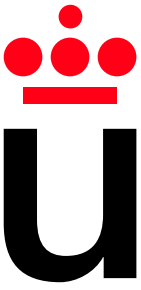

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería de Computadores

Curso 2024-2025

Trabajo Fin de Grado

DESARROLLO DE UNA APLICACIÓN DE CHAT EN TIEMPO REAL EN LA NUBE

Autor: Aitor García Camargo
Tutor: Michel Maes Bermejo

Agradecimientos

En primer lugar, me gustaría expresar mi más profundo agradecimiento a mi familia, especialmente a mis padres y a mi hermana. Con su amor, dedicación y ejemplo, no solo han sido pilares fundamentales en mi desarrollo personal y académico, sino que también han sabido mantenerme enfocado en mis metas. Su apoyo constante y sus valores han sido una guía imprescindible en este camino.

A mi pareja, con quien sueño construir un futuro lleno de proyectos y felicidad compartida, gracias por tu amor incondicional.

A mis amigos de Leganés e Illescas, gracias por las risas, los momentos de desconexión y las experiencias que han hecho este camino mucho más llevadero. Su compañía y apoyo me han recordado lo importante que es contar con amigos que alegren cada etapa de la vida.

Por último, quiero expresar mi agradecimiento a los profesores de la carrera y, especialmente, a mi tutor Michel Maes, por su guía, disposición y compromiso durante todo este proceso. Su implicación y apoyo han sido esenciales para la realización de este trabajo.

Resumen

Este Trabajo de Fin de Grado (TFG) se centra en el diseño, desarrollo e implementación de una plataforma web para comunicación en tiempo real, empleando tecnologías modernas y arquitecturas escalables. El proyecto se ha construido siguiendo un enfoque basado en microservicios, con un backend desarrollado en **Spring Boot**, un frontend dinámico en **Angular**, y un despliegue completo en **Amazon Web Services (AWS)**, líder en servicios de computación en la nube.

El objetivo principal del proyecto ha sido garantizar una solución escalable y eficiente, capaz de manejar un gran volumen de usuarios simultáneamente. Para ello, se han implementado componentes clave como un balanceador de carga mediante **AWS Elastic Load Balancer**, bases de datos distribuidas con **Amazon DocumentDB**, y un broker de mensajería basado en **Amazon MQ**. Cada iteración del desarrollo ha permitido optimizar el rendimiento y robustez de la plataforma, desde la configuración de la red privada virtual (VPC) hasta la integración de contenedores mediante **Docker** para facilitar el despliegue en diferentes entornos.

A lo largo del proyecto, el enfoque iterativo e incremental ha sido fundamental para aprender y mejorar. Cada etapa, desde la configuración inicial en local, hasta las pruebas en entornos de producción, ha brindado valiosas lecciones sobre la importancia de la computación en la nube. AWS se eligió como la plataforma ideal debido a su liderazgo en el sector y su amplio conjunto de herramientas avanzadas, que han resultado esenciales para lograr una arquitectura robusta y adaptable.

Este TFG no solo ha cumplido con los objetivos iniciales, sino que también me ha permitido aprender y manejar una amplia variedad de tecnologías modernas. La experiencia adquirida, especialmente en AWS, me ha brindado una perspectiva sólida sobre la importancia de la escalabilidad en el desarrollo de aplicaciones y cómo este enfoque será esencial en mi futuro profesional.

Palabras clave:

- Spring Boot
- Angular
- Amazon Web Services (AWS)
- STOMP
- Docker
- Escalabilidad
- JSON Web Token (JWT)
- Broker de mensajes

Índice de contenidos

Índice de tablas **XII**

Índice de figuras **XIV**

Índice de códigos **XVI**

1. Introducción **1**
1.1. Contexto . . . 1
1.2. Motivación del proyecto . . . 1

2. Objetivos **3**
2.1. Objetivos Funcionales . . . 3
2.1.1. Permitir la comunicación en tiempo real . . . 3
2.1.2. Ofrecer la creación y acceso a salas públicas . . . 4
2.2. Objetivos Técnicos . . . 4
2.2.1. Implementar alta disponibilidad y tolerancia a fallos . . . 4
2.2.2. Gestionar la comunicación y sincronización de manera eficiente . . . 4
2.2.3. Garantizar la seguridad y protección de datos . . . 4
2.2.4. Desplegar la aplicación completamente en la nube . . . 4
2.2.5. Preparar la infraestructura para una escalabilidad manual 5

3. Tecnologías, Herramientas y Metodologías **7**
3.1. Tecnologías . . . 7
3.1.1. Java . . . 7
3.1.2. Spring . . . 7
3.1.3. Spring Boot . . . 8
3.1.4. Spring Messaging . . . 8
3.1.5. Lombok . . . 8
3.1.6. Web Socket . . . 8
3.1.7. STOMP . . . 8
3.1.8. RabbitMQ . . . 9

3.1.9. ActiveMQ . . . 9
3.1.10. Reactor Netty . . . 9
3.1.11. MongoDB . . . 9
3.1.12. Spring Security . . . 10
3.1.13. JWT (JSON Web Tokens) . . . 10
3.1.14. JSON (JavaScript Object Notation) . . . 10
3.1.15. JUnit . . . 10
3.1.16. Selenium . . . 10
3.1.17. Angular . . . 11
3.1.18. TypeScript . . . 11
3.1.19. HTML . . . 11
3.1.20. CSS . . . 11
3.1.21. SockJS . . . 11
3.1.22. StompJS . . . 11
3.1.23. HAProxy . . . 12
3.1.24. Docker . . . 12
3.1.25. Bash Script . . . 12
3.2. Herramientas . . . 12
3.2.1. Maven . . . 12
3.2.2. Visual Studio Code . . . 13
3.2.3. Windows WSL (Windows Subsystem for Linux) . . . 13
3.2.4. Postman . . . 13
3.2.5. Mongo Express . . . 13
3.2.6. Git . . . 13
3.2.7. GitHub . . . 13
3.2.8. DockerHub . . . 14
3.2.9. AWS (Amazon Web Services) . . . 14
3.2.10. Por qué AWS y no otra plataforma de Cloud . . . 15
3.3. Metodologías . . . 18
3.3.1. Desarrollo Iterativo e Incremental . . . 18
3.3.2. Prácticas Ágiles Implícitas . . . 19

4. Descripción Informática **20**
4.1. Requisitos . . . 20
4.1.1. Requisitos Funcionales . . . 20
4.1.2. Requisitos No Funcionales . . . 21
4.2. Arquitectura y Análisis . . . 21
4.2.1. Primera iteriación: Arquitectura Inicial . . . 21
4.2.2. Segunda iteración: Expansión y Mejora . . . 22
4.2.3. Tercera iteración: Migración a la nube . . . 27
4.3. Diseño e Implementación . . . 29
4.3.1. Backend . . . 30

4.3.2. FrontEnd 33
4.3.3. Problemas en la Migración 35
4.4. Pruebas 36
4.5. Distribución y despliegue 37

5. Conclusiones y trabajos futuros **42**
5.1. Reflexión sobre los objetivos cumplidos 42
5.2. Aspectos pendientes y futuras ampliaciones 43
5.3. Conclusiones personales 44

Bibliografía **46**

Apéndices **49**

A. Despliegue en local **51**
A.1. Prerequisitos 51
A.2. Pasos para el despliegue 51

B. Despliegue en AWS **52**
B.1. Prerequisitos 52
B.2. Pasos para el despliegue 52

C. Repositorio GitHub **59**
C.1. Enlace al repositorio 59

Índice de tablas

3.1. Distribución del mercado Cloud en 2024 [1] 17

Índice de figuras

4.1. Arquitectura Inicial 22
4.2. Arquitectura fase 2 22
4.3. Diagrama de entidades 23
4.4. Sistema de autenticación y registro 24
4.5. Creación y gestión de salas de chat 24
4.6. Chat con nuevas funcionalidades 25
4.7. Arquitectura Final 27
4.8. Detección de JWT y autenticación de usuario 31
4.9. Recursos VPC 39

Índice de códigos

4.1. Validación de JWT en WebSocket 31
4.2. Contenedor Docker de RabbitMQ 32
4.3. Configuración RabbitMQ 33
4.4. Configuración AmazonMQ 33
4.5. Función listenerMessages() 34
4.6. Función getMessageSubject() 34
4.7. Función joinRoom(roomId: string) 35

1 Introducción

1.1. Contexto

El presente proyecto se enmarca en el ámbito del desarrollo de software moderno, donde las tecnologías web y la computación en la nube han adquirido un papel preponderante. Vivimos en una era en la que la digitalización está transformando todos los aspectos de nuestra vida diaria, desde la forma en que trabajamos y nos comunicamos hasta cómo consumimos bienes y servicios. En este contexto, el desarrollo de aplicaciones web robustas, escalables y seguras no es solo una tendencia, sino una necesidad fundamental para las organizaciones que buscan mantenerse competitivas.

1.2. Motivación del proyecto

Este proyecto surge con el objetivo de diseñar y desarrollar una **aplicación de chat distribuida en tiempo real**, explorando tecnologías cloud y arquitecturas escalables. Más allá de construir una aplicación funcional, este trabajo representa una oportunidad para **profundizar en tecnologías no vistas durante el grado**, con un enfoque en despliegue en la nube, gestión de escalabilidad y alta disponibilidad.

El desarrollo de esta solución responde a varias motivaciones clave:

1. **Aprendizaje de nuevas tecnologías**: Permite profundizar en herramientas modernas como AWS, Spring Boot, Angular y arquitecturas distribuidas, abordando problemáticas reales del desarrollo web.
2. **Demanda de aplicaciones escalables**: Los usuarios requieren servicios digitales con alta disponibilidad, respuesta en tiempo real y capacidad de crecimiento, lo que exige soluciones robustas en la nube.
3. **Experiencia práctica en Cloud Computing**: El uso de AWS en este proyecto proporciona un conocimiento valioso sobre infraestructura como servicio (IaaS), integración de bases de datos distribuidas y gestión de mensajería con Amazon MQ.
4. **Relevancia profesional**: Las competencias adquiridas en este proyecto están alineadas con las necesidades del mercado laboral, donde la computación en la nube y las arquitecturas distribuidas son cada vez más demandadas.

2

Objetivos

El objetivo principal de este proyecto es desarrollar una aplicación web de chat multisala en tiempo real, escalable y robusta. Este sistema busca proporcionar una experiencia de comunicación eficiente y fiable, manteniendo altos estándares de rendimiento y seguridad.

Para cumplir con este propósito, los objetivos se dividen en dos categorías principales: **objetivos funcionales**, relacionados con las características que ofrece la aplicación al usuario final, y **objetivos técnicos**, enfocados en los aspectos arquitectónicos y tecnológicos del sistema.

2.1. Objetivos Funcionales

2.1.1. Permitir la comunicación en tiempo real

Implementar mecanismos que garanticen una comunicación fluida y en tiempo real entre los usuarios dentro de una misma sala de chat. Esto incluye la posibilidad de enviar y recibir mensajes de texto, así como compartir imágenes y otros archivos.

2.1.2. Ofrecer la creación y acceso a salas públicas

Proveer funcionalidades para que los usuarios puedan crear, unirse y gestionar múltiples salas de chat, tanto públicas como privadas, permitiendo una experiencia personalizada y adaptada a diferentes necesidades.

2.2. Objetivos Técnicos

2.2.1. Implementar alta disponibilidad y tolerancia a fallos

Desarrollar un entorno de despliegue distribuido, utilizando balanceadores de carga y servicios cloud que aseguren la continuidad del sistema incluso ante fallos de hardware, software o infraestructura.

2.2.2. Gestionar la comunicación y sincronización de manera eficiente

Optimizar el manejo del flujo de mensajes y eventos en la aplicación para garantizar una comunicación consistente y rápida entre los usuarios. Esto incluye el uso de protocolos eficientes para mantener la sincronización en tiempo real.

2.2.3. Garantizar la seguridad y protección de datos

Incorporar medidas de seguridad avanzadas para proteger tanto la comunicación entre usuarios como los datos sensibles almacenados en el sistema. Estas medidas incluirán el uso de cifrado, autenticación robusta y cumplimiento con las mejores prácticas en seguridad informática.

2.2.4. Desplegar la aplicación completamente en la nube

Asegurar que toda la infraestructura de la aplicación se ejecute en un entorno cloud, aprovechando las ventajas de servicios gestionados, elasticidad y disponibilidad proporcionadas por la nube para facilitar el mantenimiento y la automatización del despliegue.

2.2.5. Preparar la infraestructura para una escalabilidad manual

Diseñar la infraestructura de manera modular y flexible para permitir una escalabilidad manual eficiente. Esto incluye la capacidad de añadir instancias, ajustar recursos y optimizar la arquitectura según las necesidades sin requerir una automatización completa del escalado.

3

Tecnologías, Herramientas y Metodologías

En este capítulo se listarán las tecnologías, herramientas y metodologías usadas en la aplicación.

3.1. Tecnologías

3.1.1. Java

He utilizado Java [2] como lenguaje de programación en el Backend. Se ha seleccionado debido a mi amplia experiencia en su uso y a la abundancia de bibliotecas y recursos disponibles. El proyecto está definido con la versión 17 de Java.

3.1.2. Spring

Spring [3] es un framework modular que facilita la creación de aplicaciones web. Se ha elegido por su amplia adopción en la industria y la abundancia de documentación y recursos disponibles.

3.1.3. Spring Boot

Spring Boot [4] es una herramienta que permite desarrollar aplicaciones Spring de manera rápida y sencilla, facilitando la configuración y el despliegue de proyectos. Su enfoque en la simplicidad y la reducción de la cantidad de código de configuración necesario lo convierte en una opción altamente recomendable. El proyecto utiliza la versión 3.2.8 de Spring Boot.

3.1.4. Spring Messaging

Spring Messaging [5] es un módulo que facilita la integración de sistemas de mensajería. Spring Messaging se ha utilizado para implementar la comunicación en tiempo real entre los usuarios de la aplicación de chat. Gracias a su soporte para protocolos como STOMP y WebSockets y permitiendo la integración de ActiveMQ y RabbitMQ para la gestión y distribución de los mensajes entre diferentes componentes de la aplicación de manera escalable.

3.1.5. Lombok

Lombok [6] es una biblioteca para Java que permite reducir el código repetitivo generando automáticamente constructores, getters, setters y otros métodos comunes.

3.1.6. Web Socket

WebSocket [7] es un protocolo que permite la comunicación bidireccional entre un cliente y un servidor a través de una única conexión persistente, facilitando la transferencia en tiempo real de datos.

3.1.7. STOMP

STOMP (Simple Text Oriented Messaging Protocol) [8] es un protocolo de mensajería simple basado en texto que permite la comunicación entre clientes y servidores a través de sistemas de mensajería. Resulta muy útil en una web de chat porque permite una comunicación en tiempo real bidireccional a través de WebSockets, facilitando la distribución de mensajes en un modelo de publicación-suscripción. Su simplicidad y ligereza aseguran eficiencia y baja latencia, además de ser altamente escalable.

3.1.8. RabbitMQ

RabbitMQ [9] es un sistema de mensajería que implementa el protocolo AMQP, diseñado para gestionar colas de mensajes y facilitar la comunicación entre aplicaciones o servicios distribuidos. Fue utilizado en primer lugar debido a los conocimientos previos que tenía sobre este sistema de mensajería, fue posteriormente reemplazado por ActiveMQ debido a los problemas de compatibilidad con STOMP.

STOMP plugin

RabbitMQ permite habilitar plugins [10] , dando así soporte para STOMP a través de WebSockets, este plugin permitió el uso de RabbitMQ como cola de distribución de mensajes durante las primeras versiones en local de la aplicación.

3.1.9. ActiveMQ

ActiveMQ [11] es un broker de mensajes basado en Java que implementa el protocolo JMS y soporta múltiples estándares como STOMP, AMQP y MQTT para mensajería asíncrona. El soporte nativo de STOMP lo convierte en una excelente opción para el propósito del proyecto.

3.1.10. Reactor Netty

Reactor Netty [12] es un motor de red asíncrono y reactivo que permite construir aplicaciones de red de alto rendimiento basadas en el modelo no bloqueante. En este proyecto, se utiliza para configurar un cliente TCP que permite la comunicación segura con Amazon MQ a través de STOMP. Esto incluye establecer el host, el puerto y las configuraciones de SSL del broker, y gestionar la codificación de mensajes STOMP de manera eficiente.

3.1.11. MongoDB

MongoDB [13] es una base de datos NoSQL orientada a documentos que utiliza un formato de almacenamiento basado en JSON, ofreciendo flexibilidad y escalabilidad para el manejo de datos. En este proyecto, MongoDB se utiliza para almacenar y gestionar los mensajes del chat, aprovechando su capacidad para manejar datos estructurados en JSON de forma eficiente. Esto facilita el intercambio y procesamiento de mensajes entre usuarios, permitiendo consultas

rápidas y escalabilidad para soportar un alto volumen de interacciones en tiempo real.

3.1.12. Spring Security

Spring Security [14] es un framework que proporciona soluciones para la autenticación y autorización en aplicaciones Java, permitiendo implementar políticas de acceso seguras y personalizadas. Se ha configurado para la autenticación mediante un proveedor basado en usuario y contraseña, utilizando BCrypt para el cifrado de contraseñas, además de la integración de un filtro JWT y la configuración de otras medidas como la habilitación de CORS.

3.1.13. JWT (JSON Web Tokens)

JSON Web Tokens es un estándar abierto para transmitir información de forma segura entre dos partes como un objeto JSON firmado digitalmente. En este proyecto, los JWT [15] se utilizan para autenticar a los usuarios de manera eficiente y sin necesidad de mantener sesiones en el servidor, lo que refuerza la escalabilidad y la seguridad de la aplicación.

3.1.14. JSON (JavaScript Object Notation)

JSON [16] es un formato de intercambio de datos ligero basado en texto, que utiliza una estructura de pares clave-valor. En este proyecto, se utiliza ampliamente para la transmisión de datos entre el cliente y el servidor, asegurando una comunicación eficiente y estructurada, especialmente en las interacciones relacionadas con mensajes y configuraciones en tiempo real.

3.1.15. JUnit

JUnit [17] es un framework ampliamente utilizado para realizar pruebas unitarias en aplicaciones Java. Permite crear tests parametrizables de manera sencilla y ofrece métodos muy útiles para las pruebas unitarias.

3.1.16. Selenium

Selenium [18] es una herramienta para la automatización de navegadores que permite realizar pruebas funcionales y de integración en aplicaciones web. En este proyecto se ha utilizado Selenium WebDriver en la versión 4.16.0.

3.1.17. Angular

Angular [19] es un framework de desarrollo de aplicaciones web basado en TypeScript, que permite crear interfaces de usuario dinámicas y robustas con soporte para arquitecturas modulares. En este proyecto, se ha elegido Angular a pesar de tener que ser aprendido desde cero debido a su amplia adopción en la industria y a las capacidades avanzadas que ofrece para desarrollar aplicaciones web escalables, mantenibles y altamente interactivas.

3.1.18. TypeScript

TypeScript [20] es un lenguaje de programación de código abierto que extiende JavaScript al incluir características como tipado estático y herramientas avanzadas de desarrollo.

3.1.19. HTML

HTML [21] (HyperText Markup Language) es el lenguaje estándar para estructurar y presentar el contenido de páginas web, permitiendo crear interfaces visuales para el usuario.

3.1.20. CSS

CSS [22] (Cascading Style Sheets) es un lenguaje de diseño que permite definir la presentación visual de los elementos HTML, proporcionando control sobre colores, fuentes, disposición y diseño general de las páginas web.

3.1.21. SockJS

SockJS [23] es una biblioteca de cliente/servidor que proporciona una capa de abstracción para WebSockets, asegurando compatibilidad con navegadores que no soportan WebSockets nativamente mediante transporte alternativo. Se incluyó en el proyecto para garantizar la conectividad en entornos donde WebSockets no están soportados, mejorando la robustez y accesibilidad del sistema.

3.1.22. StompJS

StompJS [24] es una biblioteca de JavaScript que permite a las aplicaciones cliente interactuar con brokers de mensajes mediante el protocolo STOMP a través

de WebSockets, simplificando la integración en aplicaciones web. Su inclusión en el proyecto es crucial en la implementación de funcionalidades en tiempo real, como el intercambio de mensajes en salas de chat, aprovechando la comunicación eficiente entre cliente y servidor mediante STOMP.

3.1.23. HAProxy

HAProxy [25] es un equilibrador de carga y proxy inverso altamente eficiente y confiable que se utiliza para distribuir el tráfico entre múltiples instancias de servidores. En este proyecto, HAProxy se emplea para garantizar una distribución uniforme de las conexiones entrantes a la aplicación. Además, HAProxy mejora la tolerancia a fallos al redirigir automáticamente las solicitudes en caso de que alguna instancia no esté disponible.

3.1.24. Docker

Docker [26] es una plataforma que permite crear, desplegar y gestionar contenedores de software. Estos contenedores encapsulan aplicaciones y sus dependencias en un entorno aislado, lo que asegura que funcionen de manera consistente en diferentes entornos. Docker se ha convertido en un estándar en la industria debido a su capacidad para simplificar el despliegue y la escalabilidad de aplicaciones.

3.1.25. Bash Script

Los scripts Bash son archivos ejecutables que contienen comandos del shell Bash, utilizados para automatizar tareas en sistemas Unix/Linux.

3.2. Herramientas

3.2.1. Maven

Maven [27] es una herramienta de gestión de proyectos y construcción que facilita la administración de dependencias y la configuración del entorno en aplicaciones Java.

3.2.2. Visual Studio Code

Visual Studio Code [28] es un editor de código altamente versátil, compatible con una amplia gama de lenguajes de programación. Utilizado para escribir el código del proyecto, aprovechando sus extensiones y características como el resaltado de sintaxis, integración con Git y depuración en tiempo real, lo que facilitó el desarrollo eficiente y organizado.

3.2.3. Windows WSL (Windows Subsystem for Linux)

WSL [29] permite a los usuarios de Windows ejecutar un entorno Linux dentro de su sistema operativo, facilitando el desarrollo y pruebas en entornos basados en Unix sin necesidad de virtualización.

3.2.4. Postman

Postman [30] es una herramienta para el desarrollo y pruebas de APIs, que permite realizar solicitudes HTTP, verificar respuestas y automatizar flujos de trabajo para garantizar la funcionalidad de las aplicaciones. También ofrece la posibilidad de realizar pruebas con WebSockets, lo que ha permitido realizar las pruebas necesarias durante el desarrollo del proyecto.

3.2.5. Mongo Express

MongoExpress [31] es una herramienta web ligera para la administración de bases de datos MongoDB. En este proyecto, se ha utilizado MongoExpress para visualizar, explorar y gestionar los datos almacenados en MongoDB de manera intuitiva.

3.2.6. Git

Git [32] es un sistema de control de versiones distribuido que permite gestionar cambios en el código fuente, facilitando la colaboración entre desarrolladores y la gestión de versiones del proyecto.

3.2.7. GitHub

GitHub [33] es una plataforma basada en Git que proporciona repositorios alojados en la nube para gestionar código fuente, colaborar en proyectos y realizar

integración continua.

3.2.8. DockerHub

DockerHub [34] es un servicio de repositorios en línea que permite almacenar, compartir y gestionar imágenes de contenedores Docker, facilitando su despliegue en múltiples entornos.

3.2.9. AWS (Amazon Web Services)

AWS [35] es una plataforma de servicios en la nube que proporciona infraestructura escalable, almacenamiento, bases de datos, herramientas de redes, y servicios de computación para desarrollar y desplegar aplicaciones de manera flexible y segura.

EC2

Amazon Elastic Compute Cloud (EC2) es un servicio que proporciona capacidad de cómputo escalable en la nube, permitiendo ejecutar y gestionar instancias de máquinas virtuales.

VPC

Amazon Virtual Private Cloud (VPC) permite crear redes privadas virtuales en AWS, proporcionando control sobre el entorno de red, incluidas subredes, tablas de enrutamiento y gateways.

LoadBalancer

Load Balancer distribuye automáticamente el tráfico de red entrante entre múltiples instancias de EC2 para mejorar la tolerancia a fallos y la disponibilidad de las aplicaciones.

DocumentDB

DocumentDB es un servicio de base de datos compatible con MongoDB, diseñado para manejar cargas de trabajo de bases de datos documentales en la nube de forma segura, escalable y eficiente.

AmazonMQ

AmazonMQ es un servicio de mensajería que proporciona brokers gestionados para protocolos como ActiveMQ, permitiendo comunicación eficiente y escalable entre aplicaciones distribuidas.

3.2.10. Por qué AWS y no otra plataforma de Cloud

En la actualidad, existen múltiples proveedores de servicios en la nube, como **Microsoft Azure**, **Google Cloud Platform (GCP)**, **IBM Cloud**, y **Oracle Cloud**, entre otros. Cada uno de ellos ofrece características únicas, pero la elección de **Amazon Web Services (AWS)** para este proyecto se basa en razones técnicas, estratégicas y prácticas que lo convierten en la opción más adecuada:

1. **Liderazgo en el mercado:** AWS es el líder indiscutible en el sector de la nube, (Tabla 1.1) [1] con la mayor cuota de mercado global y una amplia base de clientes, desde startups hasta grandes corporaciones. Esto garantiza un ecosistema maduro, estable y confiable.

2. **Amplia gama de servicios:** AWS ofrece una de las mayores colecciones de servicios de infraestructura y software, incluyendo bases de datos, herramientas de inteligencia artificial, redes, almacenamiento, herramientas de desarrollo y mucho más. Para este proyecto, servicios como **EC2**, **DocumentDB**, y **Amazon MQ** encajan perfectamente con las necesidades técnicas planteadas.

3. **Escalabilidad probada:** AWS ha demostrado su capacidad para manejar desde pequeñas aplicaciones hasta sistemas globales de gran escala. Esto proporciona confianza en que la infraestructura puede crecer junto con las demandas de la aplicación.

4. **Flexibilidad y personalización:** A diferencia de otras plataformas, AWS permite un alto grado de personalización en la configuración de infraestructura, lo que lo hace ideal para proyectos que requieren un control granular, como la creación de redes privadas virtuales (VPCs) y configuraciones específicas de balanceadores de carga y subredes.

5. **Disponibilidad global:** Con la mayor red de centros de datos distribuidos en todo el mundo, AWS permite desplegar aplicaciones en múltiples regiones y zonas de disponibilidad, asegurando baja latencia y alta disponibilidad para los usuarios.

6. **Ecosistema de soporte y comunidad:** AWS cuenta con una extensa documentación, foros comunitarios y servicios de soporte técnico que

facilitan el aprendizaje y la resolución de problemas. Este factor ha sido especialmente valioso en este proyecto, dado el aprendizaje intensivo requerido en tecnologías cloud.

7. **Costos ajustados al consumo:** AWS opera bajo un modelo de "pago por uso", lo que permite ajustar los costos a las necesidades reales del proyecto. Además de gran oferta que tiene en el "free tier"durante el primer año.

Comparativa con otras plataformas

Aunque otras plataformas como **Microsoft Azure** o **Google Cloud Platform** ofrecen características destacadas, estas no igualan el nivel de madurez, diversidad de servicios y presencia global de AWS. Por ejemplo:

- **Microsoft Azure** es una excelente opción para organizaciones que ya usan productos de Microsoft como Windows Server o Active Directory. Sin embargo, en este proyecto, la dependencia de tecnologías específicas de Microsoft no era relevante. Además, AWS cuenta con una mayor gama de servicios especializados y soporte para casos de uso más diversos.
- **Google Cloud Platform** se destaca en áreas como el análisis de datos y la inteligencia artificial, pero no tiene la misma profundidad en servicios básicos de infraestructura ni la misma presencia global que AWS.
- Otras opciones como **IBM Cloud** o **Oracle Cloud** están más orientadas a nichos específicos, como bases de datos empresariales o soluciones on-premise híbridas, que no eran prioridad en este proyecto.

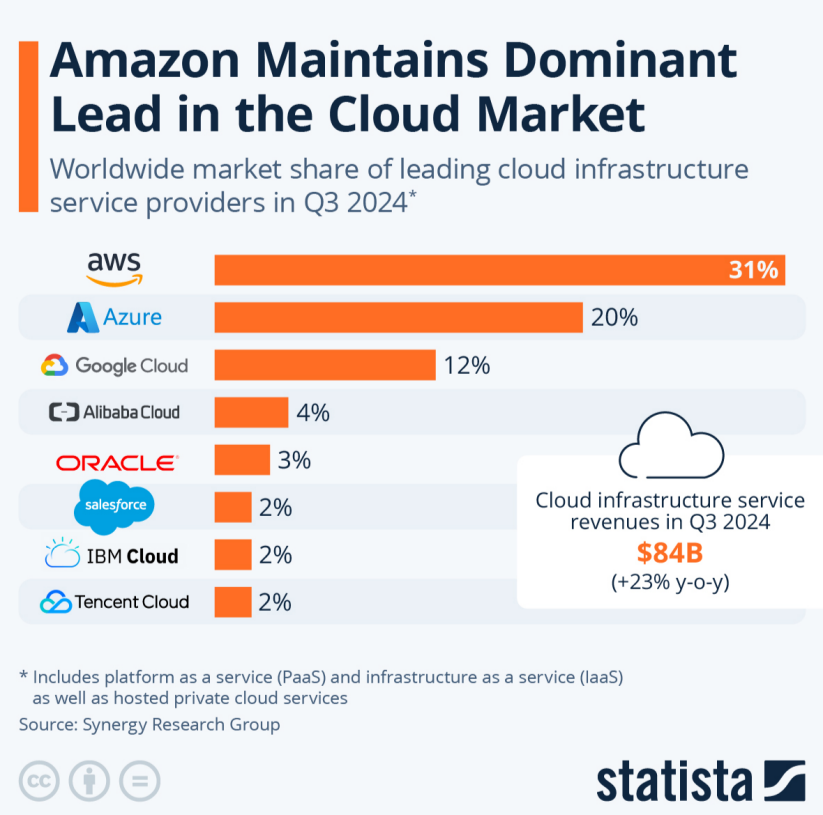

Tabla 3.1: Distribución del mercado Cloud en 2024 [1]

3.3. Metodologías

El desarrollo de este proyecto ha seguido un enfoque iterativo e incremental, complementado con el uso de prácticas estándar para la gestión del código y el despliegue continuo.

3.3.1. Desarrollo Iterativo e Incremental

La metodología iterativa e incremental permitió construir una aplicación funcional desde la primera fase y agregar nuevas funcionalidades en ciclos posteriores. Cada fase, con una duración media de tres meses, estuvo marcada por objetivos claros:

- **Fase 1**
  - **Versión 1.0:** Implementación inicial de un chat en tiempo real con soporte para múltiples salas.
  - Resultado: Una aplicación funcional que sirvió como base para las siguientes iteraciones.
- **Fase 2**
  - **Versión 2.0:** Adición de funcionalidades avanzadas para enriquecer la experiencia del usuario.
  - Resultado: Expansión del alcance de la aplicación con características mejoradas.
- **Fase 3**
  - **Versión 3.0:** Despliegue distribuido de la aplicación en la nube utilizando Amazon Web Services (AWS).
  - Resultado: Una aplicación escalable y distribuida, preparada para entornos de producción.

Este enfoque permitió adaptarse a cambios y refinar la funcionalidad en función del aprendizaje obtenido en cada ciclo.

3.3.2. Prácticas Ágiles Implícitas

Aunque no se utilizó formalmente un marco ágil como **Scrum** o **Kanban**, el desarrollo integró prácticas clave del pensamiento ágil:

- **Entrega incremental**: Cada iteración incluyó funcionalidades listas para ser probadas y utilizadas, asegurando avances concretos en cada ciclo.
- **Flexibilidad y adaptabilidad**: Se permitió ajustar prioridades y objetivos en función del aprendizaje y los cambios en los requisitos.
- **Enfoque en objetivos específicos**: Los ciclos cortos se orientaron a resolver problemas concretos, lo que contribuyó a mantener un progreso constante y medible.

La adopción de estas prácticas fomentó la colaboración efectiva, la retroalimentación continua y una implementación ajustada a las necesidades reales del proyecto.

4

Descripción Informática

En este capítulo se proporciona una descripción exhaustiva del proyecto de software realizado, detallando cada aspecto crítico de su desarrollo, desde los requisitos iniciales hasta la distribución final. El objetivo es ofrecer una visión completa y detallada de cómo se ha llevado a cabo el proyecto, abarcando tanto la planificación y el diseño como la implementación y las pruebas. A lo largo de este capítulo, se presentan los elementos clave y las decisiones técnicas que han guiado el desarrollo del software, proporcionando una base sólida para comprender el producto final.

4.1. Requisitos

En esta sección abordaremos los requisitos planteados en la aplicación, tanto los iniciales como los que se han ido planteando a lo largo del desarrollo. También se nombrarán otro tipo de requisitos a la hora de desarrollar el proyecto que no están relacionados en sí con el propio funcionamiento de la aplicación.

4.1.1. Requisitos Funcionales

- RF1. Dos o más usuarios deben poder conectarse a una sala indicando su nickname y el nombre de la sala

- RF2. Dos o más usuarios deben poder enviar y recibir mensajes desde una sala.
- RF3. Al conectarse a la sala un usuario, el sistema enviará automáticamente un mensaje.
- RF4. Los usuarios deben poder registrarse, tener un login y una sesión.
- RF5. Los usuarios deben poder enviar ficheros a otros usuarios.
- RF6. Los usuarios deben poder recuperar los mensajes previos de una sala de chat.

4.1.2. Requisitos No Funcionales

- RNF1. Debe haber más de un servidor (mínimo 2) corriendo la misma aplicación para aguantar el volumen de usuarios.
- RNF2. La aplicación (con más de un servidor) debe ser ofrecida utilizando un balanceador de carga.
- RNF3. La aplicación debe poder levantarse como un docker-compose.
- RNF4. La aplicación debe poder desplegarse en AWS, haciendo uso de sus tecnologías.

4.2. Arquitectura y Análisis

La arquitectura de la aplicación ha sido diseñada y desarrollada siguiendo una metodología **iterativa e incremental**, permitiendo adaptarse a los requerimientos del proyecto conforme evolucionaban las necesidades. Este capítulo detalla las distintas etapas de desarrollo, partiendo de una arquitectura inicial simple hasta una solución más avanzada y distribuida, preparada para entornos de producción.

4.2.1. Primera iteriación: Arquitectura Inicial

En la primera etapa del desarrollo, se construyó una arquitectura simple para validar los conceptos básicos del sistema y facilitar la implementación inicial.

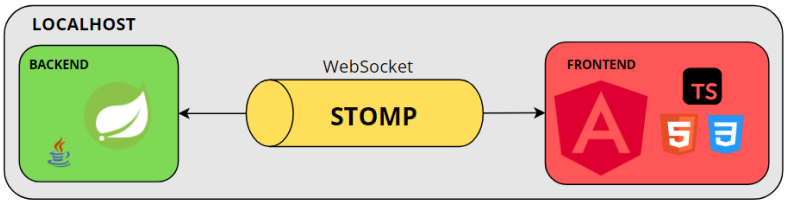

Figura 4.1: Arquitectura Inicial

Esta fase se centró en el desarrollo de un frontend básico que permitiera a los usuarios conectarse a una sala de chat sin necesidad de autenticación. Los usuarios solo debían ingresar un nombre y el nombre de la sala para participar en la conversación. Por el lado del backend, se implementó un único controlador encargado de recibir los mensajes enviados desde el frontend a través de WebSocket, utilizando el protocolo STOMP, y reenviarlos a todos los usuarios conectados a la misma sala.

El foco principal estuvo en garantizar la conexión y la comunicación en tiempo real. La arquitectura inicial permitió verificar la funcionalidad del sistema en un entorno local, donde todos los componentes se ejecutaban en un único entorno de desarrollo. Aunque esta implementación era sencilla y limitada, fue fundamental para comprobar que los conceptos básicos de la arquitectura funcionaban correctamente.

4.2.2. Segunda iteración: Expansión y Mejora

La siguiente iteración se centró en ampliar la funcionalidad y preparar el sistema para entornos distribuidos. Se introdujeron importantes mejoras tanto en las capacidades del sistema como en su diseño técnico.

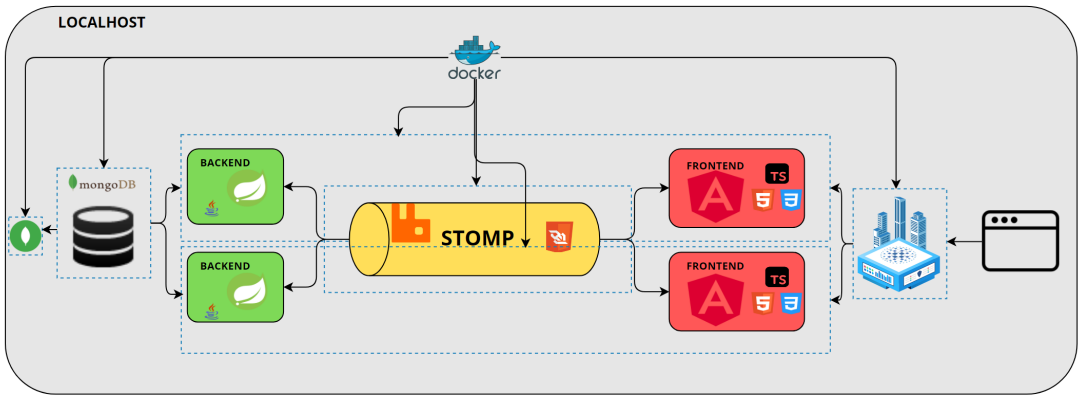

Figura 4.2: Arquitectura fase 2

1. **Persistencia de Datos:**

Para añadir nuevas funcionalidades y mejorar la experiencia del usuario, se integró una base de datos NoSQL, **MongoDB**. Esta base de datos permite gestionar datos semiestructurados, como los mensajes de chat, la información de los usuarios y las salas de chat, en un formato optimizado llamado **BSON** (*Binary JSON*). Aunque BSON es similar a JSON en su estructura, ofrece ventajas adicionales como el soporte para tipos de datos binarios y una mayor eficiencia en la serialización y deserialización de documentos [36].

MongoDB fue seleccionada en lugar de una base de datos relacional debido a su capacidad de escalabilidad horizontal y su flexibilidad para almacenar documentos dinámicos, lo que facilita el manejo de datos de distintos formatos sin necesidad de esquemas rígidos. Este enfoque resulta especialmente útil en aplicaciones de chat en tiempo real, donde la estructura de los datos puede variar y evolucionar a medida que se incorporan nuevas funcionalidades.

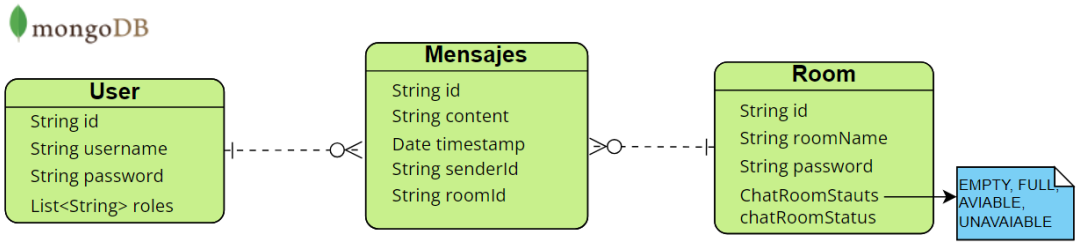

Figura 4.3: Diagrama de entidades

Gracias a la persistencia de datos, se implementaron las siguientes funcionalidades:

- **Sistema de autenticación y registro:** Los usuarios pueden registrarse e iniciar sesión, con sesiones gestionadas mediante cookies.

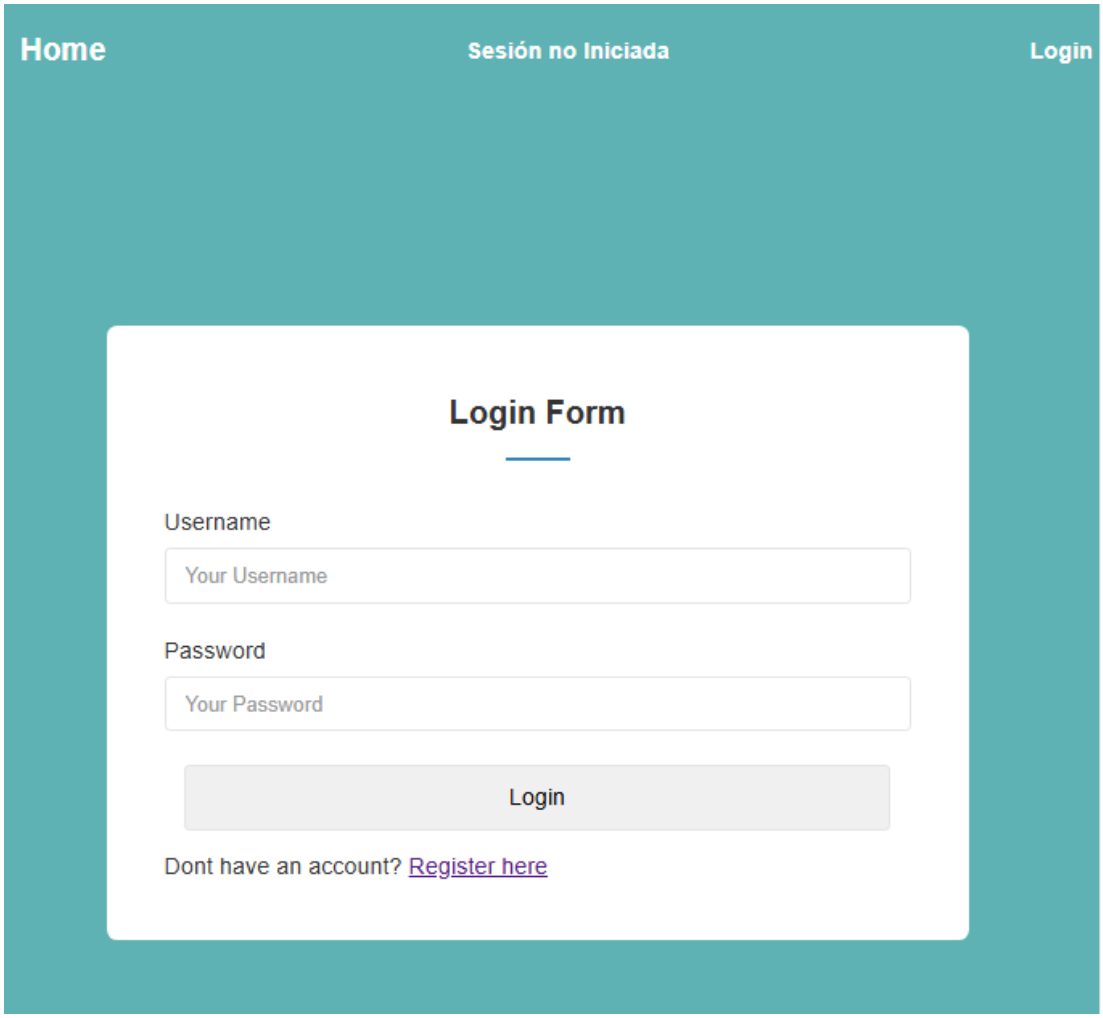

Figura 4.4: Sistema de autenticación y registro

- **Creación y gestión de salas de chat:** Los usuarios pueden crear nuevas salas y unirse a salas existentes.

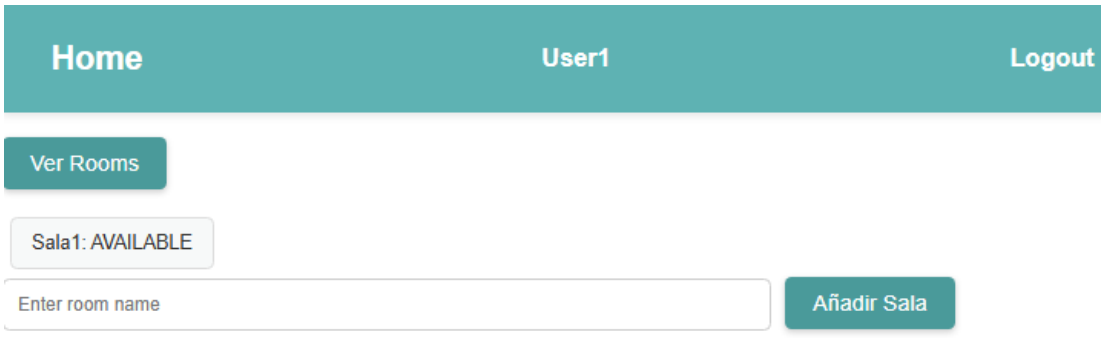

Figura 4.5: Creación y gestión de salas de chat

- **Historial de mensajes:** Los mensajes ahora se almacenan en la base de datos, permitiendo a los usuarios recuperar conversaciones previas al ingresar en una sala.

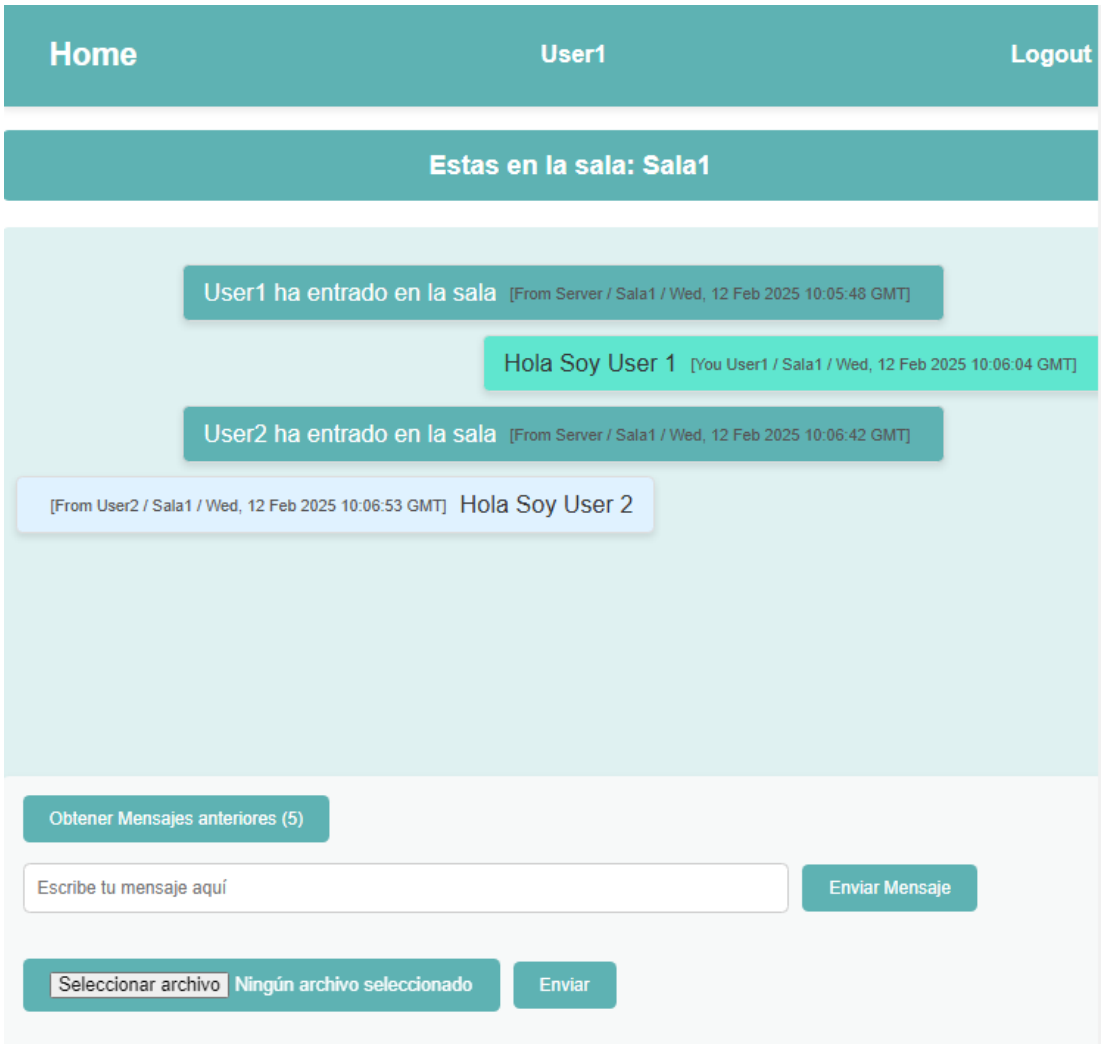

Figura 4.6: Chat con nuevas funcionalidades

2. **Instancias Múltiples:**

   Para simular escenarios de alta disponibilidad y escalabilidad, se desplegaron múltiples servicios de la aplicación utilizando Docker. En lugar de mantenerlos separados, Angular se incluyó dentro del backend, lo que permitió consolidar ambos servicios en un solo contenedor. Esta configuración simplificó la gestión del sistema al reducir el número de servicios independientes, mientras mantenía la modularidad del desarrollo.

   Adicionalmente, se incorporó **RabbitMQ** como *broker* de mensajes, lo que aseguró que todos los mensajes enviados por un cliente llegaran a

todas las instancias conectadas. RabbitMQ incorpora el protocolo **STOMP** mediante la instalación de uno de sus plugins, que permite que los diferentes contenedores del sistema gestionen mensajes de manera sincronizada y eficiente. Esto resultó fundamental para mantener la consistencia en tiempo real entre las múltiples instancias y garantizar que las comunicaciones entre usuarios, independientemente de la instancia a la que estuvieran conectados, fueran fluidas y confiables.

Gracias a esta configuración, el sistema se mantuvo preparado para manejar mayores volúmenes de usuarios y solicitudes, al tiempo que aseguraba una experiencia consistente y robusta para todos los usuarios.

3. **Balanceador de Carga:**

   Se introdujo un balanceador de carga (**HAProxy**) para distribuir equitativamente las solicitudes entre las instancias del frontend y backend. Esto no solo mejoró el rendimiento del sistema, sino que también aumentó la tolerancia a fallos, ya que una sobrecarga o fallo en una instancia no afectaría a las demás.

4. **Seguridad** La seguridad del sistema se fortaleció notablemente durante esta iteración:

   - **Gestión de sesiones con cookies y JWT**: Se reforzó la seguridad con la implementación de JSON Web Tokens (JWT), que se envían como cookies y se verifican en cada interacción para garantizar la autenticación.
   - **Restricciones en la conexión STOMP:** La conexión WebSocket ahora exige un token JWT válido durante la comunicación, asegurando que solo usuarios autenticados puedan enviar o recibir mensajes.

5. **Nuevas Funcionalidades** Además de la autenticación y el historial de mensajes, se añadieron otras características importantes para enriquecer la experiencia del usuario:

   - **Envío de imágenes y archivos de texto:** Los usuarios pueden enviar imágenes y archivos `.txt` durante las conversaciones. Estos archivos no se almacenan en la base de datos, sino que se procesan de forma temporal mientras son transmitidos.

6. **Despliegue con Docker Compose:**

   Durante esta iteración, se añadió la capacidad de desplegar todo el sistema mediante Docker-Compose, lo que facilitó enormemente la configuración y la gestión de los servicios del proyecto.

   El archivo de configuración de Docker Compose permite levantar todos los servicios necesarios, incluyendo el backend y frontend, la base de datos

MongoDB y el servicio MongoExpress, RabbitMQ y el HAProxy , con un solo comando. Este enfoque simplifica el proceso de despliegue tanto para entornos de desarrollo como de producción, asegurando que todos los componentes trabajen en conjunto de manera eficiente.

Una característica destacable de esta implementación es que, dentro del servicio de **RabbitMQ**, se configura automáticamente la instalación del plugin **STOMP**. Este plugin es esencial para que RabbitMQ pueda gestionar la comunicación en tiempo real mediante el protocolo STOMP, que es clave para la interacción fluida entre clientes y servidores en la aplicación.

Esta iteración representó un gran avance en la arquitectura del sistema, transformándolo en una solución más robusta y preparada para manejar múltiples usuarios, distribuidos en un entorno dinámico y seguro.

4.2.3. Tercera iteración: Migración a la nube

La iteración final se enfocó en trasladar el sistema a un entorno de producción utilizando servicios en la nube. Este cambio permitió mejorar la disponibilidad, escalabilidad y seguridad de la aplicación, además de simplificar la gestión operativa gracias a los servicios gestionados que ofrece AWS. A continuación, se detallan los principales cambios realizados en esta etapa.

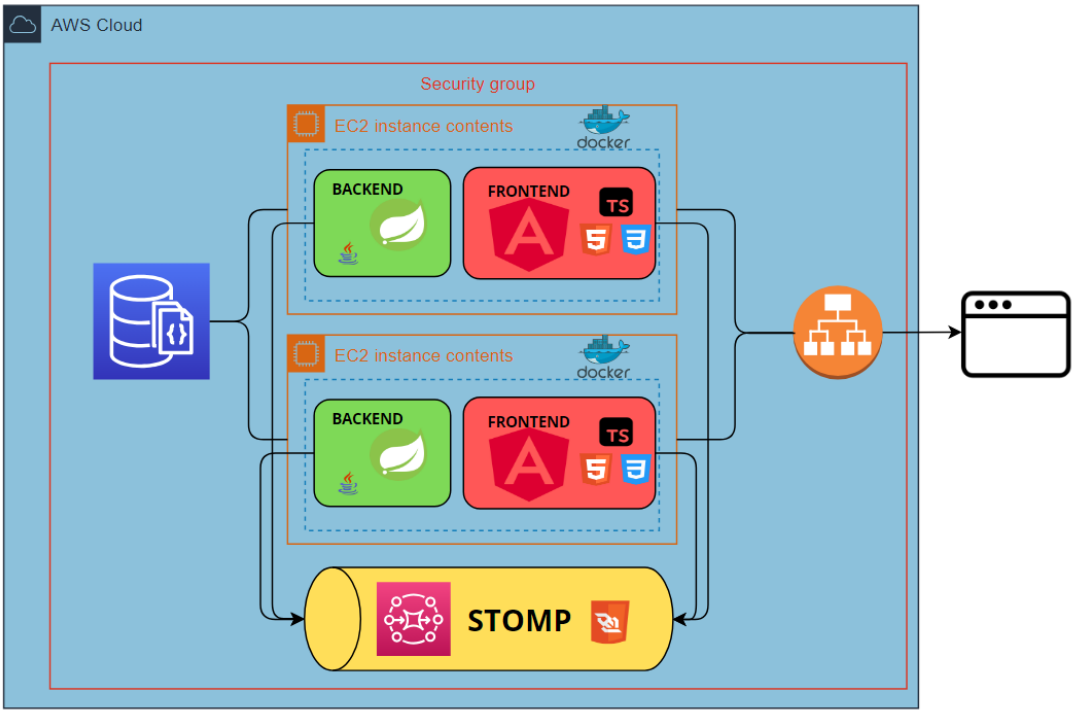

Figura 4.7: Arquitectura Final

Migración a AWS:

1. **Instancias EC2**

   Se desplegaron instancias EC2 configuradas para alojar los contenedores Docker de la aplicación. Cada instancia alberga tanto el backend como el frontend (Angular integrado dentro del backend).

2. **Configuración de VPC**

   La red fue configurada dentro de una Virtual Private Cloud (VPC) para garantizar el aislamiento y la seguridad de los recursos. Dentro de la VPC se establecieron subredes públicas y privadas:

   - **Subredes públicas:** Aquí se ubicaron las instancias EC2 y el balanceador de carga, permitiendo su acceso desde Internet.
   - **Subredes privadas:** Aquí se alojaron recursos sensibles, como la base de datos DocumentDB y el *broker* Amazon MQ, asegurando que no estuvieran expuestos directamente al exterior.

3. **Grupos de seguridad**

   Se configuraron grupos de seguridad específicos para controlar el tráfico hacia y desde los recursos:

   - El balanceador de carga permitía acceso únicamente a través de los puertos HTTP/HTTPS y las conexiones WebSocket.
   - Las instancias EC2 aceptaban solo tráfico interno desde el balanceador de carga y conexiones STOMP provenientes del *broker*.
   - DocumentDB y Amazon MQ restringieron su acceso a las instancias ubicadas dentro de la misma VPC.

4. **Balanceador de carga (Elastic Load Balancer)**

   El balanceador de carga HAProxy utilizado en iteraciones anteriores fue reemplazado por un Elastic Load Balancer (ELB) de AWS, que manejaba el tráfico entrante y lo distribuía equitativamente entre las instancias EC2 activas. Esto permitió:

   - Alta disponibilidad y tolerancia a fallos.

5. **Base de datos DocumentDB**

   La base de datos MongoDB local fue reemplazada por DocumentDB, un servicio gestionado compatible con MongoDB que ofrece:

   - Alta disponibilidad mediante réplicas distribuidas en distintas zonas de disponibilidad.
   - Escalabilidad automática para manejar picos de carga.

- Copias de seguridad automáticas, garantizando la durabilidad de los datos y simplificando la gestión.

6. **Amazon MQ (ActiveMQ)**

   El *broker* RabbitMQ fue reemplazado por Amazon MQ, un servicio gestionado que soporta ActiveMQ nativamente. Este cambio trajo varias ventajas:

   - Elimina la necesidad de gestionar manualmente la infraestructura del *broker*.
   - Ofrece colas distribuidas y duraderas, que aseguran que todos los mensajes enviados sean entregados y replicados en diferentes instancias.
   - La integración con VPC permitió que la comunicación entre Amazon MQ y las instancias EC2 fuera privada y segura.

Impacto de la Migración

1. **Escalabilidad:**

   Gracias a los servicios gestionados de AWS, como el balanceador de carga y DocumentDB, que pueden manejar aumentos en el tráfico y el volumen, el sistema se encuentrá mejor preparado para poder soportar escalabilidad.

2. **Alta disponibilidad:**

   La configuración distribuida dentro de la VPC, junto con el uso de servicios gestionados como Amazon MQ, garantiza que la aplicación permanezca operativa incluso si fallan algunos componentes o instancias.

3. **Seguridad:**

   El uso de subredes privadas, grupos de seguridad detallados y recursos aislados dentro de la VPC ha mejorado significativamente la protección del sistema frente a accesos no autorizados.

Con esta iteración, el sistema quedó completamente preparado para operar en un entorno de producción robusto, escalable y seguro, capaz de manejar cargas variables y soportar un gran número de usuarios en tiempo real.

4.3. Diseño e Implementación

En este capítulo se describen aspectos relevantes del diseño y la implementación de la aplicación, destacando soluciones técnicas a desafíos encontrados y los elementos clave del desarrollo. Además, se incluyen detalles técnicos y fragmentos de código fuente que ilustran dichas implementaciones.

4.3.1. Backend

Interceptor de Canal para Validación de JWT

Uno de los elementos más críticos en la implementación fue la seguridad en las conexiones WebSocket. Se desarrolló un interceptor personalizado que valida tokens JWT durante la conexión inicial de los clientes al servidor. Esto asegura que solo usuarios autenticados puedan establecer sesiones de comunicación.

El interceptor funciona interceptando los mensajes enviados por los clientes antes de ser procesados por el servidor. En particular, se verifica si el mensaje corresponde a un comando de conexión (`CONNECT`) y, si es así, se extrae y valida el token JWT.

Código 4.1: Validación de JWT en WebSocket

```java
@Override
public Message<?> preSend(Message<?> message, MessageChannel channel) {

    StompHeaderAccessor accessor = StompHeaderAccessor.wrap(message);
    System.out.println("--Channel Interceptor--");
    System.out.println(message);

      // Verificar si el mensaje es del tipo CONNECT
    if (accessor.getCommand() == StompCommand.CONNECT) {
        // Buscar el token JWT en los headers nativos
        String authHeader = accessor.getFirstNativeHeader("Authorization");

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String jwtToken = authHeader.substring(7);  // Eliminar "Bearer "
            System.out.println("JWT detectado: " + jwtToken);

            String decryptToken = SecurityCipher.decrypt(jwtToken);
            System.out.println("JWT Decrypt: " + decryptToken);

            if (jwtTokenProvider.validateToken(decryptToken)) {
                String username = jwtTokenProvider.getUsername(decryptToken);
                System.out.println("Usuario autenticado: " + username);

               // Establecer la autenticacion en el contexto de seguridad
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(username, null, new
                        ArrayList<>());
                SecurityContextHolder.getContext().setAuthentication(authentication);
            } else {
                System.out.println("Token invalido");
            }
        } else {
            System.out.println("No se encontro token JWT en los headers.");
        }
    }
     return message;
 }
```

```
backend_spring_2  | --Channel Interceptor--
backend_spring_2  | GenericMessage [payload=byte[0], headers={simpMessageType=CONNECT, stomp
Command=CONNECT, nativeHeaders={Authorization=[Bearer yI9zDcX9gFoXVHaW14iwO2OR2i3o93814vyCgQ
CvnmQZMeSepsi8MAXFEOHYqZ0mzvWHkiZxiY8KuW08wDqqyWjNpl5hddAvqvsRUKi5NUd9Uvi39NQfbEE512CInYZ0TD
YK/DDSw9hG1Tcm2+sdRB9NxxHiVIAQbVO/L3bFsP6grRU3pm/uAJaATzJhsAo9QKd3+u9TberuuBOrOy+sCaUE1zvPN3
vU/fEohUAsYQ7dWOrG2xZF0PmA8StwPyHz], accept-version=[1.2,1.1,1.0], heart-beat=[10000,10000]}
, simpSessionAttributes={}, simpHeartbeat=[J@36858246, simpUser=UsernamePasswordAuthenticati
onToken [Principal=org.springframework.security.core.userdetails.User [Username=user1, Passw
ord=[PROTECTED], Enabled=true, AccountNonExpired=true, CredentialsNonExpired=true, AccountNo
nLocked=true, Granted Authorities=[ROLE_USER]], Credentials=[PROTECTED], Authenticated=true,
 Details=WebAuthenticationDetails [RemoteIpAddress=172.20.0.7, SessionId=null], Granted Auth
orities=[ROLE_USER]], simpSessionId=l3q4xpwg}]
backend_spring_2  | JWT detectado: yI9zDcX9gFoXVHaW14iwO2OR2i3o93814vyCgQCvnmQZMeSepsi8MAXFE
OHYqZ0mzvWHkiZxiY8KuW08wDqqyWjNpl5hddAvqvsRUKi5NUd9Uvi39NQfbEE512CInYZ0TDYK/DDSw9hG1Tcm2+sdR
B9NxxHiVIAQbVO/L3bFsP6grRU3pm/uAJaATzJhsAo9QKd3+u9TberuuBOrOy+sCaUE1zvPN3vU/fEohUAsYQ7dWOrG2
xZF0PmA8StwPyHz
backend_spring_2  | JWT Decrypt: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyMSIsImF1dGgiOlt7ImF1dG
hvcml0eSI6IlJPTEVfUk9MRV9VU0VSIn1dLCJpYXQiOjE3MjgzNzY2OTYsImV4cCI6MTcyODM4MjA5Nn0.jfjNAq88zQ
RBsk5pnFh9KbIMKhlCmTM0HXcDNMuJI6s
backend_spring_2  | Usuario autenticado: user1
```

Figura 4.8: Detección de JWT y autenticación de usuario

Este enfoque permitió integrar la autenticación JWT con las conexiones WebSocket, fortaleciendo la seguridad del sistema al garantizar que solo usuarios válidos pudieran participar en las conversaciones.

Cambio del Broker de Mensajes

Durante la iteración de migración a la nube, uno de los cambios más significativos fue la transición de RabbitMQ con un plugin STOMP a **Amazon MQ** con soporte nativo de ActiveMQ. Este cambio se realizó debido a problemas encontrados con la gestión del plugin STOMP en entornos de producción, donde surgieron desafíos de configuración y compatibilidad.

RabbitMQ, por defecto, no soporta STOMP, pero dado que era la tecnología de mensajería con la que se había trabajado previamente, se decidió inicialmente utilizarlo como solución de mensajería. Para habilitar el soporte de STOMP en RabbitMQ, se incluyó el plugin correspondiente al configurar y ejecutar el contenedor Docker. Este enfoque permitió resolver las limitaciones iniciales y continuar con el desarrollo en un entorno conocido.

La configuración del contenedor Docker para RabbitMQ incluyó la activación del plugin STOMP al iniciar el servicio, como se muestra a continuación:

Código 4.2: Contenedor Docker de RabbitMQ

```
1    rabbitmq:
2    image: rabbitmq:3-management
3    container_name: rabbitmq
4    environment:
5      - RABBITMQ_DEFAULT_USER=guest
6      - RABBITMQ_DEFAULT_PASS=guest
7    ports:
8      - 61613:61613  # Puerto STOMP
9      - 15672:15672  # Puerto para la interfaz de gestion de
           RabbitMQ (UI)
10   volumes:
11     - ./rabbitmq/data:/var/lib/rabbitmq/mnesia  #
           Persistencia de datos
12   entrypoint: >
13      sh -c "rabbitmq-plugins enable --offline
           rabbitmq_web_stomp && rabbitmq-server"
```

Esta configuración habilitó el soporte para STOMP en RabbitMQ de manera automática al iniciar el contenedor, permitiendo que el sistema gestionara mensajes utilizando este protocolo.

En el entorno local, se mantuvo esta configuración para facilitar el desarrollo y las pruebas. Sin embargo, al migrar a la nube, se optó por **Amazon MQ**, que ofrece soporte nativo para ActiveMQ y elimina la necesidad de gestionar manualmente plugins o configuraciones adicionales. Además, Amazon MQ proporciona una solución administrada que incluye características avanzadas como alta disponibilidad y escalabilidad.

Código 4.3: Configuración RabbitMQ

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
    registry.setApplicationDestinationPrefixes("/app");
        registry.enableStompBrokerRelay("/queue", "/topic")
            // Prefijos para el broker
            .setUserDestinationBroadcast("/topic/unresolved.user.dest")
            .setUserRegistryBroadcast("/topic/registry.broadcast")
            .setRelayHost("rabbitmq")  // Host del broker de RabbitMQ
            .setRelayPort(61613) // Puerto del broker
            .setClientLogin("guest")
            .setClientPasscode("guest");
}
```

Código 4.4: Configuración AmazonMQ

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {

    // Configura el cliente TCP con Reactor Netty para Amazon MQ
    ReactorNettyTcpClient<byte[]> tcpClient = new
        ReactorNettyTcpClient<>(builder -> builder
    .host(amazonMQProperties.getRelayHost())  // Host del broker de Amazon MQ
    .port(amazonMQProperties.getRelayPort())  // Puerto del broker
    .secure(SslProvider.defaultClientProvider())  // Configuracion SSL
    , new StompReactorNettyCodec());  // Codigo de codificacion de STOMP

    // Configuracion del Message Broker para STOMP
    registry.setApplicationDestinationPrefixes("/app");
    registry.enableStompBrokerRelay("/queue", "/topic")  // Prefijos para el
        broker
        .setUserDestinationBroadcast("/topic/unresolved.user.dest")
        .setUserRegistryBroadcast("/topic/registry.broadcast")
        .setAutoStartup(true)
        .setSystemLogin(amazonMQProperties.getUser())
        .setSystemPasscode(amazonMQProperties.getPassword())
        .setClientLogin(amazonMQProperties.getUser())
        .setClientPasscode(amazonMQProperties.getPassword())
        .setTcpClient(tcpClient);
}
```

4.3.2. FrontEnd

Implementación del Listener de mensajes

La gestión de los mensajes en tiempo real es una característica clave del sistema, y en esta sección se describe cómo se implementó el listener de mensajes en el componente de las salas de chat, así como las funciones asociadas del servicio WebSocket.

En el componente Room, la función listenerMessages() permite suscribirse a los mensajes recibidos a través de un servicio WebSocket. La lógica de esta función se encarga de procesar y formatear los mensajes para mostrarlos en la interfaz del usuario.

Código 4.5: Función listenerMessages()

```
1     listenerMessages(): void {
2         this.messageSuscription =
              this.webSocketService.getMessageSubject().subscribe((messages: any)
              => {
3           console.log(messages);
4           this.messagesSesion = messages.map((item: any) => ({
5             message_username: item.username,
6             message_room: item.room,
7             message_type: (item.fileType === 'image/png' || item.fileType ===
                  'image/jpeg') ? 'img' : (item.fileType === 'text/plain' ? 'txt'
                  : 'chatMessage'),
8             message_side: item.username === 'Server' ? 'Server' :
                  (item.username === this.userId ? 'Yours' : 'Others'),
9             message_content: item.filename
10              ? `data:${item.fileType};base64,${item.content}` // Si es un
                    archivo, formatea el contenido como una imagen base64
11              : item.content, // Si es texto, simplemente usa el contenido
12            message_timeStamp: new Date(item.timeStamp).toUTCString()
13          }));
14          console.log(this.messagesSesion);
15        });
16    }
```

- **Suscripción a mensajes**: La función se suscribe a un `Subject` gestionado por el servicio WebSocket.
- **Formateo de mensajes**: Cada mensaje es procesado para incluir información como el usuario que lo envió, el tipo de mensaje (texto, imagen, archivo), el contenido y su marca de tiempo.
- **Clasificación visual**: Los mensajes se clasifican en tres tipos: mensajes del servidor, mensajes enviados por el usuario actual, y mensajes de otros usuarios.

El servicio WebSocket centraliza la lógica para suscribirse a salas, publicar mensajes, y manejar la sincronización entre usuarios en tiempo real.

getMessageSubject()

Esta función proporciona acceso al `Subject` que gestiona los mensajes.

Código 4.6: Función getMessageSubject()

```
1  getMessageSubgect(){
2      return this.messageSubject.asObservable();
3  }
```

Este método permite que los componentes del frontend se suscriban y reaccionen a los mensajes en tiempo real.

joinRoom(roomId: string)

Esta función permite a un usuario unirse a una sala específica y comenzar a recibir mensajes enviados a esa sala.

Código 4.7: Función joinRoom(roomId: string)

```
joinRoom(roomId: string) {
    if (this.stompClient) {
      this.roomSubscription = this.stompClient.subscribe(`/topic/\$
{roomId}`, (message: Message) => {
        const messageContent: ChatMessage = JSON.parse(message.body);
        const currentMessages = this.messageSubject.getValue();
        currentMessages.push(messageContent);
        this.messageSubject.next(currentMessages);
      });

      const newUser: ChatMessage = {
        room: roomId,
        content: `\${localStorage.getItem("UserLogged ")} ha entrado en la
            sala`,
        username: 'Server',
        timeStamp: new Date()
      };
      this.stompClient.publish({ destination: `/app/\$
{roomId}`, body: JSON.stringify(newUser) });
    }
  }
```

- **Suscripción a la sala**: Cada sala está asociada a un canal (**/topic/roomId**) al que se suscribe el cliente para recibir mensajes.
- **Mensaje de entrada**: Cuando un usuario se une a una sala, se envía un mensaje de bienvenida indicando su llegada.

4.3.3. Problemas en la Migración

Implementación de DocumentDB sin SSL

Durante la migración a la nube y la adopción de **Amazon DocumentDB** como base de datos para reemplazar MongoDB, surgieron problemas relacionados con la conexión segura mediante SSL. DocumentDB, por defecto, requiere que todas las conexiones sean cifradas utilizando un certificado SSL. Sin embargo, la configuración específica del proyecto presentó un conflicto cuando se intentó integrar DocumentDB con el sistema.

Conflicto con Certificados Autofirmados El sistema utiliza un certificado autofirmado para habilitar HTTPS en el backend. Este enfoque, aunque funcional para entornos de desarrollo y pruebas, generó un conflicto con la verificación estricta de certificados SSL requerida por DocumentDB. Algunos de los problemas observados fueron:

1. **Validación Estricta del Certificado:** DocumentDB requiere que los certificados sean verificados por una autoridad certificadora reconocida. Sin embargo, el uso de un certificado autofirmado provocaba errores al establecer la conexión debido a la imposibilidad de validar su origen. Los intentos de conectarse a DocumentDB con SSL activado resultaban en errores de conexión, como:
   - `SSLHandshakeException`
2. **Impacto en el Despliegue de Producción:** Aunque se intentaron soluciones como importar el certificado autofirmado al almacén de confianza de Java (keyStore), estas resultaron poco prácticas para un entorno dinámico en la nube, donde las imágenes Docker se generan y actualizan con frecuencia.

Solución Implementada: Conexión sin SSL Como medida temporal para superar estos problemas y no retrasar el despliegue, se optó por deshabilitar SSL en la conexión a DocumentDB. La configuración del backend fue ajustada para conectarse al clúster de DocumentDB utilizando un protocolo no cifrado.

Consideraciones de Seguridad Se reconocen las implicaciones de seguridad de esta decisión, ya que deshabilitar SSL expone las comunicaciones a posibles interceptaciones. Para mitigar riesgos, se implementaron las siguientes medidas:

1. **Acceso restringido a la VPC:** DocumentDB y las instancias de la aplicación están alojadas dentro de la misma **VPC** (Virtual Private Cloud), garantizando que las conexiones no sean accesibles desde fuera de la nube de AWS.
2. **Posibilidad de habilitar SSL en el futuro:** Sería posible solucionar los problemas con los certificados autofirmados mediante la integración de un certificado emitido por una autoridad reconocida.

4.4. Pruebas

Se implementaron pruebas funcionales automatizadas para validar la interacción entre dos usuarios en el sistema de chat. Estas pruebas fueron

desarrolladas utilizando **JUnit** y **Selenium WebDriver** para simular usuarios que inician sesión, envían mensajes y verifican que los mensajes son visibles para ambos participantes en tiempo real.

Configuración del Entorno de Pruebas El entorno de desarrollo y pruebas se estableció en **Windows Subsystem for Linux (WSL)**, lo que trajo consigo ciertos desafíos técnicos, como la necesidad de instalar **Google Chrome** y el **ChromeDriver** directamente en el WSL. Este paso fue esencial para garantizar la compatibilidad con el entorno y permitir la ejecución fluida de pruebas automatizadas.

Prueba de Funcionalidad de Chat La clase de pruebas `WebchatAgcApplicationTests` incluye un caso donde dos usuarios interactúan en el sistema de chat en tiempo real. A continuación, se destacan las acciones principales realizadas:

1. **Inicio de Sesión por Dos Usuarios:**
   - Dos instancias de Chrome se inicializan para representar a dos usuarios distintos.
   - Ambos usuarios ingresan sus credenciales y acceden a la interfaz de chat.
2. **Intercambio de Mensajes:**
   - Usuario 1 envía un mensaje que es recibido y visualizado por Usuario 2.
   - Usuario 2 responde con un mensaje que se verifica que sea recibido por Usuario 1.
3. **Verificación de Mensajes:** Se utilizaron aserciones como `assertNotNull` y `assertTrue` para confirmar que los mensajes enviados son visibles en las interfaces correspondientes.

4.5. Distribución y despliegue

El despliegue de la aplicación ha sido un proceso clave para garantizar su funcionalidad tanto en entornos locales como en la nube, permitiendo la adaptación del sistema a diferentes necesidades y escalabilidad. A continuación, se describen las herramientas utilizadas y cómo se llevaron a cabo los despliegues en ambos escenarios.

1. Herramientas Utilizadas El despliegue de la aplicación ha requerido el uso de diversas herramientas y tecnologías para empaquetar, configurar y ejecutar los servicios necesarios:

- **Docker:** Para empaquetar la aplicación y sus dependencias en contenedores estandarizados, facilitando su portabilidad y configuración.
- **Docker Compose:** Para orquestar múltiples servicios, como el backend, la base de datos y otros recursos.
- **Amazon Web Services (AWS):** Para desplegar la infraestructura en la nube, utilizando servicios como EC2, DocumentDB, y Amazon MQ.
- **CLI de AWS:** Herramienta empleada para configurar y gestionar recursos de manera automatizada y precisa en AWS.
- **Load Balancer:** Para distribuir el tráfico entre instancias EC2 y asegurar alta disponibilidad.
- **VPC y Grupos de Seguridad:** Para establecer redes aisladas y configurar las reglas de acceso necesarias para los servicios.

2. Despliegue Local En el entorno local, se utilizó Docker para empaquetar la aplicación y sus servicios asociados. El despliegue local permitió el desarrollo y las pruebas de la funcionalidad de la aplicación en un entorno controlado antes de pasar al entorno en la nube.

- La configuración de los contenedores se gestionó mediante un archivo `docker-compose_local.yml`, que definió la estructura de los servicios, incluyendo el backend y los puertos expuestos.
- Se configuraron certificados y reglas para evitar conflictos relacionados con HTTPS y certificados autofirmados, garantizando que el entorno simulara correctamente las condiciones de producción.

3. Despliegue en la Nube El despliegue en AWS fue significativamente más complejo debido a la necesidad de configurar infraestructura de red, instancias y servicios adicionales. Los principales pasos realizados fueron los siguientes:

1. **Preparación de la Infraestructura:**
   - **VPC (Virtual Private Cloud):** Se creó una red privada virtual para aislar los recursos.

- **Subnets Públicas:** Dos subnets fueron configuradas en diferentes zonas de disponibilidad para mejorar la tolerancia a fallos.
- **Internet Gateway y Tabla de Enrutamiento:** Permitieron la comunicación entre las instancias en la VPC y el exterior.
- **Grupos de Seguridad:** Se definieron reglas de acceso para garantizar que solo el tráfico necesario llegara a los servicios.

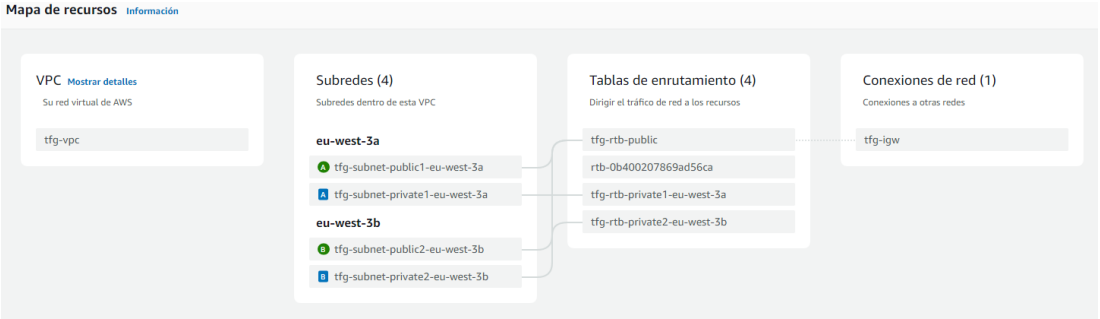

Figura 4.9: Recursos VPC

2. **Servicios de AWS Configurados:**
   - **Instancias EC2:** Dos instancias configuradas con Docker y Docker Compose, utilizadas para ejecutar los contenedores de la aplicación.
   - **Load Balancer:** Configurado para distribuir el tráfico HTTPS entre las instancias EC2.
   - **Amazon DocumentDB:** Base de datos configurada sin SSL para evitar conflictos con certificados autofirmados.
   - **Amazon MQ:** Broker de mensajería configurado para manejar la comunicación en tiempo real.
3. **Configuración de los Servicios:**
   - Los contenedores fueron construidos utilizando una imagen Docker alojada en Docker Hub: `dashshot/webchat_agc:AWScloud`.
   - El archivo `docker-compose_cloud.yml` definió los servicios desplegados en las instancias EC2, ajustando las variables de entorno necesarias para conectar con DocumentDB y Amazon MQ.
   - Se aseguraron las credenciales y claves necesarias mediante configuraciones de entorno, reduciendo los riesgos de exposición de datos sensibles.
4. **Pruebas y Ajustes:**
   - Tras el despliegue, se realizaron pruebas para verificar el correcto funcionamiento del balanceo de carga, la comunicación con el broker de mensajería, y las conexiones a la base de datos DocumentDB.

- Se ajustaron los parámetros de red y acceso en los grupos de seguridad para optimizar el rendimiento y la seguridad.

4. Diferencias entre Despliegue Local y en la Nube

- **Escalabilidad:** El entorno local permite pruebas y desarrollo, mientras que el despliegue en la nube ofrece la posibilidad de escalabilidad y tolerancia a fallos mediante la utilización de subnets y balanceo de carga.
- **Seguridad:** En la nube, se configuraron reglas más estrictas en los grupos de seguridad y VPCs para controlar el tráfico entrante y saliente.
- **Infraestructura:** En el entorno local, la infraestructura es mínima y se limita a contenedores Docker. En AWS, se requiere configurar múltiples servicios interconectados.

Esta estrategia de despliegue asegura que la aplicación pueda ejecutarse de manera eficiente y segura tanto en entornos locales como en la nube, optimizando el desarrollo y facilitando su adopción en producción.

5

Conclusiones y trabajos futuros

La realización de este trabajo ha supuesto un reto importante y un aprendizaje significativo en múltiples aspectos, tanto técnicos como personales. A lo largo del desarrollo del proyecto, se han alcanzado la mayoría de los objetivos planteados inicialmente, lo que demuestra no solo la correcta planificación y ejecución, sino también la capacidad de adaptarse a las dificultades surgidas durante el proceso.

5.1. Reflexión sobre los objetivos cumplidos

Uno de los principales objetivos del proyecto era diseñar y desarrollar una solución completa que integrara múltiples tecnologías, desde el desarrollo del frontend hasta la gestión de servicios en la nube. Este objetivo ha sido cumplido de manera satisfactoria, logrando construir una aplicación funcional y robusta que incluye componentes desarrollados en **Angular**, un backend implementado en **Spring Boot**, y una infraestructura cloud desplegada en **AWS**.

En particular, el trabajo con AWS ha sido un aspecto destacado. Antes de comenzar el proyecto, no tenía experiencia previa con servicios cloud, lo que representaba una barrera inicial. Sin embargo, a lo largo del proyecto, he adquirido conocimientos prácticos sobre una amplia gama de servicios como EC2, Amazon MQ, DocumentDB, y el uso de Load Balancers, entre otros. Esta experiencia no solo ha sido enriquecedora, sino que también ha demostrado ser de gran relevancia, considerando la importancia actual y futura de las soluciones cloud en el desarrollo de software.

Otro de los logros destacados ha sido la integración de los diferentes componentes en un entorno de desarrollo y producción, desde la configuración local mediante Docker hasta el despliegue en un entorno distribuido en la nube. Esto me ha permitido entender las diferencias y los desafíos asociados con ambos contextos, además de aprender a solucionar problemas como la conectividad entre servicios, la seguridad de la infraestructura, y la optimización del rendimiento.

Por último, se han cumplido objetivos en términos de diseño y usabilidad. La aplicación tiene una interfaz limpia y moderna, gracias a las capacidades de Angular, y proporciona una experiencia de usuario coherente y eficiente.

5.2. Aspectos pendientes y futuras ampliaciones

Aunque los objetivos principales se han alcanzado, siempre existe margen para la mejora y la ampliación del proyecto. Algunos de los aspectos que podrían trabajarse en el futuro incluyen:

1. **Escalabilidad y Resiliencia:** Aunque la aplicación utiliza un Load Balancer y subnets distribuidas, sería interesante explorar técnicas avanzadas de escalado automático con Auto Scaling Groups y arquitecturas serverless con AWS Lambda para optimizar aún más la capacidad de respuesta y el uso de recursos.

2. **Seguridad Avanzada:** Aunque se implementaron medidas básicas de seguridad, como la configuración de grupos de seguridad en AWS, se podrían incorporar mejores prácticas adicionales, como la implementación de un sistema IAM más granular o la habilitación de TLS en todos los servicios o el uso de AWS Secret Manager.

3. **Monitorización y Mantenimiento:** Integrar herramientas como Amazon CloudWatch o Prometheus podría ser un paso importante para garantizar la monitorización y el mantenimiento continuo de la aplicación en entornos de producción.

4. **Funcionalidades para el usuario:** El proyecto se ha centrado en la infraestructura necesaria para llevar a cabo los servicios de escalabilidad, dejando una web con funcionalidades limitadas para el usuario, donde se encuentra la posibilidad de un gran abanico de oportunidades para crear funcionalidades nuevas.

5.3. Conclusiones personales

Desde un punto de vista personal, este proyecto ha supuesto un antes y un después en mi desarrollo profesional. He tenido la oportunidad de trabajar con una amplia gama de tecnologías modernas, como **Angular**, **Spring Boot**, **Docker**, y una variedad de servicios de **AWS**. Cada una de estas herramientas me ha ofrecido una perspectiva distinta sobre cómo abordar problemas de desarrollo y cómo crear soluciones completas y escalables.

En particular, mi experiencia con **AWS** ha sido un punto culminante. Antes de comenzar este trabajo, apenas tenía conocimiento sobre la computación en la nube, y ahora entiendo no solo cómo desplegar aplicaciones en este tipo de plataformas, sino también su relevancia estratégica en el panorama actual de la tecnología. En mi opinión, el cloud computing no solo es una herramienta clave en el presente, sino que será aún más importante en el futuro, impulsando la innovación y permitiendo que las aplicaciones alcancen niveles de escalabilidad y disponibilidad sin precedentes.

A nivel personal, este trabajo ha sido un desafío que me ha obligado a salir de mi zona de confort, aprender nuevas tecnologías y encontrar soluciones a problemas complejos. Esta experiencia me ha ayudado a desarrollar habilidades no solo técnicas, sino también de gestión del tiempo y resolución de problemas, que serán de gran utilidad en mi carrera profesional.

En resumen, este proyecto no solo ha cumplido con los objetivos técnicos y académicos, sino que también ha sido una experiencia
profundamente enriquecedora que me ha preparado para enfrentar los retos del mundo profesional con confianza y entusiasmo.

Bibliografía

[1] F. Richter, "Amazon maintains cloud lead as microsoft edges closer," *Statista*, 2024. [Online]. Available: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

[2] Oracle, "Java, lenguaje de programación," http://www.oracle.com/technetwork/java/index.html, 1995.

[3] I. Pivotal Software, "Spring framework," https://spring.io/projects/spring-framework, 2004.

[4] P. S. Inc., "Spring boot," https://spring.io/projects/spring-boot, 2014.

[5] I. Pivotal Software, "Spring messaging," https://docs.spring.io/spring-framework/docs/current/reference/html/messaging.html, 2023.

[6] P. L. Team, "Project lombok," https://projectlombok.org/, 2009.

[7] IETF, "The websocket protocol," https://tools.ietf.org/html/rfc6455, 2011.

[8] S. Community, "Simple text oriented messaging protocol (stomp)," https://stomp.github.io/, 2013.

[9] P. Software, "Rabbitmq," https://www.rabbitmq.com/, 2007.

[10] ——, "Rabbitmq stomp plugin," https://www.rabbitmq.com/docs/stomp/.

[11] A. S. Foundation, "Apache activemq," https://activemq.apache.org/, 2004.

[12] P. Reactor, "Reactor netty," https://projectreactor.io/, 2017.

[13] I. MongoDB, "Mongodb," https://www.mongodb.com/, 2009.

[14] I. Pivotal Software, "Spring security," https://spring.io/projects/spring-security, 2008.

[15] Auth0, "Json web tokens," https://jwt.io/, 2015.

[16] D. Crockford, "Json (javascript object notation)," https://www.json.org/, 2006.

[17] J. Team, "Junit," https://junit.org/, 2002.

[18] S. Project, "Selenium," https://www.selenium.dev/, 2004.

[19] Google, "Angular - the modern web developer's platform," https://angular.io/, 2016.

[20] Microsoft, "Typescript," https://www.typescriptlang.org/, 2012.

[21] W. W. W. C. (W3C), "Html (hypertext markup language)," https://html.spec.whatwg.org/, 1993.

[22] W. W. W. Consortium, "Css (cascading style sheets)," https://www.w3.org/Style/CSS/, 1996.

[23] S. Team, "Sockjs," https://github.com/sockjs/sockjs-client, 2011.

[24] J. Mesnil, “Stompjs,” https://github.com/stomp-js/stompjs, 2013.

[25] H. Technologies, “Haproxy,” http://www.haproxy.org/, 2000.

[26] I. Docker, “Docker - open source container platform,” https://www.docker.com/, 2013.

[27] A. S. Foundation, “Maven - build automation tool,” https://maven.apache.org/, 2004.

[28] Microsoft, “Visual studio code,” https://code.visualstudio.com/, 2015.

[29] ——, “Windows subsystem for linux (wsl),” https://learn.microsoft.com/en-us/windows/wsl/, 2016.

[30] I. Postman, “Postman - api platform for building and using apis,” https://www.postman.com/, 2012.

[31] M. E. Team, “Mongo express - web-based mongodb admin interface,” https://github.com/mongo-express/mongo-express, 2014.

[32] L. Torvalds and G. Contributors, “Git - distributed version control system,” https://git-scm.com/, 2005.

[33] I. GitHub, “Github - where the world builds software,” https://github.com/, 2008.

[34] I. Docker, “Dockerhub - docker container registry,” https://hub.docker.com/, 2013.

[35] I. Amazon Web Services, “Aws (amazon web services),” https://aws.amazon.com/, 2006.

[36] I. MongoDB, “Json and bson - differences and advantages,” https://www.mongodb.com/resources/basics/json-and-bson, 2023.

Apéndices

A

Despliegue en local

A.1. Prerequisitos

Asegúrese de tener Docker instalado en su sistema.

Puede instalar Docker siguiendo las instrucciones del sitio web oficial: Guía de instalación de Docker

A.2. Pasos para el despliegue

1. Descargar archivos de configuración Ejecute los siguientes comandos en su terminal para descargar los archivos necesarios:

```
curl -O https://raw.githubusercontent.com/DashShot/TFG_AITOR/
refs/heads/main/haproxy.cfg
curl -O https://raw.githubusercontent.com/DashShot/TFG_AITOR/
refs/heads/main/fullchain.pem
curl -O https://raw.githubusercontent.com/DashShot/TFG_AITOR/
refs/heads/main/docker-compose_local.yml
```

2. Ejecutar Docker Compose Levante los servicios definidos en el archivo docker-compose_local.yml ejecutando:

```
docker compose -f docker-compose_local.yml up
```

B

Despliegue en AWS

Este documento detalla los pasos necesarios para desplegar la infraestructura en AWS utilizando la CLI de AWS.

B.1. Prerequisitos

1. **Cuenta de AWS**: Asegúrate de tener una cuenta de AWS activa.
2. **AWS CLI**: Instala la AWS CLI. Puedes seguir las instrucciones oficiales de instalación.
3. **Credenciales de AWS**: Configura tus credenciales ejecutando:

```
aws configure
```

4. **Par de claves**: Necesitas un par de claves para conectar a tu instancia EC2. Si no tienes uno, créalo con el siguiente comando:

```
aws ec2 create-key-pair --key-name MyKeyPair --query
    'KeyMaterial' --output text > MyKeyPair.pem
chmod 400 MyKeyPair.pem
```

B.2. Pasos para el despliegue

1. Crear una VPC

Una VPC (Virtual Private Cloud) es una red virtual que te permite lanzar recursos en una red aislada. Para crearla, ejecuta el siguiente comando:

```
aws ec2 create-vpc --cidr-block 10.0.0.0/16 --region eu-west-3
```

Este comando crea una VPC con un rango de direcciones IP de 10.0.0.0/16 en la región `eu-west-3` (París).

2. Crear subnets públicas

Creamos dos subnets públicas, una en la zona de disponibilidad `eu-west-3a` y otra en `eu-west-3b`:

Subnet en `eu-west-3a`:

```
aws ec2 create-subnet --vpc-id <vpc-id> --cidr-block
    10.0.1.0/24 --availability-zone eu-west-3a
```

Subnet en `eu-west-3b`:

```
aws ec2 create-subnet --vpc-id <vpc-id> --cidr-block
    10.0.2.0/24 --availability-zone eu-west-3b
```

3. Crear un Internet Gateway

Un Internet Gateway es necesario para permitir la comunicación entre las instancias en tu VPC y el mundo exterior. Crea uno ejecutando el siguiente comando:

```
aws ec2 create-internet-gateway --region eu-west-3
```

Para adjuntar el Internet Gateway a la VPC, utiliza el siguiente comando:

```
aws ec2 attach-internet-gateway --vpc-id <vpc-id>
    --internet-gateway-id <internet-gateway-id>
```

4. Crear una tabla de enrutamiento para las subredes públicas

Para que las instancias dentro de las subredes públicas puedan acceder a Internet, necesitamos una tabla de enrutamiento asociada con estas subredes. Primero, crea una tabla de enrutamiento:

```
aws ec2 create-route-table --vpc-id <vpc-id> --region eu-west-3
```

Esto devolverá un ID de tabla de enrutamiento. A continuación, crea una ruta para redirigir el tráfico hacia el Internet Gateway:

```
aws ec2 create-route --route-table-id <route-table-id>
    --destination-cidr-block 0.0.0.0/0 --gateway-id
    <internet-gateway-id>
```

Ahora, asocia esta tabla de enrutamiento a las subredes públicas que creaste anteriormente. Ejecuta los siguientes comandos:

Asocia la tabla de enrutamiento con la subnet en `eu-west-3a`:

```
aws ec2 associate-route-table --subnet-id
   <subnet-id-eu-west-3a> --route-table-id <route-table-id>
```

Asocia la tabla de enrutamiento con la subnet en `eu-west-3b`:

```
aws ec2 associate-route-table --subnet-id
   <subnet-id-eu-west-3b> --route-table-id <route-table-id>
```

5. Crear un Security Group

Un Security Group es necesario para controlar el tráfico que entra y sale de las instancias. En este paso, crearemos un Security Group que permita:

- Acceso en el puerto 22 para SSH.
- Acceso en el puerto 443 para HTTPS.
- Acceso en el puerto 61614 para el broker de ActiveMQ.
- Acceso en el puerto 27017 para DocumentDB.

Primero, crea el Security Group:

```
aws ec2 create-security-group --group-name MySecurityGroup
   --description "Security group para tr fico de aplicaci n"
   --vpc-id <vpc-id>
```

Esto devolverá un GroupId. Usa este ID para agregar las reglas de entrada:

```
aws ec2 authorize-security-group-ingress --group-id
   <security-group-id> --protocol tcp --port 22 --cidr
   0.0.0.0/0
aws ec2 authorize-security-group-ingress --group-id
   <security-group-id> --protocol tcp --port 443 --cidr
   0.0.0.0/0
aws ec2 authorize-security-group-ingress --group-id
   <security-group-id> --protocol tcp --port 61614 --cidr
   0.0.0.0/0
aws ec2 authorize-security-group-ingress --group-id
   <security-group-id> --protocol tcp --port 27017 --cidr
   0.0.0.0/0
```

6. Crear dos instancias EC2

Ahora, crearemos dos instancias EC2 en las subredes públicas para que se encuentren disponibles y puedan ser agregadas al Load Balancer.

Ejecuta el siguiente comando para lanzar la primera instancia en la subnet `eu-west-3a`:

```
aws ec2 run-instances --image-id <ami-id> --count 1
   --instance-type t2.micro --key-name MyKeyPair --subnet-id
   <subnet-id-eu-west-3a> --associate-public-ip-address
   --security-group-ids <security-group-id>
   --tag-specifications
   'ResourceType=instance,Tags=[{Key=Name,Value=Instancia1}]'
```

Lanza la segunda instancia en la subnet `eu-west-3b`:

```
aws ec2 run-instances --image-id <ami-id> --count 1
   --instance-type t2.micro --key-name MyKeyPair --subnet-id
   <subnet-id-eu-west-3b> --associate-public-ip-address
   --security-group-ids <security-group-id>
   --tag-specifications
   'ResourceType=instance,Tags=[{Key=Name,Value=Instancia1}]'
```

Reemplaza `<ami-id>` con el ID de la AMI que desees usar, `<subnet-id-eu-west-3a>` y `<subnet-id-eu-west-3b>` con los IDs de las subredes, y `<security-group-id>` con el ID del grupo de seguridad creado anteriormente.

7. Crear un Target Group

Ahora, crearemos un Target Group para agregar las instancias. Este Target Group manejará el tráfico que será distribuido por el Load Balancer.

```
aws elbv2 create-target-group --name MyTargetGroup --protocol
   HTTPS --port 443 --vpc-id <vpc-id> --target-type instance
```

Esto creará un Target Group llamado `MyTargetGroup` que escuchará en el puerto 443.

8. Registrar las instancias en el Target Group

Para asociar las instancias EC2 al Target Group creado, primero obtén los IDs de las instancias que lanzaste previamente y regístralas en el Target Group.

```
aws elbv2 register-targets --target-group-arn
   <target-group-arn> --targets Id=<instance-id-1>
   Id=<instance-id-2>
```

Reemplaza `<target-group-arn>` con el ARN del Target Group y `<instance-id-1>` y `<instance-id-2>` con los IDs de las instancias EC2.

9. Crear un Load Balancer

Crearemos el Load Balancer para distribuir el tráfico entre las instancias. Crea el Load Balancer de la siguiente manera:

```
aws elbv2 create-load-balancer --name MyLoadBalancer --subnets
   <subnet-id-eu-west-3a> <subnet-id-eu-west-3b>
   --security-groups <security-group-id> --scheme
   internet-facing --load-balancer-type application --region
   eu-west-3
```

10. Crear un Listener para el Load Balancer

Finalmente, crea un listener que escuche en el puerto 443 y redirija el tráfico hacia el Target Group:

```
aws elbv2 create-listener --load-balancer-arn
   <load-balancer-arn> --protocol HTTPS --port 443
   --default-actions
   Type=forward,TargetGroupArn=<target-group-arn>
```

Reemplaza `<load-balancer-arn>` con el ARN del Load Balancer y `<target-group-arn>` con el ARN del Target Group.

11. Crear un clúster de DocumentDB

En este paso, configuraremos un clúster de Amazon DocumentDB con las siguientes características:

1. Una instancia en el clúster.
2. TLS desactivado.
3. Asociado a las subredes creadas anteriormente.

12. Crear un grupo de subredes para DocumentDB

Antes de crear el clúster, necesitamos un grupo de subredes que asocie las subnets públicas creadas anteriormente:

```
aws docdb create-db-subnet-group \
    --db-subnet-group-name MyDocDBSubnetGroup \
    --db-subnet-group-description "Subnets para DocumentDB" \
    --subnet-ids <subnet-id-eu-west-3a> <subnet-id-eu-west-3b>
```

Reemplaza `<subnet-id-eu-west-3a>` y `<subnet-id-eu-west-3b>` con los IDs de las subnets creadas previamente.

13. Crear el clúster DocumentDB

Ahora, crea el clúster:

```
aws docdb create-db-cluster \
    --db-cluster-identifier MyDocumentDBCluster \
    --engine docdb \
    --master-username <usernameDB> \
    --master-user-password <passwordDB> \
    --db-subnet-group-name MyDocDBSubnetGroup \
    --vpc-security-group-ids <security-group-id> \
    --port 27017 \
    --enable-tls false
```

14. Crear una instancia en el clúster

Finalmente, agrega una instancia al clúster:

```
aws docdb create-db-instance \
    --db-instance-identifier MyDocDBInstance \
    --db-cluster-identifier MyDocumentDBCluster \
    --engine docdb \
    --db-instance-class db.t3.medium
```

15. Crear un broker de ActiveMQ

En este paso, configuraremos un broker de ActiveMQ utilizando Amazon MQ. Este broker será accesible desde las instancias EC2 y estará asociado al Security Group configurado previamente para permitir el acceso en el puerto 61614.

16. Crear un broker

Ejecuta el siguiente comando para crear un broker de ActiveMQ:

```
aws mq create-broker \
    --broker-name MyActiveMQBroker \
    --engine-type ActiveMQ \
    --engine-version 5.18.4 \
    --deployment-mode SINGLE_INSTANCE \
    --broker-instance-class mq.t3.micro \
    --publicly-accessible true \
    --security-groups <security-group-id> \
    --subnet-ids <subnet-id-eu-west-3a> <subnet-id-eu-west-3b>
       \
    --authentication-strategy SIMPLE \
```

```
    --users
       '[{"username":"<usernameMQ>","password":"<passwordMQ>"}]'
```

17. Conectar a las instancias EC2

Primero, obtén la dirección IP pública de ambas instancias creadas anteriormente. Conéctate a cada una utilizando el comando SSH:

```
ssh -i MyKeyPair.pem ec2-user@<public-ip-address>
```

18. Instalar Docker

Ejecuta el siguiente comando en ambas instancias para instalar Docker:

```
curl -sSL https://get.docker.com | sudo sh
```

19. Descargar y configurar Docker Compose

Descarga el archivo `docker-compose_cloud.yml` desde el repositorio de GitHub:

```
curl -O https://raw.githubusercontent.com/DashShot/TFG_AITOR/
refs/heads/main/docker-compose_cloud.yml
```

Edita el archivo descargado para ajustar las variables de entorno necesarias. Utiliza un editor de texto como `vim`:

```
vim docker-compose_cloud.yml
```

Guarda y cierra el archivo.

20. Iniciar el despliegue con Docker Compose

Ejecuta el siguiente comando para iniciar los contenedores:

```
docker compose -f docker-compose_cloud.yml up -d
```

C

Repositorio GitHub

C.1. Enlace al repositorio

El proyecto ha sido alojado en la plataforma Github.

El enlace es el siguiente: `https://github.com/DashShot/TFG_AITOR`