correctamente, incluyendo la conexión, suscripción, envío y recepción de mensajes en tiempo real mediante *STOMP* y *RabbitMQ*.
2. La persistencia de mensajes es sólida, reflejando que todo mensaje enviado queda almacenado en la base de datos sin pérdidas ni inconsistencias.
3. El historial de mensajes y la paginación funcionan adecuadamente, retornando datos coherentes, ordenados y equivalentes a los registros reales de la base de datos.
4. El manejo de estado de lectura opera de forma precisa, identificando mensajes no leídos y registrando la lectura en tiempo real mediante *WebSocket*.
5. Las operaciones *CRUD* del servicio *MessageServiceImpl* están correctamente implementados, permitiendo crear, editar y eliminar mensajes sin afectar la integridad del módulo.

En conjunto, estas validaciones permiten concluir que el módulo de mensajería desarrollado es estable, coherente con el diseño técnico y está preparado para operar en el entorno real de producción de *BeatMap*.

CAPÍTULO 5: CONCLUSIONES

El desarrollo del módulo de mensajería de *BeatMap* permitió resolver un desafío fundamental: habilitar un servicio de comunicación robusto, seguro y adaptable a los flujos de interacción propios de la *BeatMap*. Este proyecto no solo abordó la implementación técnica del servicio, sino que también su validación exhaustiva mediante pruebas unitarias, de integración y pruebas funcionales mediante procesos *UAT*, logrando un entendimiento y aprendizaje significativo en el diseño e implementación de una arquitectura *backend* robusta, mantenible y escalable orientado a la comunicación en tiempo real. Las conclusiones que siguen sintetizan los resultados obtenidos, los aportes generados, las limitaciones del trabajo actual y las posibilidades de mejoras a futuro.

5.1 Cumplimiento de objetivos y validez de la solución

La solución desarrollada cumplió plenamente con los objetivos definidos en un principio. Dentro de los logros más importantes destacan:

1. Diseño e implementación de un servicio de mensajería en tiempo real utilizando *WebSocket* y *STOMP* capaz de transmitir mensajes sin pérdidas de información y con una estabilidad operacional.
2. Implementar una arquitectura escalable y desacoplada utilizando *RabbitMQ* como *broker* de mensajería, lo que permitió separar el envío, entrega y recepción de mensajes de forma independiente.
3. El diseño de un modelo de datos normalizado y consistente con soporte para chats privados, historiales de mensajes, estados de lecturas y paginación optimizada.
4. La implementación de reglas de negocio y de seguridad para el manejo de idempotencia en la creación de chats, validaciones de acceso y manejo de excepciones controlado.

El servicio resultante es confiable, mantenible y está técnicamente preparado para integrarse con el frontend oficial de la plataforma. El módulo de mensajería aporta a *BeatMap* un componente crítico para su evolución como plataforma social y colaborativa, ya que se diseñó y se implementó sobre un *backend* modular que separa adecuadamente la lógica de negocio, la persistencia de datos y la comunicación en tiempo real. Esto permite que la arquitectura de *BeatMap* no solo cumpla con los requerimientos actuales, sino que también está preparada para escalar en el futuro a una arquitectura basada en microservicios cuando el volumen de usuarios y operaciones aumente de manera significativa. Ya que el *backend* de la plataforma fue construido siguiendo principios de desacoplamiento y separación de responsabilidades. Este modo, la plataforma está preparada para crecer sin comprometer su rendimiento y su estabilidad.

5.2 Limitaciones identificadas y puntos de mejora a futuro.

Aunque el módulo de mensajería desarrollado cumple plenamente con los objetivos técnicos y funcionales definidos para este proyecto, durante el proceso de implementación y validación se identificaron diversas limitaciones inherentes al alcance del proyecto, así como oportunidades claras de mejora para futuras versiones del módulo. Estas observaciones no comprometen ni la calidad ni la validez de la solución, pero sí permiten proyectar su evolución natural a medida que *BeatMap* crezca y enfrente mayores demandas operativas. Una de las limitaciones principales corresponde a la falta de soporte para mensajes multimedia, tales como imágenes, audios o videos. El modelo de datos actual está optimizado para texto, por lo que la incorporación de contenido multimedia requerirá un rediseño estructural y la integración con un servicio de almacenamiento externo como *AWS S3*, aprovechando que *BeatMap* se encuentra desplegado en la infraestructura de *AWS*. Asimismo, el servicio no incorpora mecanismos de cifrado *end-to-end*, lo que implica que la seguridad de los mensajes depende del canal *HTTPS*. En una futura versión podría integrarse cifrado asimétrico o *hashing* previo al envío para reforzar la protección frente a ataques de intermediarios. Otra limitación identificada corresponde al uso de *PostgreSQL* como motor de búsqueda de mensajes. Si bien resulta eficiente para operaciones relacionales, no es ideal para búsquedas textuales extensas. Por ello, una línea de trabajo recomendada consiste en migrar el historial de mensajes a una base de datos documental como *MongoDB*, la cual permite indexación de texto completo, búsquedas más rápidas y escalabilidad horizontal para manejar volúmenes de mensajes crecientes.

A nivel de arquitectura, el modelo de datos podría beneficiarse de procesos adicionales de normalización y optimización, especialmente si se incorporan nuevos tipos de contenido, reacciones, archivos adjuntos o eventos de chat. También se sugiere integrar herramientas de monitoreo y trazabilidad como *Sentry*, que permitirían registrar errores en tiempo real y detectar fallas antes de que impacten al usuario final. Finalmente, se identificó que algunos *endpoints* retornan objetos demasiado completos, lo que podría exponer información sensible. Por ello, se recomienda ajustar los *DTOs* para entregar únicamente los atributos estrictamente necesarios, mejorando la seguridad y evitando filtraciones de datos innecesarias. En conjunto, estas limitaciones y propuestas constituyen una hoja de ruta clara para la evolución del módulo de mensajería, fortaleciendo su seguridad, rendimiento, capacidad de extensibilidad y preparación para escenarios de alto tráfico, especialmente cuando *BeatMap* avance hacia una arquitectura desacoplada de microservicios que responda al crecimiento proyectado de su base de usuarios.

REFERENCIAS BIBLIOGRÁFICAS

Amazon. (2024). AWS. Obtenido de ¿Qué es una API RESTful?: <https://aws.amazon.com/es/what-is/restful-api/>

Auth0. (s.f.). *JSON Web Tokens*. Obtenido de auth0 Docs: <https://auth0.com/docs/secure/tokens/json-web-tokens>

auth0. (s.f.). *What is Authentication?* Obtenido de auth0: <https://auth0.com/es/intro-to-iam/what-is-authentication>

AWS. (2023). *amazon web services*. Obtenido de Amazon: <https://aws.amazon.com/es/compare/the-difference-between-rabbitmq-and-kafka/>

Casero, A. (16 de Abril de 2024). *¿Qué es el estándar AMQP (Advanced Message Queuing Protocol)?* Obtenido de KeepCoding: <https://keepcoding.io/blog/que-es-el-estandar-amqp/>

Chakray. (s.f.). *10 Ventajas Clave de las API REST en 2025: Escalabilidad, Seguridad y Más*. Obtenido de Chakray: <https://chakray.com/es/cuales-son-las-ventajas-de-una-api-rest/>

Cloudflare. (s.f.). *¿Qué es una API?* Obtenido de CloudFlare: <https://www.cloudflare.com/es-es/learning/security/api/what-is-an-api/>

datdata. (6 de Marzo de 2025). *SQL Server, MySQL y PostgreSQL: ¿Cuál elegir?* Obtenido de datdata: <https://www.datdata.com/blog/sql-server-mysql-y-postgresql-cual-elegir#:~:text=MySQL%20es%20uno%20de%20los,facilidad%20de%20uso%20y%20velocidad>

Ellingwood, J. (15 de Diciembre de 2023). *What is an ORM*. Obtenido de prisma: <https://www.prisma.io/dataguide/types/relational/what-is-an-orm>

Fanchi, C. (9 de Diciembre de 2022). *Redis: What It Is, What It Does, and Why You Should Care*. Obtenido de backendless: <https://backendless.com/redis-what-it-is-what-it-does-and-why-you-should-care/>

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Universidad de California.

Ganji, A. (12 de marzo de 2025). *Spring Boot: Simplifying Java Microservices Development*. Obtenido de medium: <https://medium.com/%40ganjiarchana06/spring-boot-simplifying-java-microservices-development-9c306c816db5>

Geeks for geeks . (4 de agosto de 2025). *Geeks for geeks*. Obtenido de Introduction to Spring Framework: <https://www.geeksforgeeks.org/advance-java/introduction-to-spring-framework/>

Github. (2025). *github docs*. Obtenido de Github: <https://docs.github.com/en/webhooks/about-webhooks>

IBM. (11 de Octubre de 2021). *IBM*. Obtenido de IBM.com: <https://www.ibm.com/mx-es/think/topics/message-brokers>

IBM. (10 de Mayo de 2024). *¿Qué es Redis?* Obtenido de IBM: <https://www.ibm.com/mx-es/think/topics/redis>

- Lgarciacantillano. (8 de Octubre de 2022). *¿Qué es la Pirámide de Pruebas?* Obtenido de Medium: <https://medium.com/nिकासource/qu%C3%A9-es-la-pir%C3%A1mide-de-pruebas-6ac4356c963f>
- Melnikov, A., & Fette, I. (Diciembre de 2011). *RFC*. Obtenido de Internet Engineering Task Force: <https://datatracker.ietf.org/doc/html/rfc6455>
- Microsoft. (22 de 10 de 2015). *Patrones en la práctica*. Obtenido de Microsoft Ignite: <https://learn.microsoft.com/es-es/archive/msdn-magazine/2009/february/patterns-in-practice-convention-over-configuration>
- Obregon, A. (3 de mayo de 2023). *Groovy and Grails — Rapid Web Application Development Made Easy*. Obtenido de Medium: <https://medium.com/@AlexanderObregon/groovy-and-grails-rapid-web-application-development-made-easy-63a378f1ef7a>
- okta. (2 de Septiembre de 2024). *Authentication vs. Authorization*. Obtenido de okta: <https://www.okta.com/identity-101/authentication-vs-authorization/>
- Parabavidez, S. (26 de abril de 2024). *Cómo elegir el mejor framework para tu web*. Obtenido de playful: <https://playfulagency.com/blog/tecnologia/como-elegir-el-mejor-framework-para-tu-web/>
- PostgreSQL. (s.f.). *A Brief History of PostgreSQL*. Obtenido de postgresql: <https://www.postgresql.org/docs/current/history.html>
- Red Hat. (10 de Octubre de 2022). *¿Qué es Apache Kafka?* Obtenido de redhat: <https://www.redhat.com/es/topics/integration/what-is-apache-kafka>
- Reddit. (2023). *what is the difference between opinionated and convention over configuration in spring boot?* Obtenido de java reddit: https://www.reddit.com/r/java/comments/1alapo3/what_is_the_difference_between_opinionated_and/
- Redgate Software. (Junio de 2025). *DB-Engines rankings*. Obtenido de DB-engines: <https://db-engines.com/en/ranking#:~:text=Spatial%20DBMS%20Vector%20DBMS%201230.38%2B3.82,44.81>
- Rocher, G., Ledbrook, P., Palmer, M., & Brown, J. (s.f.). *The Grails Framework - Reference Documentation*. Obtenido de Grails: <https://grails.apache.org/docs/1.2.1/guide/single.html>
- Ruby on rails. (s.f.). *Getting Started with Rails*. Obtenido de ruby on rails: https://guides.rubyonrails.org/getting_started.html
- Sheldon, R. (15 de Julio de 2024). *WebSocket*. Obtenido de TechTarget: <https://www.techtarget.com/whatis/definition/WebSocket>
- Spring Boot. (2025). *Overview about spring boot*. Obtenido de Spring Boot: <https://spring.io/projects/spring-boot>
- Susnjara, S., & Smalley, I. (9 de Julio de 2025). *¿Qué son las pruebas de software?* Obtenido de IBM: <https://www.ibm.com/es-es/think/topics/software-testing>

Universidad Europea. (22 de Marzo de 2022). *Blog Ingeniería: Universidad Europea*. Obtenido de Universidad Europea: <https://universidadeuropea.com/blog/para-que-sirve-gestor-base-datos/#:~:text=Un%20sistema%20gestor%20de%20bases,eficaz%20de%20administrar%20esa%20informaci%C3%B3n>