Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo y Despliegue de SEHHA, una Plataforma de Teleconsulta Médica Basada en Microservicios y WebRTC en EKS

Autor: Soulaiman El Maimouni Mechbal

Tutor: María Teresa Ariza Gómez

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2024

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo y Despliegue de SEHHA, una Plataforma de Teleconsulta Médica Basada en Microservicios y WebRTC en EKS

Autor:
Soulaiman El Maimouni Mechbal

Tutor:
María Teresa Ariza Gómez
Profesor titular

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Trabajo Fin de Grado: Desarrollo y Despliegue de SEHHA, una Plataforma de Teleconsulta Médica Basada en Microservicios y WebRTC en EKS

Autor: Soulaiman El Maimuni Mechbal

Tutor: María Teresa Ariza Gómez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

A mis padres por enseñarme el amor verdadero y a luchar

A mis hermanos por estar siempre a mi lado

Agradecimientos

La realización de este Trabajo de Fin de Grado (TFG) ha sido posible gracias al apoyo y la colaboración de muchas personas e instituciones, a quienes me gustaría expresar mi más sincero agradecimiento.

En primer lugar, quiero agradecer profundamente a mi familia. A mi madre Naima y a mi padre Abdellah, por enseñarme el valor de la lucha y el sacrificio. Sus incansables esfuerzos y su amor incondicional han sido mi mayor fuente de inspiración y fortaleza. A mis hermanos Sufian, Mohammad Reda y Oussama, por su constante apoyo y aliento. Sus palabras de ánimo y su presencia han sido esenciales para mantenerme motivado y enfocado durante este arduo proceso.

Quiero expresar mi gratitud a mi tutora, María Teresa Ariza Gómez, por su constante guía, apoyo y orientación a lo largo de todo el proceso. Su experiencia y conocimientos han sido fundamentales para la correcta realización de este trabajo. Su paciencia, dedicación y disposición para ayudarme en cada etapa del proyecto me han inspirado a superar los desafíos y a seguir adelante con confianza.

Agradezco también a la Universidad de Sevilla, y en particular a la Escuela Técnica Superior de Ingeniería, por proporcionarme los recursos y el entorno académico necesarios para llevar a cabo este proyecto. El acceso a las instalaciones, bibliotecas y herramientas tecnológicas ha sido crucial para el desarrollo del TFG.

Finalmente, quiero reconocer a todos los profesionales y expertos cuyos trabajos y publicaciones han servido como base y referencia para la elaboración de este TFG. Su contribución al conocimiento en el campo de la telemedicina y la ingeniería ha sido invaluable para mi investigación y comprensión del tema.

A todos, muchas gracias.

Soulaiman El Maimouni Mechbal

Sevilla, 2024

Resumen

El presente Trabajo de Fin de Grado (TFG) tiene como objetivo diseñar e implementar una aplicación de telemedicina basada en una arquitectura de microservicios. Esta solución busca proporcionar una plataforma eficiente y accesible que permita a pacientes y profesionales de la salud realizar consultas médicas de manera remota. La arquitectura del sistema se fundamenta en el uso de AWS como proveedor de servicios en la nube, Kubernetes para la orquestación y balanceo de carga, y Docker para el empaquetamiento de los servicios. El *frontend* está desarrollado con React, utilizando Axios para las solicitudes HTTP y Socket.io para la comunicación en tiempo real con un servicio de señalización para WebRTC. Los microservicios backend, implementados en Node.js, incluyen servicios de gestión de usuarios, gestión de citas y señalización de vídeo, cada uno con su propia base de datos independiente en MongoDB Atlas. La comunicación asíncrona entre microservicios se maneja mediante RabbitMQ, asegurando una entrega de mensajes fiable entre los servicios. La seguridad del sistema se garantiza mediante JWT para la autenticación y HTTPS para la transmisión segura de datos. La escalabilidad y resiliencia se logran a través de las capacidades de Kubernetes. Este TFG no solo aborda la creciente demanda de soluciones de teleconsulta, sino que también implementa mejores prácticas en desarrollo y despliegue de software, proporcionando una base sólida para futuras expansiones y mejoras.

Abstract

The objective of this thesis is to design and implement a telemedicine application based on a microservices architecture. This solution aims to provide an efficient and accessible platform that allows patients and healthcare professionals to conduct medical consultations remotely. The system architecture is based on the use of AWS as the cloud service provider, Kubernetes for orchestration and load balancing, and Docker for service packaging. The frontend is developed with React, using Axios for HTTP requests and Socket.io for real-time communication with a signaling service for WebRTC. The backend microservices, implemented in Node.js, include user management, appointment management, and video signaling services, each with its own independent database in MongoDB Atlas. Asynchronous communication between microservices is managed via RabbitMQ, ensuring reliable message delivery between services. System security is guaranteed through JWT for authentication and HTTPS for secure data transmission. Scalability and resilience are achieved through Kubernetes capabilities. This TFG not only addresses the growing demand for teleconsultation solutions but also implements best practices in software development and deployment, providing a solid foundation for future expansions and improvements.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xiv**

Índice de Figuras **xx**

1 Introducción **25**

1.1 Contexto y objetivos del TFG *25*

1.1.1 Contexto 25

1.1.2 Objetivos del TFG 27

1.1.3 Alcance del Proyecto 28

1.2 Introducción a la Arquitectura del Sistema *28*

1.2.1 Microservicios Principales 28

1.2.2 Componentes y Tecnologías del Sistema 29

1.2.3 Seguridad y Escalabilidad 29

1.3 Estructura de la memoria *29*

2 Tecnologías Usadas **31**
2.1. Microservicios y su importancia en el desarrollo de software *31*
2.1.1 Introducción a los Microservicios 31
2.1.2 Evolución de la Arquitectura de Software: 32
2.1.3 Beneficios de los Microservicios 32
2.1.4 Desafíos y Consideraciones 33
2.1.5 Casos de Uso y Ejemplos Reales 35
2.2 Tecnologías usadas *35*
2.2.1 React 36
2.2.2 Bootstrap 36
2.2.3 Node.js 36
2.2.4 Express.js 37
2.2.5 MongoDB 37
2.2.6 JSON Web Token (JWT) 38
2.2.7 RabbitMQ 38
2.2.8 Socket.io 38
2.2.9 WebRTC 39
2.2.10 Docker 39
2.2.11 Kubernetes 39
2.2.12 Amazon Web Services (AWS) 40
2.2.13 eksctl 41
2.2.14 GitHub Actions 41
2.2.15 Docker Hub 41
2.2.16 GitHub 42
2.2.17 Git Bash 42
2.2.18 Visual Studio Code 42
2.2.19 Postman 43
2.2.20 Navegadores (Chrome y Firefox) 43

3 Descripción del Sistema **45**
3.1. Visión general de la aplicación *45*
3.1.1 Introducción a la aplicación 45
3.1.2 Funcionalidades Principales de SEHHA 46
3.2 Público objetivo y plataformas *47*
3.2.1 Público Objetivo de SEHHA 47
3.2.2 Plataformas 47
3.3 Arquitectura del sistema: *48*
3.3.1 Introducción a la Arquitectura del Sistema 48
3.3.2 Componentes Principales de la Arquitectura 49
3.3.3 Diagrama de Arquitectura 50

4 Microservicio de Autenticación y Gestión de Usuarios **53**
4.1 Introducción al Microservicio *53*
4.1.1 Descripción General: 53
4.2 Arquitectura del Microservicio *54*
4.2.1 Diagrama de Arquitectura 54
4.2.2 Componentes Internos 54
4.3 Implementación *55*
4.3.1 Lenguajes y Tecnologías Utilizadas 55
4.3.2 Endpoints y APIs 55
4.3.3 Bases de Datos y Almacenamiento 55
4.3.4 Controladores 56
4.4 Comunicación con Otros Microservicios *57*
4.4.1 Métodos de Comunicación 57

4.4.2 Flujos de Datos 57
4.5 Seguridad *57*
4.5.1 Mecanismos de Autenticación y Autorización 57
4.5.2 Protección de Datos 57
4.6 Escalabilidad y Tolerancia a Fallos *58*
4.7 Pruebas *58*
4.7.1 Registro de Nuevo Usuario 59
4.7.2 Inicio de Sesión de Usuario 61
4.8 Conclusión *62*

5 Microservicio de gestión de citas **63**
5.1 Introducción al Microservicio *63*
5.1.1 Descripción General 63
5.2 Arquitectura del Microservicio *64*
5.2.1 Introducción a la Arquitectura CLEAN 64
5.2.2 Componentes de la Arquitectura CLEAN 64
5.2.3 Principios de la Arquitectura CLEAN 64
5.2.4 Componentes Internos 65
5.3 Implementación *67*
5.3.1 Lenguajes y Tecnologías Utilizadas 67
5.3.2 Endpoints y APIs 67
5.3.3 Bases de Datos y Almacenamiento 67
5.4 Comunicación con Otros Microservicios *69*
5.4.1 Métodos de Comunicación 69
5.4.2 Flujos de Datos 69
5.5 Seguridad *70*
5.5.1 Mecanismos de Autenticación y Autorización 70
5.5.2 Protección de Datos 70
5.6 Escalabilidad y Tolerancia a Fallos *70*
5.7 Pruebas *70*
5.7.1 Autenticación de usuario y especialista 71
5.7.2 Creación de bloques de tiempo y reserva de citas 71
5.8 Conclusión *73*

6 Visión General de WebRTC **74**
6.1 Introducción a WebRTC *74*
6.2 APIs de WebRTC *74*
6.3 Protocolos de WebRTC *74*
6.4 Fases de la Comunicación en WebRTC *74*
6.5 Topologías de Red en WebRTC *75*
6.6 Control de Congestión *76*
6.7 Casos de Uso de WebRTC *76*
6.8 Seguridad en WebRTC *76*
6.9 Conclusión *76*

7 Microservicio de señalización de vídeo **77**
7.1 Introducción al Microservicio *77*
7.1.1 Descripción General 77
7.2 Arquitectura del Microservicio *77*
7.2.1 Diagrama de Arquitectura 77
7.3 Componentes Internos *78*
7.3.1 Server 78
7.3.2 Controladores 78
7.3.3 Modelos de Datos 78
7.3.4 Utilidades 79
7.4 Implementación *79*

7.4.1 Lenguajes y Tecnologías Utilizadas 79
7.4.2 Flujos de Datos y Comunicación 79
7.5 Mensajería y Secuencia de Eventos *79*
7.5.1 Mensajería de WebSocket 79
7.5.2 Secuencia de Eventos 80
7.6 Seguridad *81*
7.6.1 Mecanismos de Autenticación y Autorización 81
7.6.2 Protección de Datos 82
7.7 Despliegue y Escalabilidad *82*
7.7.1 Despliegue en EKS 82
7.7.2 Balanceo de Carga y Alta Disponibilidad 82
7.8 Pruebas *82*
7.8.1 Conexión de los Clientes 82
7.8.2 Mensajes “ready” y Creación/Unión de Sala 82
7.8.3 Intercambio de Mensajes `offer` y los candidatos del usuario 83
7.8.4 Intercambio de Mensajes `answer` y los candidatos del especialista 84
7.8.5 Finalización de la Llamada con Mensaje `bye` 84
7.9 Conclusión *85*

8 Frontend de SEHHA y su interacción con los microservicios 86
8.1 Introducción al Frontend *86*
8.1.1 Descripción General 86
8.2 Arquitectura del Frontend *86*
8.2.1 Estructura del Proyecto 86
8.2.2 Componentes Principales 87
8.2.3 Gestión de Estado 87
8.3 Implementación *87*
8.3.1 Lenguajes y Tecnologías Utilizadas 87
8.3.2 Estilos y Responsividad 87
8.3.3 Rutas y Navegación 87
8.4 Integración con el Backend *89*
8.4.1 Comunicación con Microservicios 89
8.4.2 Autenticación y Autorización 89
8.5 Páginas y Funcionalidades *91*
8.5.1 Página de Inicio 91
8.5.2 Páginas de Login y Signup 91
8.5.3 Gestión de Citas para Usuarios 93
8.5.4 Gestión de Calendarios para Especialistas 96
8.5.5 Videollamadas 98
8.6 Implementación de WebRTC *99*
8.6.1 Configuración de ICE Servers 99
8.6.2 Creación y Manejo de Conexiones Peer-to-Peer 100
8.6.3 Manejo de Mensajes de Señalización 101
8.6.4 Interacción del Usuario: Los botones de la interfaz 103
8.7 Seguridad *104*
8.7.1 Autenticación y Autorización 104
8.7.2 Protección de Datos 105
8.8 Despliegue *105*
8.8.1 Dockerización 105
8.8.2 Despliegue en Kubernetes 105
8.9 Conclusión *105*

9 Despliegue de la Aplicación SEHHA en AWS 106
9.1 Arquitectura General *106*
9.1.1 Componentes Principales del Despliegue 106

9.2 Repositorios y Automatización con GitHub Actions *107*
9.2.1 Repositorios de GitHub: 107
9.2.2 Workflows de GitHub Actions 107
9.2.3 Dockerfile para el Frontend: 108
9.2.4 Configuración de Nginx: 109
9.3 Configuración de la Infraestructura *110*
9.3.1 Creación del Clúster con eksctl 110
9.3.2 Despliegue de Recursos con kubectl 111
9.4 Configuración de DNS y Certificados con Route 53 y ACM *114*
9.4.1 Compra del Dominio y Configuración de DNS 114
9.5 Gestión de Datos *116*
9.5.1 Arquitectura de la Base de Datos 116
9.5.2 Conexión y Seguridad de las Bases de Datos 118
9.5.3 Uso de RabbitMQ en CloudAMQP 118
9.6 Seguridad *119*
9.6.1 Certificados SSL/TLS 119
9.7 Pruebas *119*
9.8 Conclusión *120*

10 Conclusiones y Líneas Futuras **122**
10.1 Evaluación del Cumplimiento de Objetivos *122*
10.1.1 Objetivo General 122
10.1.2 Objetivos Específicos 122
10.2 Líneas Futuras *123*
10.2.1 Mejora de la Interfaz de Usuario 123
10.2.2 Integración con Otros Sistemas de Salud 123
10.2.3 Escalabilidad Global 123
10.2.4 Funcionalidades Avanzadas de Telemedicina 123
10.2.5 Seguridad y Cumplimiento 123
10.2.6 Optimización del Rendimiento 123
10.2.7 Ampliación del Soporte Multiplataforma 123
10.2.8 Extensión del Alcance 124

Referencias **125**

ÍNDICE DE FIGURAS

Figura 2-1. Diagrama de microservicios 31
Figura 2-2. Amazon logo 35
Figura 2-3. Logo de Netflix 35
Figura 2-4. Logo de React 36
Figura 2-5. Logo de Bootstrap 36
Figura 2-7. Logo de Node.js 36
Figura 2-8. Logo de Express.js 37
Figura 2-9. Logo de MongoDB 37
Figura 2-10. Logo de JWT 38
Figura 2-11. Logo de RabbitMQ 38
Figura 2-12. Logo de Socket.io 38
Figura 2-12. Logo de WebRTC 39
Figura 2-13. Logo de Docker 39
Figura 2-14. Logo de Kubernetes 39
Figura 2-15. Logo de AWS 40
Figura 2-16. Logo de Amazon EKS 40
Figura 2-17. Logo de AWS Certificate Manager 41
Figura 2-18. Logo de AWS Route53 41
Figura 2-19. Logo de Github 42
Figura 2-20. Logo de Git Bash 42
Figura 2-21. Logo de Visual Studio Code 42
Figura 2-22. Logo de Postman 43
Figura 3-1. Logo de SEHHA 45

Figura 3-2. Colores de marca de SEHHA 45
Figura 3-3. Arquitectura de SEHHA 50
Figura 4-1. Arquitectura del microservicio de gestión de usuarios 54
Figure-4-2. Registro de nuevo usuario 59
Figure-4-3. Usuario creado en la base de datos 59
Figure-4-4. Usuario creado en la base de datos de citas 60
Figure-4-5. Error: Correo ya está en uso 60
Figure-4-6. Error: Correo ya está en uso 61
Figure-4-7. Autenticación exitosa de un usuario 61
Figure-4-8. Autenticación no exitosa del usuario, correo incorrecto 62
Figure-4-9. Autenticación no exitosa del usuario. Contraseña incorrecta. 62
Figure 5-1. Arquitectura CLEAN por Robert C. Martin 64
Figura 5-2. Autenticación del usuario y obtención del token JWT 71
Figura 5-3. Autenticación del especialista y obtención del token JWT 71
Figure 5-4. Bearer token generado por el especialista enviado en una cabecera Authorization 72
Figure 5-5. Creación exitosa del bloque de tiempo 72
Figure 5-6. Error de autorización si no se envía el token 72
Figure 5-7. Cita reservada correctamente 73
Figure 5-8. Citas reservadas por parte del usuario 73
Figure 6-1. Topologías de red en WebRTC 75
Figura 7-1. Arquitectura del microservicio de señalización de vídeo 77
Figura 7-2. Diagrama de Secuencia de Mensajes de Señalización WebRTC 80
Figura 7-3. Inicio del servidor y de las conexiones a sockets 82
Figura 7-4. Logs del inicio de la llamada 83
Figura 7-5. Estado de la base de datos tras el inicio de la llamada por parte del usuario 83
Figura 7-6. Estado de la base de datos tras el inicio de la llamada por parte del especialista 83
Figura 7-7. Logs del envío del mensaje de oferta y los ICE candidates por parte del usuario 84
Figura 7-8. Logs del envío del mensaje de respuesta y los ICE candidates por parte del especialista 84
Figura 7-9. Logs del envío del intercambio de mensajes de tipo “bye” para cerrar la sesión 85
Figura 7-10. Estado de la base de datos tras el cierre de sesión 85
Figura 8-1. App.jsx, componente principal del frontend de SEHHA 88
Figure 8-2. Componente de gestión de contexto AuthContextProvider 89
Figura 8-3. Index.jsx, el componente principal App dentro de AuthContextProvider 89
Figura 8-4. Hooks useLogin, useSignup y useLogout 90
Figura 8-5. Página de inicio 91
Figura 8-6. Selección del tipo de usuario para el registro 92
Figura 8-7. Selección del tipo de usuario para la autenticación 92
Figura 8-8. Signup de usuarios 92
Figura 8-9. Login de usuarios 93

Figura 8-10. Signup de especialistas 93
Figura 8-11. Login de especialistas 93
Figura 8-12. Pantalla de selección de citas para usuarios 94
Figura 8-13. Ventana de confirmación de la cita 95
Figura 8-14. Notificación de cita reservada 95
Figura 8-15. Visualización de citas reservadas para usuarios 96
Figura 8-16. Pantalla de gestión de horarios para especialistas 97
Figura 8-17. Visualización de citas reservadas para especialistas 98
Figura 8-19. Página de videollamadas para especialistas 99
Figura 8-20. Configuración de ICE Servers 100
Figura 8-21. Función para iniciar una llamada 100
Figura 8-22. Manejo de mensajes de señalización 101
Figura 8-23. Manejo de ofertas entrantes 102
Figura 8-24. Manejo de respuestas y candidatos ICE 102
Figura 8-25. Captura y transmisión de MediaStreams 103
Figura 8-26. Finalización de la llamada 103
Figura 8-27. Interfaz de usuario para la gestión de videollamadas 103
Figura 8-28. Inicialización de la videollamada por parte del especialista 104
Figura 8-29. Inicialización de la videollamada por parte del usuario 104
Figure 9-1. Workflow de Github Actions para automatizar la contenerización del frontend de SEHHA 107
Figure 9-2. Dokerfile del frontend de SEHHA 108
Figure 9-3. Configuración del servidor Nginx para servir el frontend de SEHHA 109
Figura 9-4. Commit a repositorio remoto souel22/tfg-sehha-front en la rama feature/basic-frontend 109
Figura 9-5. Ejecución del workflow en GitHub Actions. 110
Figura 9-6. Contenedores en Docker Hub después del push y la ejecución de los workflows. 110
Figura 9-7. Ejecución del comando para crear el clúster EKS. 111
Figura 9-8. Despliegue de los recursos en EKS 113
Figura 9-9. Recursos en el namepace sehha-app desplegados en EKS 114
Figura 9-10. SEHHA accesible a través del balanceador de cargas 114
Figura 9-11. Configuración de registros DNS en Route 53. 115
Figura 9-12. Solicitud y validación de certificado en ACM. 116
Figura 9-13. Uso del ARN del certificado en la configuración del Load Balancer. 116
Figura 9-14. Cluster “sehha-appointement-cluster” en el que se alojan las bases de datos de SEHHA 117
Figura 9-15. Estructura de Clúster en MongoDB Atlas mostrando las bases de datos independientes para cada microservicio. 117
Figura 9-16. Instancia “sehha-mq” en la nube para alojar las colas de RabbitMQ de SEHHA 118
Figura 9-17. Colas de RabbitMQ de SEHHA en RabbitMQ Manager 118
Figura 9-18. Certificado SSL/TLS activo y verificado en el navegador, indicando una conexión segura. 119
Figura 9-19. Inicio de llamada por parte de un especialista en un dispositivo de tipo Tablet 120

Figura 9-20. Inicio de llamada por parte de un usuario en un dispositivo de tipo Smartphone 120

1 INTRODUCCIÓN

El auge de la telemedicina ha demostrado ser una respuesta efectiva y necesaria para enfrentar los desafíos contemporáneos en el acceso a servicios de salud. La capacidad de realizar consultas médicas de manera remota no solo ha reducido el riesgo de contagio durante la pandemia de COVID-19, sino que también ha mejorado el acceso a atención médica en áreas rurales y para personas con movilidad reducida. Este proyecto de Fin de Grado (TFG) se centra en el diseño e implementación de una aplicación de teleconsulta basada en una arquitectura de microservicios, lo que permite una mayor flexibilidad, escalabilidad y resiliencia en la gestión de los servicios de salud.

1.1 Contexto y objetivos del TFG

1.1.1 Contexto

La necesidad de una solución eficiente y accesible para la teleconsulta médica ha aumentado significativamente, especialmente en los últimos años. La pandemia de COVID-19 destacó la importancia de poder acceder a servicios de salud de manera remota para reducir el riesgo de contagio y para ofrecer servicios médicos en áreas con acceso limitado a centros de salud. Las teleconsultas permiten a los pacientes recibir atención médica sin tener que desplazarse, lo que es crucial para aquellos con movilidad reducida o en zonas rurales.

1.1.1.1 Descripción General del Contexto:

La telemedicina tuvo sus comienzos en Canadá y EE. UU. en la década de los 1950, apareciendo por primeras veces en la literatura médica y en las visitas de telemedicina entre estudiantes de medicina y profesores en la universidad de Nebraska a finales de dicha década. Durante la década de 1960, se registraron los primeros casos de telemedicina asíncrona o telemedicina de almacenamiento y reenvío mediante la transmisión de electrocardiogramas o de pruebas de imagen, y de la transmisión en vivo en zonas rurales y urbanas. Unos de los pioneros fueron el Massachusetts General Hospital (MGH) y el Nebraska Psychiatric Institute (NPI).

En la década de los 1970, y gracias a la participación de la National Aeronautics and Space Administration (NASA) y el Department of Health and Human Services (DHHS), se lanzaron programas para conectar a pacientes en áreas remotas con proveedores de atención hospitalaria.

En los 80 empezó a expandirse la telemedicina a nivel global, con especial aceleración a finales de la década por los avances tecnológicos que abarataron los costes de los equipamientos.

En la década de 1990, la telemedicina experimentó un gran avance con el surgimiento de la World Wide Web e Internet. Se crearon organizaciones importantes como la American Telemedicine Association (ATA) y la Office for the Advancement of Telehealth (OAT) del DHHS. Se establecieron varios programas de telemedicina, incluyendo programas de tele-radiología, servicios de telesalud directos al consumidor, telesalud en prisiones, telepsiquiatría, etc.

En el nuevo milenio, los gobiernos estatales y juntas médicas en EE. UU. empezaron a desarrollar sus propias políticas de telemedicina, empezando por la Kentucky Telehealth Network en 2005, seis Centros Regionales de Telemedicina en 2006 y el American Recovery and Reinvestment Act (ARRA) de 2009 destinado a promover transacciones seguras y eficientes hacia registros médicos electrónicos.

En la década de 2010, la ley del Affordable Care Act (ACA) ha continuado con la asignación de fondos federales a programas y servicios de telemedicina con el objetivo de conseguir el triple objetivo de mejorar la salud de la población, mejorar la experiencia de la atención del paciente y reducir el costo per cápita de la atención. [2]

La subutilización de la telemedicina terminó con la llegada de la pandemia de COVID-19, que llevó a las personas a aislarse y evitar buscar servicios de salud en hospitales y clínicas locales. Los proveedores de salud implementaron rápidamente servicios de telemedicina para satisfacer las necesidades de sus pacientes. Las visitas de telemedicina aumentaron 10 veces en comparación con los niveles previos a la pandemia en algunos sistemas de salud. A pesar del aumento inicial en la utilización de la telemedicina durante la pandemia, las tendencias recientes sugieren un declive en la utilización de estos servicios. Como causas de este declive podemos destacar:

- Retorno a la Normalidad: La utilización de la telemedicina disminuyó rápidamente a medida que las personas volvían a estilos de vida anteriores a la pandemia.
- Visitas Presenciales: Las visitas de atención médica regresaron a clínicas y hospitales, con trabajadores de la salud agotados y fatigados tratando de ponerse al día con exámenes de detección perdidos y citas atrasadas.
- Relegación de la Tecnología: En lugar de continuar con la tecnología de telemedicina para ahorrar en recursos de atención médica, la tecnología fue nuevamente relegada a un segundo plano [3].

En este periodo también se espera que el auge del mercado de la telemedicina en tiempo real basada en aplicaciones móviles se traduzca en que el mercado de la telemedicina alcance casi los 112 mil millones de dólares en el año 2025.

Se espera que, en las próximas décadas, la telemedicina se convierta en una norma social completamente integrada con otros elementos de la salud, ofreciendo diversas opciones para la entrega de atención médica y garantizando el acceso efectivo y eficiente sin importar la ubicación del paciente. La tecnología seguirá siendo fundamental en la evolución y expansión de la telemedicina. [2]

1.1.1.2 Estado Actual de la Tecnología y su Relevancia:

Los microservicios representan una arquitectura de software en la que una aplicación se divide en pequeños servicios independientes que se ejecutan en su propio proceso y se comunican entre sí mediante interfaces bien definidas. Cada servicio se enfoca en una funcionalidad de negocio específica y puede ser desarrollado, desplegado y escalado de manera independiente. Esta arquitectura permite una mayor flexibilidad y escalabilidad, esencial en aplicaciones de telemedicina que deben gestionar múltiples servicios como autenticación de usuarios, gestión de citas, videollamadas y almacenamiento de registros médicos. Algunos de los ejemplos de Proyectos y Soluciones Similares son los siguientes:

- **Portal Telemedicina**: Portal Telemedicina utiliza una arquitectura de microservicios basada en la nube para gestionar registros médicos y proporcionar servicios de telemedicina. La adopción de esta arquitectura ha permitido a la empresa escalar sus operaciones y gestionar eficientemente los datos de millones de pacientes. [4]
- **Amwell**: Amwell utiliza una arquitectura de microservicios serverless y multinube para gestionar sus servicios de teleconsulta. Esto incluye la capacidad de adaptarse a cualquier escala de procesamiento necesaria para abordar los volúmenes de visitas [5]
- **Microservice Chatbot Architecture for Chronic Patient Support**: Este proyecto presenta una arquitectura de microservicios para un chatbot diseñado para apoyar a pacientes crónicos, facilitando la comunicación y el seguimiento de su estado de salud.

Los microservicios tienen como beneficios la escalabilidad, permite escalar partes específicas del sistema según la demanda, como las videollamadas durante picos de consultas. El despliegue continuo, Facilitando la implementación de actualizaciones y nuevas características sin interrumpir otros servicios y resiliencia debido al aumento de la tolerancia a fallos, ya que un problema en un microservicio no afecta necesariamente al resto del sistema.

Por otro lado, los microservicios requieren una infraestructura robusta para gestionar y orquestar los servicios, así como para monitorear su rendimiento y mantener la coherencia de los datos a través de múltiples

microservicios, lo que puede ser un reto.

Docker y Kubernetes son ampliamente utilizados para desplegar y gestionar aplicaciones de telemedicina en la nube. Permiten una implementación rápida y escalable de servicios de salud, lo que es crucial para manejar grandes volúmenes de datos y usuarios. Esto es útil a su vez para asegurar la escalabilidad y la resiliencia a fallos de los microservicios [6][7]

Para la comunicación entre distintos usuarios, se puede utilizar WebRTC. WebRTC facilita la comunicación en tiempo real mediante navegadores web sin necesidad de plugins adicionales, lo que lo hace ideal para teleconsultas y consultas de emergencias. Esta tecnología es esencial para proporcionar una experiencia de usuario fluida y efectiva en telemedicina. [8][9]

1.1.2 Objetivos del TFG

1.1.2.1 Objetivo General

El objetivo principal del TFG es diseñar e implementar una aplicación de teleconsulta basada en una arquitectura de microservicios. Esta aplicación debe permitir a los pacientes y profesionales de la salud realizar consultas médicas de manera remota, facilitando la gestión de citas, la comunicación en tiempo real y la administración de usuarios de forma segura y eficiente.

Este objetivo responde a la creciente necesidad de soluciones de teleconsulta en el sector de la salud, especialmente destacada durante la pandemia de COVID-19. La implementación de una arquitectura de microservicios se alinea con las mejores prácticas de desarrollo moderno, permitiendo escalabilidad, flexibilidad y mantenibilidad del sistema. Además, aborda problemas como la accesibilidad a servicios médicos y la eficiencia en la gestión de citas y comunicaciones entre pacientes y médicos.

1.1.2.2 Objetivos Específicos

1. Desarrollar un microservicio de autenticación y gestión de usuarios:
    - Implementar mecanismos de registro, inicio de sesión y gestión de perfiles.
    - Garantizar la seguridad de los datos de los usuarios mediante la implementación de JWT para la autenticación.
    - Este objetivo asegura que solo usuarios autorizados pueden acceder al sistema, protegiendo datos sensibles y manteniendo la integridad del sistema.
2. Implementar un microservicio para la gestión de citas:
    - Permitir a los usuarios reservar, modificar y cancelar citas.
    - Gestionar la disponibilidad de los profesionales de la salud y los slots de tiempo.
    - Facilita la organización y administración de consultas médicas, mejorando la eficiencia y accesibilidad para los pacientes y médicos.
3. Desarrollar un microservicio de señalización WebRTC:
    - Facilitar la comunicación en tiempo real entre pacientes y médicos.
    - Integrar funcionalidades de videollamadas y chat en la aplicación.
    - Permite la comunicación directa y en tiempo real, una característica esencial para las teleconsultas, haciendo el sistema funcional y útil en el contexto de la atención médica remota.
4. Diseñar e implementar el frontend utilizando React y React Bootstrap:
    - Crear una interfaz de usuario intuitiva y accesible.
    - Asegurar la responsividad y usabilidad de la aplicación en diferentes dispositivos usando un framework como bootstrap.
    - Asegurar una experiencia de usuario positiva y eficiente, fundamental para la adopción y uso

continuo de la aplicación.

1.1.3 Alcance del Proyecto

Este proyecto incluye:

- Desarrollo de un sistema de autenticación y gestión de usuarios.
- Implementación de un microservicio para la gestión de citas.
- Desarrollo de un microservicio para la señalización WebRTC.
- Diseño e implementación del frontend utilizando React y React Bootstrap.
- Realización de pruebas unitarias e integradas para asegurar la calidad del sistema.
- Despliegue de la aplicación en contenedores Docker y haciendo uso de servicios AWS como EKS, Balanceadores de carga y Route53, el servicio de DNS de AWS .

Este proyecto no incluye:

- Integración con sistemas de información de salud existentes.
- Conformidad con reglamentos de protección de datos como el Health Insurance Portability and Accountability Act (HIPAA) y el Reglamento General de Protección de Datos (RGPD)
- Gestión de perfiles de pacientes y profesionales de salud.

El proyecto debe completarse en un período determinado y con los recursos disponibles, por lo que hay que delimitar el alcance para asegurar la viabilidad. Además de priorizar las funcionalidades esenciales permite construir una base sólida y escalable, sobre la cual se pueden añadir más características en futuras iteraciones.

1.2 Introducción a la Arquitectura del Sistema

La arquitectura del sistema desarrollado para SEHHA, nombre de la aplicación que se va a desarrollar en este TFG, se basa en un enfoque de microservicios, diseñado para ofrecer una solución robusta, escalable y eficiente para la teleconsulta médica. Este enfoque permite que los diferentes componentes del sistema funcionen de manera autónoma, facilitando la mantenibilidad y escalabilidad del sistema. A continuación, se presenta una descripción general de los componentes clave de la arquitectura y sus interacciones.

1.2.1 Microservicios Principales

1.2.1.1 Microservicio de Autenticación y Gestión de Usuarios

Este componente se encarga de la autenticación y autorización de los usuarios mediante el uso de tokens JWT (JSON Web Tokens). Gestiona las operaciones de registro e inicio de sesión tanto para pacientes como para especialistas, asegurando que solo los usuarios autorizados puedan acceder a los recursos del sistema.

1.2.1.2 Microservicio de Gestión de Citas

Responsable de la creación, modificación, cancelación y gestión de las citas médicas. Este microservicio permite a los pacientes reservar y gestionar sus citas con especialistas, y a estos últimos, administrar su disponibilidad y horarios de manera eficiente.

1.2.1.3 Microservicio de Gestión de Señalización de Videollamadas WebRTC

Gestiona la señalización necesaria para establecer y mantener videollamadas entre pacientes y médicos mediante la tecnología WebRTC. Este microservicio maneja la señalización para la conexión en tiempo real, mientras que la comunicación directa de audio y vídeo ocurre de manera peer-to-peer en los navegadores de los usuarios.

1.2.2 Componentes y Tecnologías del Sistema

1.2.2.1 Frontend

Desarrollado utilizando React y Bootstrap, el frontend de SEHHA proporciona una interfaz de usuario intuitiva y responsiva. Interactúa con los microservicios a través de APIs RESTful, utilizando Axios para la comunicación y Socket.io para la señalización en tiempo real para WebRTC.

1.2.2.2 Backend

Construido con Node.js y Express.js, el backend está dividido en microservicios que cumplen funciones específicas. Cada microservicio tiene su propia API y base de datos, gestionada por MongoDB, lo que facilita la independencia y escalabilidad de cada componente.

1.2.2.3 Base de Datos

SEHHA utiliza MongoDB Atlas, una base de datos NoSQL en la nube, para almacenar los datos de usuarios, citas y otros elementos críticos. Cada microservicio tiene su propia base de datos, asegurando que los datos sean manejados de manera autónoma y eficiente.

1.2.2.4 Mensajería Asíncrona

RabbitMQ se emplea para la comunicación asíncrona entre microservicios, garantizando la entrega fiable de mensajes y la coordinación de eventos importantes como la creación de citas.

1.2.3 Seguridad y Escalabilidad

1.2.3.1 Seguridad

La seguridad del sistema se garantiza mediante el uso de JWT para la autenticación y autorización de usuarios. Las contraseñas se cifran utilizando algoritmos seguros antes de ser almacenadas en la base de datos. Toda la comunicación entre los componentes se realiza a través de HTTPS, asegurando la transmisión segura de datos.

1.2.3.2 Escalabilidad

SEHHA se despliega en contenedores Docker y se orquesta utilizando Kubernetes en un clúster de Amazon EKS (Elastic Kubernetes Service). Kubernetes permite la escalabilidad automática del sistema basado en la carga de trabajo, asegurando que el sistema pueda responder eficientemente a aumentos en la demanda.

1.3 Estructura de la memoria

La estructura de la memoria de este Trabajo de Fin de Grado (TFG) está organizada de manera que facilite la comprensión del desarrollo, implementación y despliegue de la aplicación SEHHA. El documento se inicia con una sección de **Introducción** donde se detallan el contexto, los objetivos y el alcance del proyecto, así como una visión general de la arquitectura del sistema y sus componentes principales. La segunda sección, **Tecnologías Usadas**, describe las diversas tecnologías empleadas en el desarrollo de la aplicación, incluyendo microservicios, bases de datos y herramientas de despliegue. La tercera sección, **Descripción del Sistema**, ofrece una visión detallada de las funcionalidades de la aplicación, el público objetivo y las plataformas soportadas, seguido de un análisis profundo de la arquitectura y los componentes del sistema. Subsecciones específicas están dedicadas a cada microservicio, abarcando su arquitectura, implementación, comunicación entre servicios, seguridad, escalabilidad y pruebas realizadas. La **Visión General de WebRTC** proporciona una comprensión de esta tecnología de comunicación en tiempo real, fundamental para la funcionalidad de videollamadas de SEHHA. La sección final de **Despliegue de la Aplicación SEHHA en AWS** aborda la arquitectura de despliegue, automatización de procesos, configuración de infraestructura, gestión de datos y seguridad. Finalmente, la sección de **Conclusión y Líneas Futuras** resume los logros del proyecto, discute las limitaciones encontradas y propone posibles mejoras y extensiones para trabajos futuros. Cada sección está

diseñada para proporcionar un enfoque integral y detallado del desarrollo del proyecto, asegurando una documentación exhaustiva y clara.

2 TECNOLOGÍAS USADAS

Esta sección proporciona una visión detallada de la arquitectura de microservicios y su relevancia en el desarrollo de software moderno. Comienza con una introducción a los microservicios, explicando sus conceptos básicos y diferenciándolos de las arquitecturas monolíticas. La evolución de la arquitectura de software se analiza, destacando la transición de sistemas monolíticos a arquitecturas orientadas a servicios (SOA) y finalmente a microservicios.

2.1. Microservicios y su importancia en el desarrollo de software

2.1.1 Introducción a los Microservicios

2.1.1.1 Definición y Concepto Básico:

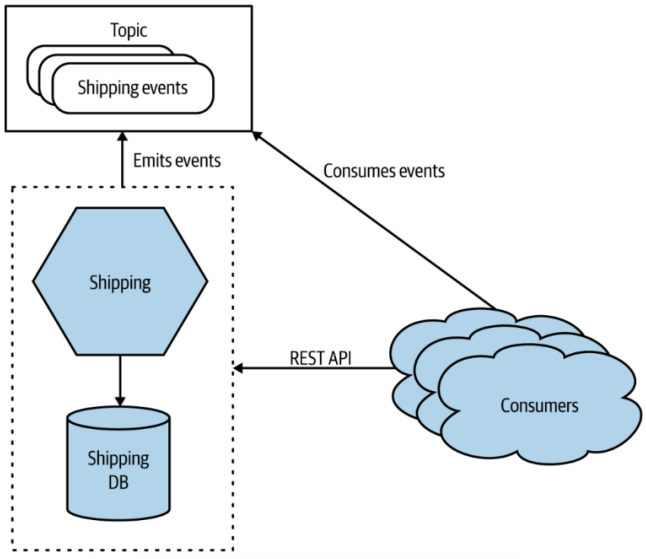


Figura 2-1. Diagrama de microservicios

Según el libro Building Microservices de Sam Newman, la arquitectura de microservicios se basa en una orientación a servicios, donde la comunicación entre servicios se realiza a través de la red. Un microservicio es un servicio modelado alrededor de un dominio específico. Este dominio representa una funcionalidad concreta y su acceso se realiza a través de uno o varios endpoints de red. La funcionalidad del microservicio se ofrece de manera transparente al exterior, mientras que su implementación interna permanece oculta. Uno de los patrones de arquitectura que separa la implementación interna de la interfaz externa es la arquitectura hexagonal. A continuación, se explican los conceptos claves de los microservicios:

- Despliegue Independiente: Los microservicios pueden ser modificados y desplegados de manera independiente, sin afectar a los demás. Esto requiere que los microservicios estén débilmente acoplados.
- Modelados alrededor de un Dominio de Negocio: Los microservicios se modelan en torno a funcionalidades específicas. Esta estructura permite que los cambios en una funcionalidad se realicen de manera eficiente mediante el despliegue de un microservicio específico, en lugar de una aplicación monolítica completa.
- Estado Propio: Cada microservicio mantiene su propio estado. Para acceder a los datos de otro microservicio, es necesario comunicarse directamente con él. Los microservicios deciden qué parte de su estado interno compartir.
- Tamaño: Los microservicios deben tener una interfaz lo más pequeña posible para minimizar el acoplamiento y facilitar la comunicación entre servicios.
- Flexibilidad: Los microservicios son flexibles en desarrollo, despliegue y escalabilidad. Pueden adaptarse rápidamente a cambios en el negocio o en la demanda del usuario.
- Alineación entre la Arquitectura y la Organización: Los equipos de desarrollo se alinean con los microservicios, utilizando un lenguaje ubicuo y siendo responsables de los cambios en todas las capas: frontend, backend y datos. Aunque a menudo la interfaz de usuario no es gestionada por un microservicio específico, hay un frontend que integra todos los microservicios.

Ahora que tenemos una noción clara de lo que es un microservicio, es importante compararlo con una arquitectura monolítica.

En una arquitectura monolítica, todas las funcionalidades forman parte de una única unidad de despliegue. Esto significa que cualquier cambio en la aplicación requiere el despliegue de toda la unidad, lo que puede ser ineficiente y propenso a errores en entornos de desarrollo y operación rápidos.

Esta distinción entre microservicios y arquitectura monolítica subraya la eficiencia y la agilidad que los microservicios aportan a las aplicaciones modernas, especialmente en el contexto de la telemedicina, donde la escalabilidad, la flexibilidad y la capacidad de desplegar cambios rápidamente son cruciales para responder a las necesidades dinámicas del sector salud.

2.1.2 Evolución de la Arquitectura de Software:

La evolución de las arquitecturas de software ha sido impulsada por la necesidad de manejar aplicaciones cada vez más complejas y escalables. Aquí se presenta un resumen de esta evolución:

- Sistemas Monolíticos: Los sistemas monolíticos son aplicaciones donde todos los componentes están integrados y funcionan como una sola unidad. Esta arquitectura era común en las primeras aplicaciones de software debido a su simplicidad y facilidad de desarrollo inicial. Conforme las aplicaciones crecieron en tamaño y complejidad, las limitaciones de los sistemas monolíticos se hicieron evidentes, especialmente en escalabilidad y mantenimiento. [10]
- Arquitectura Orientada a Servicios (SOA): SOA surgió como una respuesta a las limitaciones de los sistemas monolíticos. En SOA, las aplicaciones se dividen en servicios independientes que se comunican a través de interfaces bien definidas. Esto permitió una mayor flexibilidad y reusabilidad de los componentes. [11]
- Microservicios: Los microservicios son una evolución de SOA, caracterizados por servicios aún más pequeños y autónomos que se pueden desarrollar, desplegar y escalar de manera independiente. Esta arquitectura ganó popularidad por su capacidad para manejar aplicaciones escalables y flexibles. [12]

2.1.3 Beneficios de los Microservicios

2.1.3.1 Mantenibilidad y Flexibilidad

La arquitectura de microservicios permite actualizar y mantener cada servicio de forma aislada, sin necesidad

de intervenir en todo el sistema. Esto facilita la implementación de mejoras y correcciones de errores. Los microservicios permiten adoptar nuevas tecnologías rápidamente. Cada servicio puede utilizar la tecnología más adecuada para su funcionalidad específica, lo que facilita la adaptación a cambios en los requisitos del negocio.

En un sistema monolítico, cualquier cambio puede afectar a toda la aplicación, lo que hace que la mantenibilidad sea más compleja y costosa. En contraste, los microservicios permiten cambios localizados, reduciendo el riesgo y el costo.

Podemos elegir la tecnología adecuada para cada microservicio, asegurando que se utilice la herramienta correcta para cada tarea específica. La arquitectura de microservicios permite experimentar y adoptar nuevas tecnologías de manera más rápida y segura. Las pruebas en microservicios suponen un menor riesgo, ya que cualquier problema se limita al servicio específico que se está probando.

2.1.3.2 Escalabilidad

Los microservicios permiten escalar solo aquellos servicios que lo requieren, en lugar de escalar toda la aplicación. Esto resulta en un uso más eficiente de los recursos.

Ejemplos de Escalabilidad Horizontal y Vertical:

- Escalabilidad Horizontal: Añadir más instancias del mismo microservicio para distribuir la carga.
- Escalabilidad Vertical: Aumentar los recursos (CPU, memoria) de una instancia de microservicio específica.

2.1.3.3 Desarrollo y despliegue independientes

Los equipos pueden desarrollar microservicios de manera independiente, sin interferir en el trabajo de otros equipos. Esto acelera el proceso de desarrollo y despliegue. Los despliegues son menos costosos y riesgosos.

En sistemas monolíticos, pequeños cambios pueden afectar toda la aplicación, mientras que, en microservicios, los cambios son aislados. La independencia de los microservicios fomenta una colaboración más eficiente entre equipos, permitiendo una mayor agilidad y rapidez en el desarrollo.

2.1.3.4 Resiliencia y tolerancia a fallos

Los microservicios mejoran la resiliencia al permitir que los fallos se limiten a los servicios específicos donde ocurren, sin afectar el resto del sistema, permitiendo que el resto del sistema continúe funcionando normalmente.

2.1.3.5 Alineación organizacional

Los equipos pequeños, enfocados en microservicios específicos, son más productivos y efectivos. La propiedad de los microservicios puede cambiar para alinearse con las necesidades y cambios en la organización.

2.1.3.6 Componibilidad

La funcionalidad de los microservicios puede ser consumida de diferentes maneras, permitiendo una mayor flexibilidad en cómo los usuarios interactúan con los componentes del sistema.

Pensar en cómo los clientes interactúan con los componentes del sistema es crucial para diseñar microservicios efectivos y útiles.

2.1.4 Desafíos y Consideraciones

2.1.4.1 Developer Experience (DX)

La experiencia del desarrollador puede verse afectada negativamente debido a los microservicios que son intensivos en recursos. Por ejemplo, los microservicios basados en JVM pueden ser pesados para los equipos locales. En el otro extremo, el desarrollo en la nube puede introducir nuevos desafíos y dependencias.

2.1.4.2 Sobrecarga de Tecnología

Existe la posibilidad de acabar desarrollando microservicios en distintas tecnologías. Aunque esto proporciona opciones, también puede introducir una sobrecarga tecnológica y complejidad adicional. Esto se manifiesta en:

- Problemas de Consistencia de Datos entre Microservicios: Mantener la consistencia de datos entre múltiples microservicios es un desafío significativo.
- Complejidad en la Gestión: La gestión de múltiples microservicios introduce complejidades adicionales en términos de orquestación, monitoreo y mantenimiento.
- Herramientas y Prácticas para Manejar la Complejidad: Herramientas de orquestación como Kubernetes y prácticas de monitoreo avanzadas son esenciales para manejar la complejidad de los microservicios.

2.1.4.3 Comunicación entre Servicios

La comunicación entre microservicios presenta desafíos específicos, incluyendo problemas de latencia y consistencia de datos. La comunicación a través de la red introduce latencia y puede complicar la consistencia de datos entre servicios.

2.1.4.4 Seguridad

Los microservicios presentan desafíos de seguridad específicos, como la necesidad de asegurar cada servicio individualmente.

- Implementar autenticación y autorización robustas.
- Utilizar HTTPS para todas las comunicaciones.
- Monitorizar y auditar constantemente los servicios.

2.1.4.5 Reporting

En una arquitectura de microservicios, los datos están distribuidos a través de múltiples esquemas lógicamente aislados.

Enfoques más modernos como el uso de streaming para permitir informes en tiempo real sobre grandes volúmenes de datos pueden funcionar bien con una arquitectura de microservicios, aunque típicamente requieren la adopción de nuevas ideas y tecnologías asociadas.

2.1.4.6 Latencia

La información que previamente fluía dentro de un único proceso ahora necesita ser serializada, transmitida y deserializada a través de redes, lo que puede resultar en un aumento de la latencia del sistema.

2.1.4.7 Testing

El alcance de las pruebas end-to-end se vuelve muy amplio. Ahora es necesario ejecutar pruebas a través de múltiples procesos, todos los cuales deben ser desplegados y configurados apropiadamente para los escenarios de prueba.

Se debe estar preparado para falsos negativos que ocurren cuando problemas ambientales, como la caída de instancias de servicios o tiempos de espera en la red, causan fallos en las pruebas.

2.1.4.8 Coste

Es probable que en el corto plazo se observe un aumento en los costos debido a varios factores. Se necesitarán más procesos, más computadoras, más red, más almacenamiento y más software de soporte, lo que incurrirá en costos adicionales de licencias.

Cualquier cambio introducido en un equipo o una organización ralentizará el proceso en el corto plazo. Tomará

tiempo aprender nuevas ideas y descubrir cómo usarlas eficazmente. Esto resultará en una desaceleración directa en la entrega de nuevas funcionalidades o la necesidad de añadir más personas para compensar este costo.

2.1.5 Casos de Uso y Ejemplos Reales

2.1.5.1 Empresas que Utilizan Microservicios [13]

2.1.5.1.1 Amazon

Figura 2-2. Amazon logo

Amazon fue uno de los pioneros en la adopción de la arquitectura de microservicios. La compañía descompuso su aplicación monolítica en cientos de microservicios independientes, cada uno enfocado en una funcionalidad específica del negocio, como el procesamiento de pedidos, la gestión de inventarios y la recomendación de productos.

2.1.5.1.2 Netflix:

Figura 2-3. Logo de Netflix

Netflix migró a una arquitectura de microservicios para manejar su rápido crecimiento y la necesidad de una entrega continua de nuevas funcionalidades. Descompusieron su sistema en cientos de microservicios que se comunican entre sí a través de APIs bien definidas.

2.2 Tecnologías usadas

Para construir una aplicación robusta y eficiente, es fundamental utilizar una variedad de herramientas y tecnologías que aborden las necesidades específicas de cada componente del sistema. Esto no solo optimiza el rendimiento, sino que también permite una mayor flexibilidad y capacidad de adaptación a los cambios. La diversidad tecnológica permite abordar de manera específica y eficiente los distintos aspectos del desarrollo, desde la interfaz de usuario hasta la comunicación y la gestión de datos. Una arquitectura diversificada facilita la escalabilidad, mejora la resiliencia y permite la adopción rápida de nuevas tecnologías.

2.2.1 React

Figura 2-4. Logo de React

React es una biblioteca de JavaScript diseñada para construir interfaces de usuario dinámicas y reutilizables mediante el uso de componentes. En este proyecto, React se utiliza para desarrollar la interfaz de usuario. La capacidad de React para manejar el DOM de manera eficiente y reutilizar componentes facilita la creación de una experiencia de usuario coherente. Un ejemplo de su uso en el proyecto es la creación de formularios de registro y autenticación, así como el manejo de la vista de gestión de citas.. [14]

2.2.2 Bootstrap

Figura 2-5. Logo de Bootstrap

Bootstrap es un framework CSS que facilita el diseño de interfaces responsivas y estéticamente agradables mediante el uso de un sistema de grid y componentes predefinidos. En este proyecto, Bootstrap se integra para asegurar que la aplicación sea visualmente atractiva y accesible en diversos dispositivos. Sus componentes predefinidos y su diseño responsivo permiten crear una interfaz de usuario que se adapta a diferentes tamaños de pantalla, mejorando así la experiencia del usuario en dispositivos móviles y de escritorio. [15]

2.2.3 Node.js

Figura 2-7. Logo de Node.js

Node.js es un entorno de ejecución de JavaScript basado en el motor V8 de Chrome que permite a los desarrolladores construir aplicaciones del lado del servidor de manera eficiente y escalable. En el proyecto,

Node.js 2.2.4 se utiliza para el desarrollo del backend del sistema, facilitando la creación de servidores y aplicaciones web de alto rendimiento. Las ventajas de Node.js incluyen su arquitectura basada en eventos y su modelo de E/S no bloqueante, lo que permite manejar múltiples conexiones simultáneamente sin bloquear el hilo de ejecución. En este proyecto, Node.js se utiliza para el desarrollo del backend para cada uno de los microservicios, facilitando la creación de lógica que sirve las peticiones de los usuarios de estos microservicios.

Node.js permite a los desarrolladores escribir aplicaciones en JavaScript tanto para el frontend como para el backend, promoviendo un desarrollo más consistente y reutilizable. También es conocido por su ecosistema de paquetes, accesibles a través de npm (Node Package Manager), que proporciona miles de módulos adicionales para extender la funcionalidad de las aplicaciones. Esto incluye módulos para gestionar autenticación, procesamiento de datos, conexión a bases de datos, y más, facilitando el desarrollo de aplicaciones modernas y eficientes.

2.2.4 Express.js

Figura 2-8. Logo de Express.js

Express.js es un framework de Node.js para construir aplicaciones web y APIs RESTful de manera sencilla y eficiente. En el proyecto, Express.js se emplea para desarrollar el backend del sistema, permitiendo a los distintos microservicios que se van a desarrollar exponer APIs robustas y flexibles. Sus ventajas incluyen la simplicidad, flexibilidad y el uso de middleware. Express.js permite a los desarrolladores configurar rápidamente servidores web y APIs, manejar solicitudes HTTP con rutas específicas y utilizar middleware para gestionar funcionalidades adicionales como la autenticación, la validación de datos y el manejo de errores. [16]

2.2.5 MongoDB

Figura 2-9. Logo de MongoDB

MongoDB es una base de datos NoSQL diseñada para el almacenamiento de datos no estructurados, ofreciendo flexibilidad de esquemas y escalabilidad horizontal. MongoDB se utiliza para gestionar datos no estructurados, asegurando flexibilidad y escalabilidad en el manejo de grandes volúmenes de información. A diferencia de las bases de datos relacionales tradicionales, MongoDB almacena datos en documentos JSON flexibles, lo que permite a los desarrolladores almacenar y consultar datos de manera eficiente y sin necesidad de esquemas predefinidos. En este proyecto, MongoDB se utiliza para gestionar datos no estructurados de usuarios, citas y otros elementos críticos. Se utiliza la librería “mongoose” para establecer esquemas para los distintos modelos de datos que se establecen en cada uno de los microservicios. [17]

2.2.6 JSON Web Token (JWT)

Figura 2-10. Logo de JWT

JWT (JSON Web Tokens) es un estándar para la autenticación y autorización segura mediante tokens compactos y auto-contenidos. Los JWT se utilizan para autenticar y autorizar usuarios, asegurando la seguridad de las APIs. Sus ventajas incluyen la seguridad y facilidad de uso. Los JWT se usan para transmitir información entre dos partes de manera segura y compactos, lo que los hace ideales para usar en entornos de servicios web modernos, permitiendo la autenticación basada en tokens que pueden verificarse fácilmente sin tener que almacenar más información en el servidor. En este proyecto, los JWT se utilizan para autenticar y autorizar usuarios, asegurando la seguridad de las APIs, estos tokens son creados por el microservicio de gestión de usuarios, y luego son utilizados por el resto de servicios para la autorización a nivel de APIs basada en roles y en el frontend se usan para gestionar las sesiones de usuarios y especialistas autenticados.

2.2.7 RabbitMQ

Figura 2-11. Logo de RabbitMQ

RabbitMQ es un intermediario de mensajes que facilita la comunicación asíncrona entre diferentes componentes del sistema. En este proyecto, RabbitMQ se emplea para gestionar la mensajería asíncrona entre microservicios, mejorando la eficiencia y fiabilidad de las comunicaciones internas. Utiliza un sistema de encolado que asegura la entrega fiable de mensajes, lo cual es crucial para mantener la coherencia y la integridad de los datos en el sistema. [18]

2.2.8 Socket.io

Figura 2-12. Logo de Socket.io

Socket.io es una biblioteca de JavaScript que permite la comunicación bidireccional en tiempo real entre clientes web y servidores. En este proyecto, Socket.io se utiliza para la señalización entre pares en WebRTC, facilitando la comunicación en tiempo real entre los usuarios a través del servicio de señalización de vídeo. Sus ventajas incluyen la comunicación bidireccional en tiempo real y la facilidad de implementación. Socket.io permite a las aplicaciones web y móviles establecer conexiones persistentes y bidireccionales, permitiendo la transmisión de

datos en tiempo real como chats, notificaciones y actualizaciones en vivo, lo cual es crucial para aplicaciones que requieren una comunicación rápida y continua. [19]

2.2.9 WebRTC

Figura 2-12. Logo de WebRTC

WebRTC es una tecnología que permite la comunicación en tiempo real entre navegadores, soportando transmisiones de audio, vídeo y datos. WebRTC se utiliza para habilitar videoconferencias y transmisiones de datos en tiempo real entre usuarios (pacientes) y especialistas, crucial para la funcionalidad de teleconsulta. Entre sus ventajas se encuentran la baja latencia y la transmisión de audio y vídeo. WebRTC permite a las aplicaciones web y móviles capturar y transmitir audio, vídeo y datos directamente desde los navegadores sin necesidad de plugins adicionales, lo cual es esencial para aplicaciones de teleconsulta médica que requieren comunicación instantánea y de alta calidad. [20]

2.2.10 Docker

Figura 2-13. Logo de Docker

Docker es una plataforma de contenerización que permite empaquetar aplicaciones y sus dependencias en contenedores, asegurando que se ejecuten de manera consistente en cualquier entorno. Docker se emplea para facilitar la contenerización y despliegue de microservicios, asegurando consistencia y portabilidad entre entornos de desarrollo y producción. Sus ventajas incluyen el aislamiento de aplicaciones y la consistencia entre entornos. Docker permite a los desarrolladores empaquetar una aplicación con todas sus partes necesarias (como bibliotecas y dependencias) y enviarla como un solo paquete, asegurando que la aplicación se ejecute de manera uniforme sin importar el entorno en el que se despliegue. Todos los microservicios y el frontend desarrollados en este proyecto están contenerizados para su fácil despliegue en el entorno en Kubernetes.

2.2.11 Kubernetes

Figura 2-14. Logo de Kubernetes

Kubernetes es una plataforma de orquestación de contenedores que automatiza el despliegue, escalado y manejo

de aplicaciones en contenedores. En este proyecto, utilizamos Amazon Elastic Kubernetes Service (EKS), que es un servicio de AWS que ofrece clusters de Kubernetes gestionados, para gestionar la orquestación y el despliegue de contenedores Docker en un entorno de producción. EKS permite una alta disponibilidad y escalabilidad, proporcionando herramientas para el monitoreo, la gestión del ciclo de vida y la recuperación ante fallos de las aplicaciones desplegadas. Además, EKS facilita la exposición de los servicios mediante balanceadores de carga de AWS, asegurando una distribución eficiente del tráfico y alta disponibilidad. También utilizamos los certificados SSL/TLS generados por AWS Certificate Manager (ACM) para asegurar la comunicación HTTPS en nuestros servicios.

2.2.12 Amazon Web Services (AWS)

Figura 2-15. Logo de AWS

Amazon Web Services (AWS) proporciona la infraestructura en la nube para la implementación y operación de la aplicación. Utilizamos varios servicios de AWS para diferentes propósitos:

Figura 2-16. Logo de Amazon EKS

- EKS (Elastic Kubernetes Service): Para la orquestación y gestión de contenedores. EKS permite desplegar, gestionar y escalar aplicaciones en contenedores utilizando Kubernetes, y facilita la creación de clústeres de Kubernetes. Esto asegura que la aplicación pueda manejar cargas de trabajo variables, manteniendo la alta disponibilidad y el rendimiento óptimo.

Figura 2-17. Logo de AWS Certificate Manager

- ACM (AWS Certificate Manager): Para la gestión y despliegue de certificados SSL/TLS, garantizando que la comunicación entre los servicios sea segura. ACM simplifica la administración de certificados al automatizar su renovación y eliminación.
- Load Balancers: AWS ofrece balanceadores de carga que distribuyen automáticamente el tráfico entrante entre los microservicios. Esto evita sobrecargas en los servidores y asegura que las solicitudes se manejen de manera eficiente, mejorando la disponibilidad y la tolerancia a fallos del sistema.

Figura 2-18. Logo de AWS Route53

- Route 53: Amazon Route 53 es un servicio web de DNS escalable y altamente disponible diseñado para proporcionar una manera fiable de direccionar las solicitudes de los usuarios a la infraestructura de aplicaciones implementada en AWS. En este proyecto, Route 53 se utiliza para gestionar el dominio sehha-telemedicine.com, redirigiendo las peticiones entrantes a los balanceadores de carga de AWS. Esto asegura que las solicitudes de los usuarios lleguen a los microservicios correctos, distribuyendo el tráfico de manera eficiente y mejorando la disponibilidad del sistema. Además, Route 53 facilita la gestión de registros DNS, lo que simplifica la configuración y administración del dominio.

2.2.13 eksctl

eksctl es una herramienta de línea de comandos para la creación y gestión de clústeres de Kubernetes en Amazon EKS. En este proyecto, eksctl se utiliza para simplificar y automatizar el proceso de despliegue y configuración del clúster de Kubernetes en EKS. Esto incluye la creación de nodos, configuración de redes y otras tareas relacionadas con la gestión del clúster. eksctl permite realizar operaciones complejas con comandos simples, lo que acelera el proceso de implementación y reduce el riesgo de errores de configuración.

2.2.14 GitHub Actions

GitHub Actions es una herramienta de integración y entrega continua (CI/CD) que se utiliza para automatizar los flujos de trabajo de desarrollo. En este proyecto, GitHub Actions se utiliza para automatizar la construcción, pruebas y despliegue de los microservicios. Esto garantiza que cada cambio en el código sea probado y desplegado automáticamente, facilitando un ciclo de desarrollo ágil y eficiente. GitHub Actions permite configurar pipelines que se ejecutan en eventos específicos, como commits o pull requests, asegurando que el código en producción sea siempre de alta calidad.

2.2.15 Docker Hub

Docker Hub es un servicio de registro de imágenes de contenedores que permite almacenar y compartir imágenes Docker. En este proyecto, Docker Hub se utiliza para almacenar las imágenes de los microservicios construidas durante el proceso de integración continua. Esto facilita la distribución y despliegue de las imágenes en diferentes entornos. Al utilizar Docker Hub, se asegura que las versiones correctas de las imágenes estén disponibles para los despliegues, y se mantiene un historial de las versiones anteriores para facilitar la reversión si es necesario.

2.2.16 GitHub

Figura 2-19. Logo de Github

GitHub es una plataforma de alojamiento de código que permite el control de versiones y la colaboración en el desarrollo de software. En este proyecto, GitHub se utiliza para almacenar y gestionar los repositorios de código de los microservicios. Facilita la colaboración entre desarrolladores, permitiendo el seguimiento de cambios, la revisión de código y la gestión de issues y pull requests. GitHub también integra herramientas de CI/CD, como GitHub Actions, para automatizar los flujos de trabajo de desarrollo y despliegue.

2.2.17 Git Bash

Figura 2-20. Logo de Git Bash

Git Bash es una herramienta de línea de comandos que proporciona una interfaz de Bash para Windows, permitiendo el uso de comandos de Git. En este proyecto, Git Bash se utiliza para el control de versiones durante el desarrollo, facilitando la gestión de repositorios y la colaboración en el código. Git Bash permite ejecutar comandos de Git, scripts de Bash y otras herramientas de línea de comandos en un entorno familiar para los desarrolladores, mejorando la eficiencia y la productividad.

2.2.18 Visual Studio Code

Figura 2-21. Logo de Visual Studio Code

Visual Studio Code es un entorno de desarrollo integrado (IDE) que ofrece herramientas y extensiones para

facilitar el desarrollo de software. En este proyecto, Visual Studio Code se utiliza como IDE principal para escribir, depurar y gestionar el código de los microservicios. Con su amplia gama de extensiones, Visual Studio Code proporciona soporte para múltiples lenguajes de programación, integración con Git, depuración avanzada y personalización del entorno de desarrollo, lo que mejora la productividad y la calidad del código.

2.2.19 Postman

Figura 2-22. Logo de Postman

Postman es una herramienta para desarrollar, probar y documentar APIs. En este proyecto, Postman se utiliza para enviar peticiones HTTP a los microservicios y verificar su funcionamiento, facilitando las pruebas y la validación de las APIs durante el desarrollo. Postman permite crear colecciones de pruebas que pueden ejecutarse automáticamente, generando informes detallados sobre el comportamiento de las APIs y asegurando que cumplan con los requisitos especificados.

2.2.20 Navegadores (Chrome y Firefox)

Google Chrome y Mozilla Firefox son navegadores web utilizados para probar y depurar la interfaz de usuario desarrollada en React. En este proyecto, ambos navegadores se utilizan para asegurar que el frontend funcione correctamente y sea compatible con diferentes navegadores, proporcionando una experiencia de usuario consistente y fluida. Las herramientas de desarrollo incorporadas en estos navegadores permiten inspeccionar el DOM, depurar JavaScript, analizar el rendimiento y solucionar problemas de compatibilidad, garantizando una alta calidad en la interfaz de usuario.

3 DESCRIPCIÓN DEL SISTEMA

La sección "Descripción del Sistema" proporciona una visión exhaustiva sobre el diseño, las funcionalidades y la arquitectura de SEHHA, la aplicación de teleconsulta médica desarrollada en este TFG. Esta sección se organiza en torno a varios subtemas clave que abarcan desde una visión general de la aplicación hasta los detalles técnicos de su arquitectura.

3.1. Visión general de la aplicación

3.1.1 Introducción a la aplicación

3.1.1.1 Nombre y Propósito de la Aplicación:

Figura 3-1. Logo de SEHHA

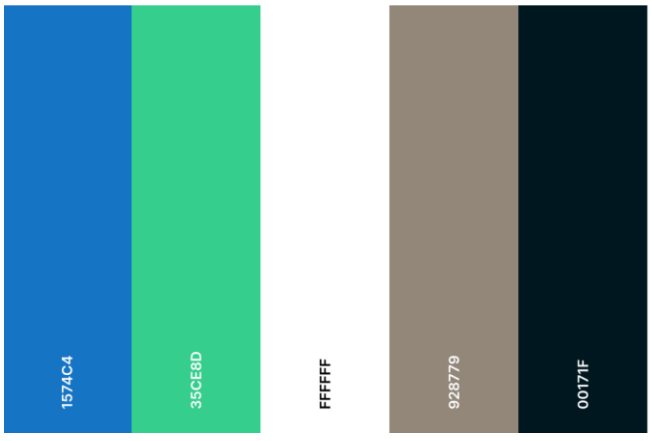


Figura 3-2. Colores de marca de SEHHA

La aplicación desarrollada se denomina SEHHA, del árabe “صحة” que significa "salud”. Este nombre refleja la misión central de la aplicación de proporcionar acceso a servicios de salud a través de una plataforma de

teleconsulta médica eficiente y segura.

3.1.1.2 Propósito general y misión de la aplicación:

El propósito general de SEHHA es proporcionar una plataforma integral de teleconsulta médica, permitiendo a los pacientes recibir atención médica de alta calidad sin necesidad de desplazarse físicamente a un centro de salud. La misión de la aplicación es mejorar el acceso a servicios de salud, especialmente en áreas rurales o con disponibilidad limitada de infraestructura médica, facilitando la continuidad de la atención médica y superando las barreras geográficas y logísticas para ofrecer una atención accesible y conveniente.

SEHHA se desarrolló en un contexto de creciente demanda de soluciones de telemedicina, una necesidad que se intensificó especialmente durante la pandemia de COVID-19. Durante este periodo, se hizo evidente la importancia de minimizar los riesgos de contagio mientras se proporciona atención médica a distancia. La aplicación aborda varios problemas críticos al proporcionar una alternativa conveniente para los pacientes, eliminando la necesidad de esperas prolongadas y desplazamientos innecesarios, especialmente para aquellos con movilidad reducida o con horarios laborales estrictos.

El desarrollo de SEHHA se centró en resolver problemas clave en el acceso a la atención médica. En muchas áreas rurales y remotas, la falta de infraestructura médica adecuada limita el acceso a servicios de salud esenciales. SEHHA permite a los residentes de estas áreas acceder a profesionales de la salud sin necesidad de viajar largas distancias, garantizando una atención continua y de calidad. Además, al ofrecer una plataforma para consultas médicas en línea, SEHHA contribuye a disminuir la afluencia de pacientes en hospitales y clínicas, permitiendo que estos centros se concentren en casos más graves y urgentes.

En resumen, SEHHA se posiciona como una solución innovadora y necesaria en el campo de la telemedicina, mejorando significativamente la calidad de vida de sus usuarios al proporcionar un acceso fácil y eficiente a servicios de salud de alta calidad.

3.1.2 Funcionalidades Principales de SEHHA

SEHHA, una aplicación de teleconsulta médica, está diseñada para mejorar la accesibilidad y la eficiencia de la atención médica mediante una serie de funcionalidades innovadoras. A continuación, se describen las principales funcionalidades de SEHHA y cómo abordan las necesidades tanto de los pacientes como de los profesionales de la salud.

3.1.2.1 Registro y Autenticación

Los usuarios pueden registrarse en la plataforma proporcionando información básica como nombre, correo electrónico y contraseña. La autenticación se maneja mediante tokens JWT, asegurando que solo los usuarios autorizados puedan acceder a la aplicación.

Esta funcionalidad proporciona una capa de seguridad esencial, protegiendo la información sensible de los usuarios. Facilita un acceso rápido y seguro a todas las funcionalidades de la aplicación, mejorando la confianza y la satisfacción del usuario.

3.1.2.2 Gestión de Citas para Usuarios

Los usuarios pueden buscar y reservar citas con los especialistas disponibles según su conveniencia. Los usuarios tienen la opción de cancelar citas si sus planes cambian, proporcionando flexibilidad y control. En el momento de la cita, los usuarios pueden unirse a una sesión de videollamada directamente desde la aplicación.

Estas funcionalidades mejoran la accesibilidad a los servicios de salud, permitiendo a los pacientes gestionar sus citas de manera eficiente y flexible. Poder unirse a las citas de forma virtual reduce la necesidad de desplazamiento, crucial para quienes viven en áreas rurales.

3.1.2.3 Gestión de Citas para Especialistas

Los especialistas pueden definir sus horarios disponibles creando timeslots para citas. Si es necesario, los

especialistas pueden eliminar timeslots, gestionando su tiempo de manera efectiva. Los especialistas pueden unirse a las videollamadas programadas con los pacientes.

Estas funcionalidades permiten a los especialistas gestionar sus horarios de manera más efectiva, optimizando su tiempo y aumentando la eficiencia de su práctica. La capacidad de unirse a las citas virtuales facilita la prestación de atención médica sin necesidad de presencia física.

3.1.2.4 Funcionalidad de Videollamada

SEHHA utiliza WebRTC para facilitar videollamadas entre pacientes y médicos, ofreciendo una comunicación en tiempo real de alta calidad.

Las videollamadas permiten consultas médicas remotas, lo cual es especialmente beneficioso en situaciones de emergencia o para pacientes que no pueden desplazarse fácilmente. Esta funcionalidad mejora significativamente la accesibilidad a la atención médica y puede ayudar a reducir la carga en las instalaciones de salud físicas.

3.1.2.5 Selección y Filtrado de Especialistas

Los usuarios pueden seleccionar especialistas de diversas especializaciones según sus necesidades médicas. Los usuarios pueden filtrar las citas disponibles según el tiempo, el especialista o la especialidad.

La posibilidad de seleccionar y filtrar especialistas según la necesidad del usuario garantiza que los pacientes encuentren rápidamente el profesional adecuado. Esto mejora la experiencia del usuario al permitir una búsqueda eficiente y personalizada, facilitando el acceso a la atención médica especializada.

En conjunto, estas funcionalidades hacen de SEHHA una herramienta integral y eficaz para la teleconsulta médica, mejorando tanto la eficiencia de los profesionales de la salud como la accesibilidad y conveniencia para los pacientes. La combinación de seguridad, gestión eficiente de citas, videollamadas y opciones de selección y filtrado de especialistas crea una plataforma robusta y útil para la prestación de servicios de salud en el entorno digital.

3.2 Público objetivo y plataformas

3.2.1 Público Objetivo de SEHHA

La aplicación SEHHA está diseñada para servir a dos grupos principales de usuarios: los pacientes y los profesionales de la salud. Los pacientes que usan SEHHA buscan acceder a servicios médicos a distancia por limitaciones físicas, logísticas o por preferencia personal. Estos pacientes pueden incluir aquellos con movilidad reducida, residentes en áreas rurales con acceso limitado a instalaciones médicas, y aquellos que prefieren la conveniencia de recibir atención médica desde su hogar. Además, los pacientes con enfermedades crónicas que requieren monitoreo y consultas regulares también encuentran en SEHHA una solución valiosa para gestionar sus necesidades de salud de manera eficiente.

Por otro lado, los profesionales de la salud, como médicos, especialistas, psicólogos, terapeutas y otros proveedores de atención médica utilizan SEHHA para gestionar sus horarios de consulta, llevar a cabo citas médicas de forma remota y mantener una comunicación constante con sus pacientes. Estos profesionales valoran la capacidad de la plataforma para organizar y facilitar sus interacciones con los pacientes, optimizando así su tiempo y recursos.

3.2.2 Plataformas

3.2.2.1 Plataformas Soportadas

SEHHA está disponible a través de una interfaz web responsive, permitiendo accesibilidad desde múltiples dispositivos, incluyendo computadoras de escritorio, tabletas y teléfonos móviles. Aunque no existe una aplicación móvil nativa, el diseño responsive de SEHHA garantiza que la plataforma sea accesible y funcional en cualquier dispositivo con acceso a un navegador web moderno.

3.2.2.2 Justificación de la Elección de estas Plataformas

La decisión de hacer que SEHHA esté disponible a través de una interfaz web responsive en lugar de desarrollar aplicaciones móviles nativas se basa en varios factores clave:

- Accesibilidad Universal: Al ser accesible desde cualquier navegador web, SEHHA no limita su uso a usuarios de ciertos sistemas operativos móviles, asegurando así que todos los usuarios, independientemente de si utilizan dispositivos Android, iOS u otros, puedan acceder a la plataforma sin restricciones.
- Facilidad de Mantenimiento: Mantener una única versión de la aplicación web reduce significativamente los costos y el esfuerzo de mantenimiento. Las actualizaciones y mejoras se pueden implementar de manera uniforme y simultánea para todos los usuarios sin la necesidad de gestionar versiones separadas para diferentes sistemas operativos.
- Experiencia de Usuario Consistente: Un diseño web responsive asegura que los usuarios tengan una experiencia de usuario consistente en todos sus dispositivos. La interfaz se adapta automáticamente al tamaño de la pantalla, garantizando una usabilidad óptima en computadoras de escritorio, tabletas y teléfonos móviles.
- Optimización de Recursos: Desarrollar y mantener aplicaciones móviles nativas para múltiples plataformas puede ser costoso y consumir muchos recursos. Al centrarse en una única plataforma web, SEHHA puede optimizar el uso de recursos y asegurar una mayor eficiencia en el desarrollo y mantenimiento de la aplicación.

Estas decisiones estratégicas permiten que SEHHA maximice su accesibilidad y usabilidad, proporcionando una solución eficiente y conveniente tanto para los pacientes como para los profesionales de la salud.

3.3 Arquitectura del sistema:

3.3.1 Introducción a la Arquitectura del Sistema

3.3.1.1 Visión General

La arquitectura de SEHHA está basada en un enfoque de microservicios, diseñado para ofrecer una solución robusta, escalable y eficiente para la teleconsulta médica. Este diseño se estructura en tres microservicios principales y un frontend, cada uno desempeñando funciones específicas y colaborando de manera orquestada para proporcionar una experiencia de usuario fluida y efectiva.

La arquitectura de SEHHA se compone de los siguientes componentes principales:

- Microservicio de Autenticación y Autorización: Este microservicio se encarga de gestionar el registro, inicio de sesión y manejo de roles y permisos de los usuarios. Utiliza JWT (JSON Web Tokens) para asegurar que solo los usuarios autenticados y autorizados puedan acceder a las diferentes funcionalidades de la aplicación. Este microservicio asegura la protección de datos sensibles y mantiene la integridad y seguridad del sistema.
- Microservicio de Gestión de Citas: Responsable de la creación, modificación, cancelación y gestión de las citas médicas. Permite a los usuarios reservar y gestionar sus citas de manera eficiente y a los profesionales de la salud administrar su disponibilidad y horarios. Este microservicio maneja la lógica de negocio relacionada con la programación y administración de citas.
- Microservicio de Gestión de Señalización de Videollamadas WebRTC: Facilita la comunicación en tiempo real entre pacientes y médicos mediante la tecnología WebRTC. Este microservicio maneja la señalización necesaria para establecer y mantener videollamadas, asegurando que la transmisión de audio y vídeo sea fluida y de alta calidad. Se encarga del intercambio de ofertas, respuestas y candidatos ICE entre los pares conectados.
- Frontend: Desarrollado utilizando React y React Bootstrap, el frontend de SEHHA proporciona una interfaz de usuario intuitiva y responsiva. Aunque no hay aplicaciones móviles nativas, el diseño responsive permite que la aplicación sea accesible y funcional en dispositivos móviles a través de

navegadores web. El frontend interactúa con los microservicios a través de APIs RESTful para ofrecer una experiencia de usuario coherente y eficiente.

3.3.2 Componentes Principales de la Arquitectura

3.3.2.1 Frontend

El frontend de SEHHA está desarrollado utilizando React y Bootstrap, sirve como la interfaz principal para todos los usuarios, incluidos pacientes y profesionales de la salud. Las funcionalidades que maneja incluyen:

Registro y Autenticación: Permite a los usuarios registrarse y acceder a la plataforma utilizando credenciales seguras.

- Gestión de Citas: Los usuarios pueden reservar, modificar y cancelar citas. Los profesionales de la salud pueden crear y gestionar sus horarios de citas.
- Videollamadas: Integra la funcionalidad de videollamadas a través de WebRTC, permitiendo consultas en tiempo real entre pacientes y médicos.
- Filtrado de Citas: Los usuarios pueden buscar y filtrar citas según criterios como tiempo, especialista o especialidad.
- Interfaz Intuitiva: La interfaz está diseñada para ser intuitiva y fácil de usar, con una disposición clara y accesible de todas las funciones.

3.3.2.2 Comunicación con el Backend

Para la comunicación con los microservicios del backend, el frontend utiliza axios, una biblioteca de JavaScript para hacer solicitudes HTTP. Axios permite enviar solicitudes a las APIs REST de los microservicios de autenticación y gestión de usuarios, así como al microservicio de gestión de citas. Esta comunicación es crucial para el intercambio de datos entre el frontend y el backend, garantizando una experiencia de usuario fluida y dinámica. Por otra parte, se usa socket.io para la interacción con el servidor de señalización de WebRTC.

3.3.2.3 Backend

El backend de SEHHA está construido utilizando Node.js y Express.js, formando parte de la stack MERN (MongoDB, Express.js, React, Node.js). Node.js permite ejecutar JavaScript en el servidor, ofreciendo un entorno eficiente y escalable para manejar múltiples solicitudes simultáneamente. Express.js, un marco web minimalista para Node.js, facilita la creación de APIs.

Para la gestión de MongoDB, se utiliza Mongoose, una biblioteca que proporciona una solución basada en esquemas para modelar datos en MongoDB. Actualmente, SEHHA utiliza MongoDB Atlas, una solución de base de datos en la nube, para gestionar y almacenar datos.

Cada microservicio en el backend tiene su propia API, lo que facilita la interacción con el frontend. Las APIs están diseñadas para ser RESTful, lo que permite una comunicación clara y estructurada entre el frontend y el backend. Además, para asegurar que los servicios estén débilmente acoplados, se utilizan colas de mensajes. Esto permite que los microservicios se comuniquen entre sí de manera eficiente sin depender directamente uno del otro. Un caso especial es el servicio de señalización de WebRTC que utiliza socket.io.

3.3.2.4 Base de Datos

SEHHA utiliza MongoDB, una base de datos NoSQL conocida por su flexibilidad y escalabilidad. Cada microservicio tiene su propia base de datos, asegurando que cada uno sea propietario de sus propios datos. Esto no solo mejora la seguridad y la gestión de datos, sino que también facilita la escalabilidad y el mantenimiento del sistema. La separación de bases de datos permite que cada microservicio maneje sus datos de manera independiente, lo que es crucial para mantener una arquitectura de microservicios efectiva.

3.3.3 Diagrama de Arquitectura

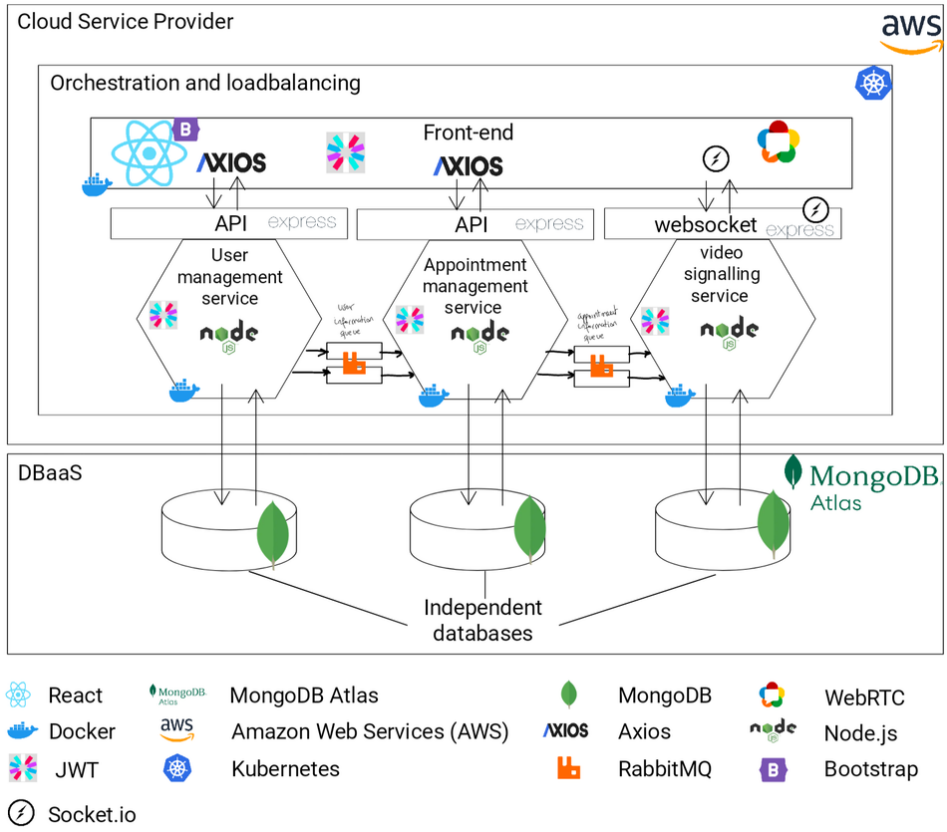


Figura 3-3. Arquitectura de SEHHA

El diagrama de arquitectura del sistema proporciona una visión integral de cómo están estructurados los componentes y cómo interactúan entre sí.

AWS proporciona la infraestructura necesaria para soportar todos los componentes del sistema. Kubernetes maneja la orquestación de contenedores Docker, asegurando una distribución equilibrada de las cargas de trabajo y permitiendo una escalabilidad automática según la demanda. El balanceo de carga distribuye las solicitudes entrantes de manera eficiente entre los diferentes microservicios, evitando sobrecargas y garantizando un rendimiento óptimo.

En el frontend, React se utiliza para crear una interfaz de usuario dinámica y responsiva. Axios facilita las operaciones CRUD (crear, leer, actualizar y eliminar) a través de solicitudes HTTP hacia los microservicios. Socket.io es crucial para la comunicación en tiempo real, especialmente para las videollamadas, que utilizan WebRTC para transmitir vídeo y audio en tiempo real. Esta configuración proporciona una experiencia de usuario coherente y fluida.

Cada microservicio en el backend está diseñado para cumplir una función específica. El servicio de gestión de usuarios gestiona el registro y autenticación de usuarios (pacientes y especialistas), utilizando JWT para garantizar que solo los usuarios autenticados puedan acceder a los recursos. El servicio de gestión de citas se encarga de la gestión de citas, permitiendo a los usuarios programar, modificar y cancelar citas según su conveniencia y a los profesionales gestionar su disponibilidad. El servicio de señalización de vídeo maneja la señalización necesaria para establecer y mantener videollamadas usando webRTC, coordinando las conexiones en tiempo real.

MongoDB Atlas se utiliza para almacenar los datos de cada microservicio de manera independiente, lo que mejora la escalabilidad y asegura que cada servicio pueda operar de manera autónoma.

La comunicación entre los componentes del sistema se maneja mediante varios métodos clave. Las APIs REST, implementadas con Axios en el frontend, permiten una comunicación estructurada y fácil de mantener con los microservicios. Socket.io proporciona una comunicación bidireccional de baja latencia entre el frontend y el

Video Signalling Service, utilizando WebSockets para manejar la transmisión de vídeo y audio en tiempo real. Internamente, los microservicios se comunican de manera asíncrona mediante RabbitMQ, lo que facilita una comunicación fiable y eficiente incluso si algún componente está temporalmente inactivo.

La seguridad del sistema se garantiza mediante varios mecanismos. JWT se utiliza para autenticar a los usuarios, generando tokens que se incluyen en las solicitudes para asegurar el acceso a los recursos protegidos. La autorización se maneja mediante roles y permisos, asignando roles específicos a los usuarios, lo que determina sus capacidades dentro del sistema. Los datos sensibles, como contraseñas, se hashean antes de almacenarse y toda la comunicación se realiza a través de HTTPS para asegurar la transmisión segura de datos.

La escalabilidad del sistema se asegura mediante Kubernetes, que gestiona automáticamente los contenedores Docker basándose en la carga de trabajo. Esto permite que el sistema responda eficientemente a aumentos en la demanda sin intervención manual. El balanceo de carga distribuye las solicitudes entrantes entre múltiples instancias de microservicios, evitando sobrecargas y asegurando un rendimiento óptimo.

En conclusión, la combinación de estas tecnologías crea un sistema robusto, escalable y seguro, ideal para aplicaciones de teleconsulta. Cada componente está diseñado para trabajar de manera eficiente y autónoma, facilitando el mantenimiento y la escalabilidad del sistema. Las estrategias de seguridad y tolerancia a fallos aseguran que el sistema pueda operar de manera continua y fiable, incluso en situaciones adversas. Esta arquitectura proporciona una base sólida para el desarrollo y operación de aplicaciones de teleconsulta, garantizando una experiencia de usuario de alta calidad y un rendimiento óptimo del sistema.

4 MICROSERVICIO DE AUTENTICACIÓN Y GESTIÓN DE USUARIOS

4.1 Introducción al Microservicio

4.1.1 Descripción General:

El microservicio de autenticación y gestión de usuarios es un componente crítico dentro del sistema de teleconsulta. Su principal función es autenticar a los usuarios y especialistas mediante un sistema seguro que utiliza tokens JWT. Este servicio maneja las operaciones de registro e inicio de sesión, asegurando que solo los usuarios y especialistas autorizados puedan acceder a los recursos del sistema. En un futuro, este microservicio se expandirá para gestionar también la información de los usuarios, proporcionando un manejo integral de perfiles y roles dentro del sistema.

Las funcionalidades principales incluyen:

- Registro de Nuevos Usuarios y Especialistas: Permite que tanto los usuarios como los especialistas se registren en el sistema proporcionando sus datos personales. Las contraseñas se almacenan como Hash en base de datos.
- Autenticación de Usuarios y Especialistas: Verifica las credenciales y genera un token JWT para autenticación.
- Generación de Tokens JWT: Utiliza el algoritmo RS256 para crear tokens seguros, con una clave privada que se usa en la creación de los tokens y una clave pública que se usa para validarlos.

4.2 Arquitectura del Microservicio

4.2.1 Diagrama de Arquitectura

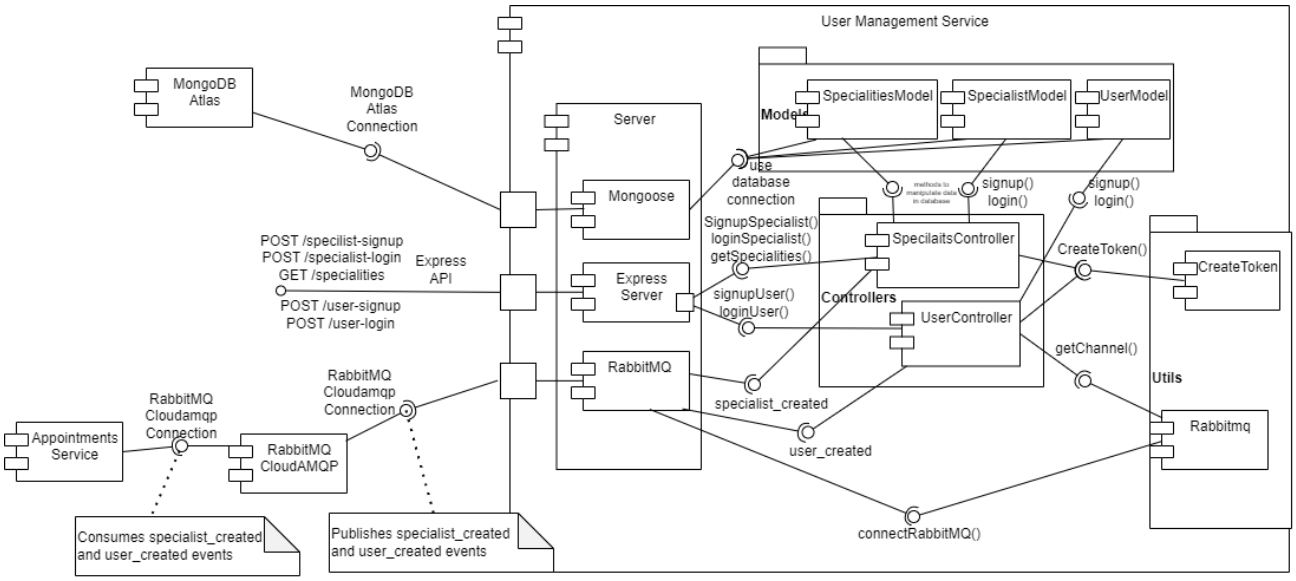


Figura 4-1. Arquitectura del microservicio de gestión de usuarios

El diagrama presentado es un diagrama de componentes que ilustra la arquitectura del microservicio de autenticación y gestión de usuarios, destacando los principales componentes y sus interacciones. El Server es el núcleo del microservicio, compuesto por tres subcomponentes: Mongoose, que facilita la conexión y gestión de la base de datos MongoDB Atlas; Express Server, que expone las rutas API necesarias para las operaciones de registro y autenticación; y RabbitMQ, que maneja la mensajería asíncrona para la comunicación entre microservicios.

Los Controladores (UserController y SpecialistController) gestionan las operaciones de registro e inicio de sesión. Estos interactúan con los Modelos de Datos definidos en Mongoose (userModel, specialistModel, specialitiesModel) para realizar operaciones CRUD. Además, los controladores utilizan las Utilidades (CreateToken para generar tokens JWT y RabbitMQ para gestionar eventos). Las rutas de la API (/user-signup, /user-login, /specialist-signup, /specialist-login, /specialities) están claramente definidas y conectadas a los controladores correspondientes.

El diagrama también resalta los Eventos de RabbitMQ (user_created y specialist_created), que son generados por las operaciones de registro y enviados a RabbitMQ. Estos eventos son consumidos por el microservicio de gestión de citas, lo que permite una integración fluida y comunicación efectiva entre microservicios.

4.2.2 Componentes Internos

- Server: El componente del servidor es el núcleo del microservicio. Inicializa el servidor Express, que sirve la API utilizada para interactuar con el servicio. Este servidor establece y gestiona las conexiones con dos elementos críticos: la base de datos MongoDB en MongoDB Atlas y el sistema de mensajería RabbitMQ. Mongoose se utiliza para facilitar la conexión y gestión de la base de datos MongoDB, permitiendo la manipulación de datos de manera eficiente y segura. Además, RabbitMQ se encarga de manejar la mensajería asíncrona, facilitando la comunicación entre diferentes microservicios dentro del sistema.
- Controladores: Los controladores son responsables de gestionar las operaciones de inicio de sesión y registro de usuarios y especialistas. Además, son los encargados de la generación de tokens JWT para la autenticación segura. Dentro de los controladores se encuentran dos componentes principales: UserController y SpecialistController. Estos controladores interactúan con los modelos de datos para

realizar operaciones de CRUD (Create, Read, Update, Delete) y utilizan utilidades específicas para la generación de tokens JWT y la conexión con RabbitMQ. Por simplicidad tenemos acciones de Creación a la hora de realizar un registro de usuario o especialista y de lectura cuando se realiza un acción de autenticación.

- Modelos de Datos: Los modelos de datos están definidos utilizando Mongoose y representan los esquemas de usuarios y especialistas (userModel y specialistModel). Cada modelo incluye métodos estáticos para las operaciones de login y signup. Estos métodos validan los datos de entrada y generan hashes seguros para las contraseñas utilizando algoritmos de hashing robustos. Los modelos facilitan la estructura y manipulación de los datos almacenados en MongoDB,
- Utilidades: Incluyen funciones para crear tokens JWT usando la librería "jsonwebtoken" y gestionar la conexión con RabbitMQ. Las colas RabbiMQ gestionan los eventos de creación de usuarios y especialistas enviando información acerca de estos al microservicio de gestión de citas. El microservicio de gestión de citas usa esta información para tener representaciones de usuarios y especialistas en su propia base de datos, separando de esta manera las bases de datos de ambos servicios y manteniendo el acoplamiento débil entre ambos.

4.3 Implementación

4.3.1 Lenguajes y Tecnologías Utilizadas

El microservicio de autenticación y gestión de usuarios está desarrollado en Node.js usando el Framework Express.js para desarrollar y servir las APIs. Como base de datos se usa MongoDB. Para mantener el acoplamiento débil entre servicios, se envía la información de los usuarios y especialistas creados hacia el microservicio de gestión de citas usando RabbitMQ.

4.3.2 Endpoints y APIs

El microservicio expone varias rutas API que permiten la interacción con el sistema de autenticación y gestión de usuarios. Los endpoints principales incluyen:

- POST /user-login: Autentica a los usuarios verificando sus credenciales. Si las credenciales son correctas, se genera y retorna un token JWT que el usuario puede utilizar para autenticarse en futuras solicitudes.
- POST /user-signup: Permite el registro de nuevos usuarios. Los datos del usuario se almacenan en la base de datos y se genera un token JWT.
- POST /specialist-login: Autentica a los especialistas verificando sus credenciales. Similar al login de usuarios, retorna un token JWT si la autenticación es exitosa.
- POST /specialist-signup: Permite el registro de nuevos especialistas. Al igual que el registro de usuarios, los datos se almacenan y se genera un token JWT.
- GET /specialities: Devuelve una lista de especialidades disponibles, permitiendo a los usuarios y especialistas consultar las especialidades registradas en el sistema.

4.3.3 Bases de Datos y Almacenamiento

La información de usuarios y especialistas se almacena en una base de datos MongoDB. Los esquemas de la base de datos están definidos utilizando Mongoose, una librería de modelado de datos para MongoDB en Node.js.

Esquema de Usuario (userModel):

- email: Correo electrónico del usuario (único).
- password: Contraseña cifrada utilizando Argon2, un algoritmo de hashing seguro.

- firstName: Nombre del usuario.
- lastName: Apellido del usuario.
- phone: Número de teléfono.
- dateOfBirth: Fecha de nacimiento.
- gender: Género del usuario.

Esquema de Especialista (specialistModel):

- email: Correo electrónico del especialista (único).
- password: Contraseña cifrada utilizando Argon2.
- firstName: Nombre del especialista.
- lastName: Apellido del especialista.
- phone: Número de teléfono.
- specialities: Referencias a las especialidades del especialista, permitiendo almacenar múltiples especialidades relacionadas con cada especialista.

Esquema de especialidad (specialityModel):

- name: Nombre de la especialidad
- description: Texto descriptivo de la especialidad

La utilización de Argon2 para el cifrado de contraseñas asegura que las contraseñas estén protegidas con un algoritmo de hashing robusto, proporcionando una capa adicional de seguridad. Además, la estructura flexible de MongoDB permite la fácil adaptación y expansión de los esquemas de datos conforme el sistema evoluciona.

4.3.4 Controladores

Los controladores son responsables de gestionar las solicitudes entrantes a las rutas API y de interactuar con los modelos de datos y utilidades para cumplir con las operaciones solicitadas. Existen dos controladores principales:

4.3.4.1 UserController

- signupUser(): Este método maneja el registro de nuevos usuarios. Recibe los datos del usuario desde la solicitud, valida los datos, y utiliza el modelo userModel para crear un nuevo registro en la base de datos. Luego, genera un token JWT utilizando la utilidad CreateToken y lo retorna en la respuesta.
- loginUser(): Este método maneja la autenticación de usuarios existentes. Recibe las credenciales del usuario, las verifica contra los datos almacenados en la base de datos utilizando el modelo userModel, y si son correctas, genera y retorna un token JWT.

4.3.4.2 SpecialistController

- signupSpecialist(): Similar al método de registro de usuarios, este método maneja el registro de nuevos especialistas. Valida los datos del especialista, los almacena en la base de datos utilizando el modelo specialistModel, genera un token JWT y lo retorna en la respuesta.
- loginSpecialist(): Maneja la autenticación de especialistas existentes. Verifica las credenciales contra los datos almacenados, y si son correctas, genera y retorna un token JWT.
- getSpecialities(): Este método permite obtener la lista de especialidades disponibles. Realiza una consulta a la base de datos utilizando el modelo specialitiesModel y retorna la lista de especialidades en la respuesta.

Estos controladores aseguran que todas las operaciones de registro y autenticación se realicen de manera segura

y eficiente, interactuando con los modelos de datos para almacenar y recuperar información y con las utilidades para la generación de tokens JWT. Además, manejan la publicación de eventos a RabbitMQ cuando se crean nuevos usuarios o especialistas, facilitando la integración y comunicación entre microservicios.

4.4 Comunicación con Otros Microservicios

4.4.1 Métodos de Comunicación

La comunicación entre este microservicio y otros componentes del sistema se realiza principalmente a través de APIs REST para operaciones síncronas y RabbitMQ para la mensajería asíncrona. Esto permite una interacción eficiente y fiable, manejando tanto solicitudes directas como eventos asíncronos.

4.4.2 Flujos de Datos

- Registro de Usuario y Especialista: Cuando un nuevo usuario o especialista se registra, se crea el usuario en la base de datos MongoDB, y a continuación se envía un mensaje a RabbitMQ para notificar al microservicio de gestión de citas, excluyendo la contraseña. Los eventos generados son “user_created” y “specialist_created” respectivamente.
- Autenticación: Durante la autenticación, se verifica las credenciales y se genera un token JWT. Este token se utiliza para acceder a otros microservicios, garantizando que solo usuarios autenticados puedan realizar solicitudes.

4.5 Seguridad

4.5.1 Mecanismos de Autenticación y Autorización

Se utiliza JWT para autenticar a los usuarios y especialistas. Tras el registro o la autenticación exitosas, se genera un token JWT que se envía al cliente. Este token se incluye en la cabecera “Authorization” de las solicitudes HTTP como Bearer Token para acceder a recursos protegidos. La validación del token asegura que solo los usuarios autorizados puedan interactuar con el sistema.

El payload del token JWT (JSON Web Token) que se genera en el microservicio de autenticación y gestión de usuarios generalmente incluye información relevante sobre el usuario o especialista que se ha autenticado. Esta información es útil para identificar al usuario y gestionar sus permisos dentro del sistema. Basado en el código proporcionado, aquí está el formato del payload para los tokens de usuario y especialista incluyendo los claims, vemos que el token tiene tres claims, id, email y role, que sirven para identificar el usuario que realiza cualquier acción en el sistema y su rol.

Cuando un usuario se autentica o se registra, el payload del token JWT contiene los siguientes campos:

```
{
  "id": "user_id",
  "email": "user_email",
  "role": "user"
}
```

Cuando un especialista se autentica o se registra, el payload del token JWT contiene los siguientes campos:

```
{
  "id": "specialist_id",
  "email": "specialist_email",
  "role": "specialist"
}
```

4.5.2 Protección de Datos

La seguridad en el microservicio de autenticación y gestión de usuarios se aborda mediante la implementación

de técnicas avanzadas para proteger tanto los datos almacenados como los datos en tránsito. Las contraseñas de los usuarios y especialistas se cifran utilizando Argon2, un algoritmo de hashing seguro que ofrece alta resistencia a ataques de fuerza bruta y hardware especializado. Este algoritmo genera hashes robustos que se almacenan en la base de datos MongoDB, asegurando que incluso si la base de datos fuera comprometida, las contraseñas no podrían ser descifradas fácilmente.

Además, toda la comunicación entre el frontend y los microservicios se realiza a través de HTTPS (HyperText Transfer Protocol Secure), que utiliza el protocolo SSL/TLS para cifrar los datos transmitidos. Esta medida garantiza que la información sensible, como credenciales de inicio de sesión y tokens JWT, se transmita de manera segura, protegiéndola contra interceptaciones y ataques de tipo man-in-the-middle. Para implementar HTTPS en el despliegue del microservicio en un clúster de Kubernetes, se utiliza un certificado SSL/TLS generado por Amazon Certificate Manager (ACM). Este certificado se asigna al balanceador de cargas correspondiente a este servicio, permitiendo que el tráfico HTTPS se redirija de manera segura al dominio video.sehha-telemedicine.com.

4.6 Escalabilidad y Tolerancia a Fallos

El microservicio de autenticación y gestión de usuarios se despliega en contenedores Docker y se orquesta mediante Kubernetes en un clúster de Amazon EKS (Elastic Kubernetes Service). Kubernetes permite la escalabilidad automática del microservicio basándose en la carga de trabajo, asegurando que el sistema pueda responder eficientemente a aumentos en la demanda. Esta capacidad de escalado dinámico optimiza el uso de recursos y garantiza que el servicio se mantenga operativo incluso durante picos de alta demanda.

Kubernetes también se encarga de reiniciar automáticamente los contenedores que fallan, asegurando la disponibilidad continua del servicio. Para exponer el servicio al exterior, se utiliza un balanceador de carga de AWS, configurado desde Kubernetes, que distribuye el tráfico entrante de manera uniforme y asegura que las solicitudes lleguen a los pods disponibles. Además, RabbitMQ facilita la entrega de mensajes incluso si algún microservicio no está disponible temporalmente, contribuyendo a la resiliencia del sistema y manteniendo la comunicación efectiva entre los diferentes microservicios.

4.7 Pruebas

Las pruebas se realizan enviando peticiones con Postman, a continuación, se muestran algunas de las pruebas realizadas para el registro y autenticación de un usuario.

4.7.1 Registro de Nuevo Usuario

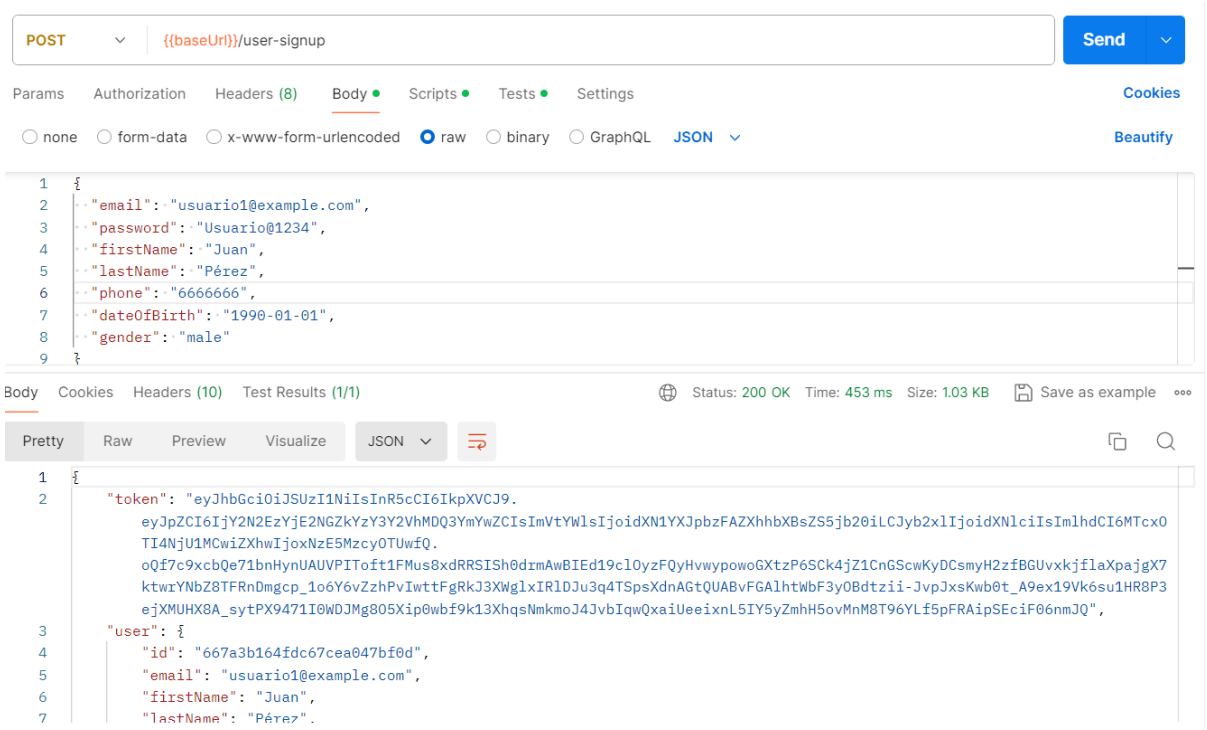


Figure-4-2. Registro de nuevo usuario

Este caso de prueba verifica el proceso de registro de un nuevo usuario en el sistema. Al enviar los datos requeridos del usuario, el sistema debería crear un nuevo registro en la base de datos y devolver un token JWT que el usuario puede utilizar para autenticarse en futuras solicitudes.

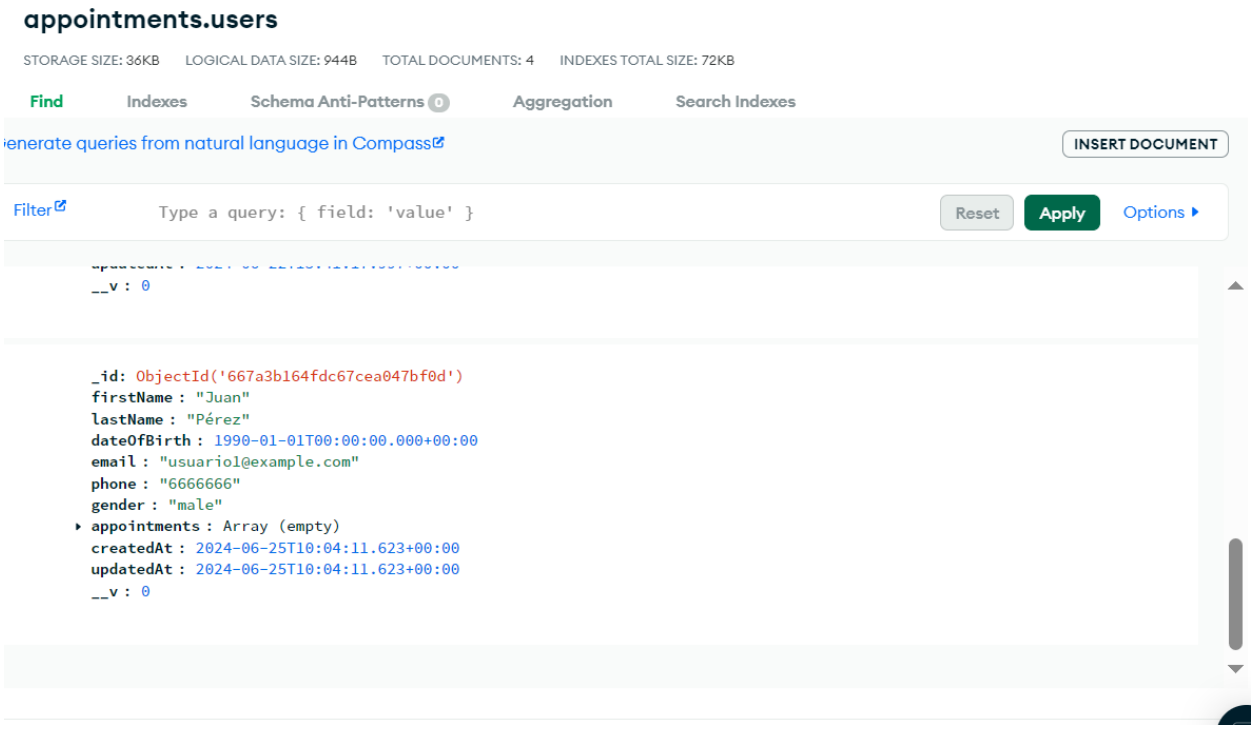


Figure-4-3. Usuario creado en la base de datos

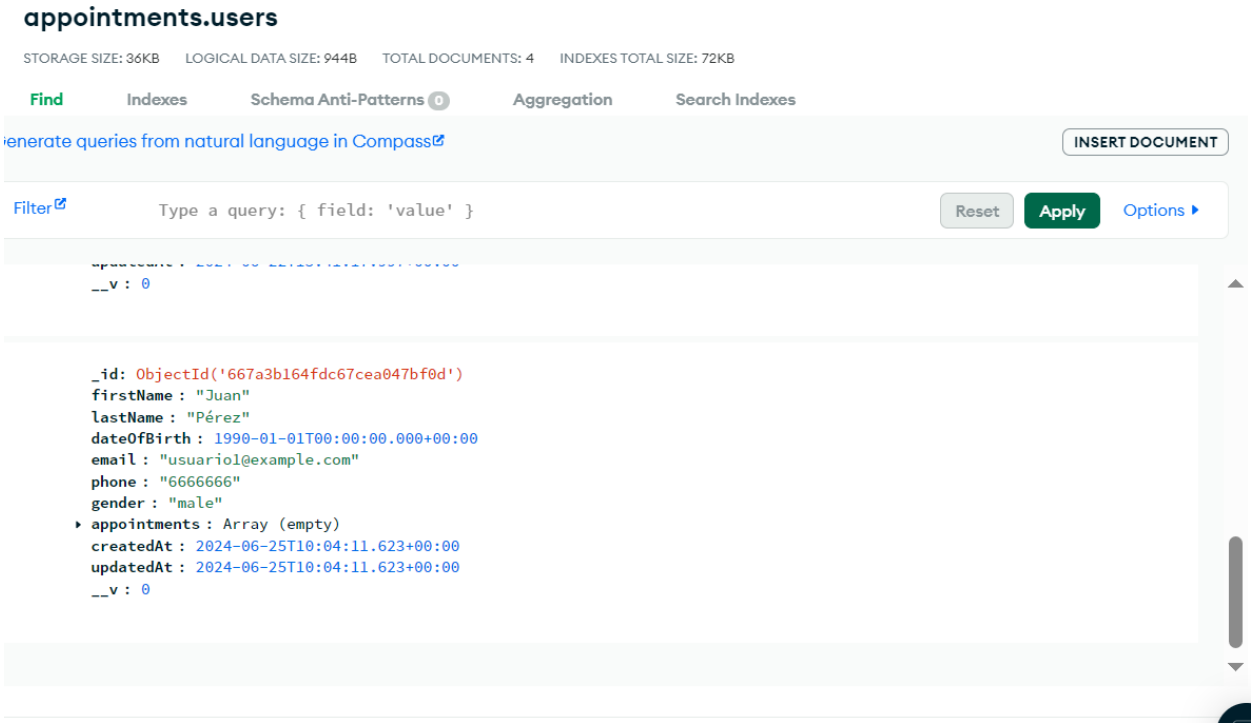


Figure-4-4. Usuario creado en la base de datos de citas

En la figura 4-3 se observa que el usuario ha sido creado en la base de datos del microservicio de gestión de citas, por otra parte en la figura 4-4, se observa que se ha creado en la base de datos del servicio de gestión de citas gracias a la comunicación de los datos de usuario creado a través de RabbitMQ.

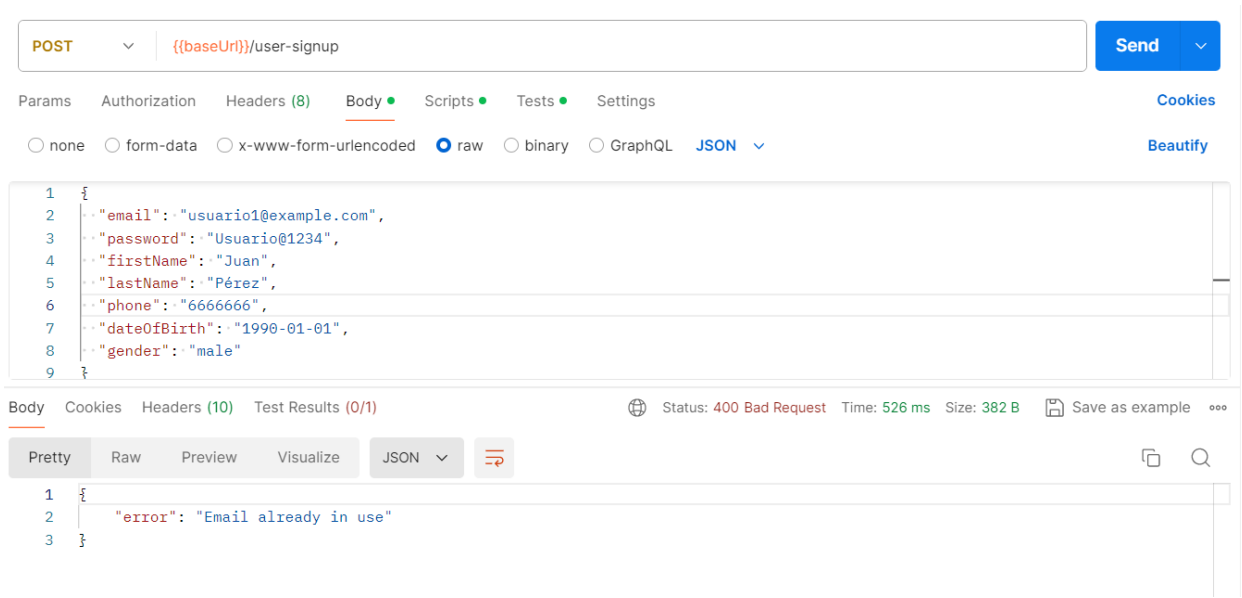


Figure-4-5. Error: Correo ya está en uso

En la figura anterior observamos que si se lanza una petición de creación de usuario con el correo electrónico que ha sido usado anteriormente para crear el usuario, se recibe un error “400 Bad Request” en el que se recibe un mensaje de error indicando que el correo electrónico se encuentra en uso.

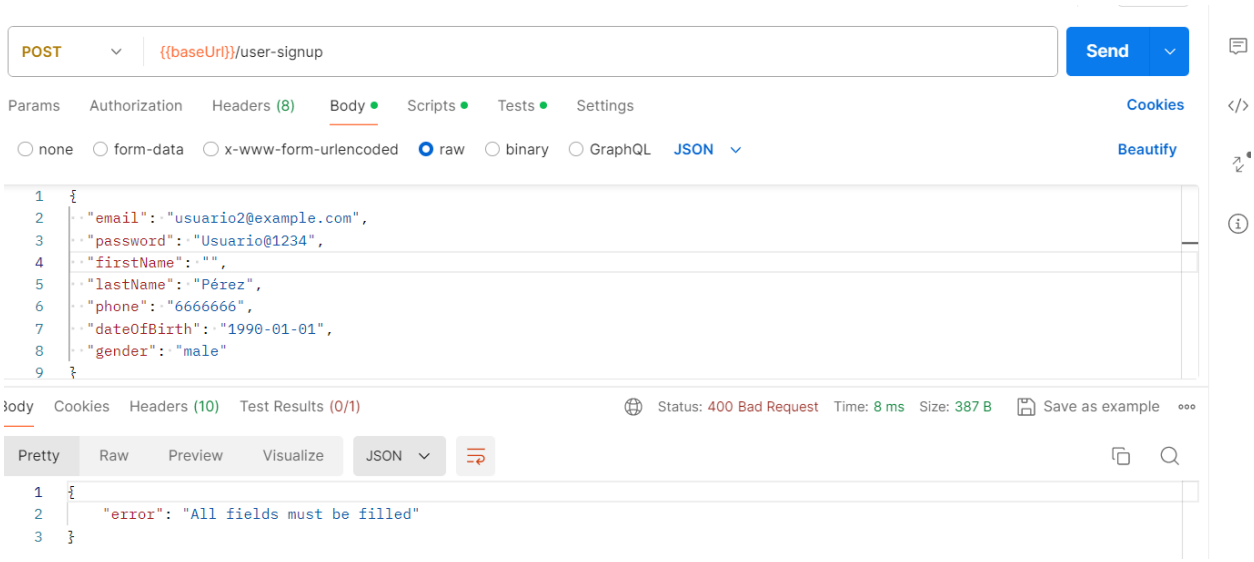


Figure-4-6. Error: Correo ya está en uso

A la hora de lanzar una petición, todos los campos deben tener un valor, por lo tanto, si se envía una petición con un campo vacío, se devuelve el siguiente error “400 Bad request” indicando que todos los campos deben ser rellenados.

4.7.2 Inicio de Sesión de Usuario

Este caso de prueba valida el proceso de autenticación de usuarios existentes. Al enviar las credenciales del usuario, el sistema debería verificar su autenticidad y devolver un token JWT si las credenciales son correctas.

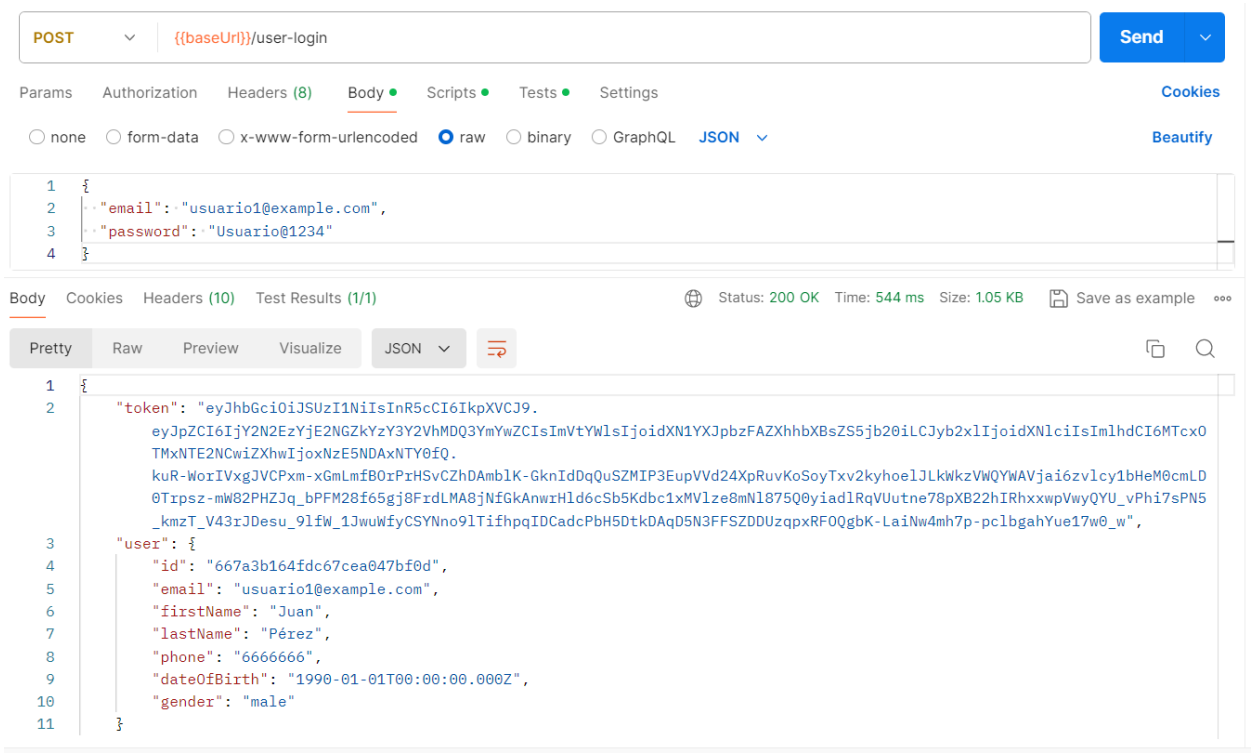


Figure-4-7. Autenticación exitosa de un usuario

Enviando las credenciales que hemos utilizado anteriormente para registrar un nuevo usuario, observamos en la figura 4-7 que se puede autenticar correctamente. Se recibe el token JWT que se puede utilizar para la gestión de sesiones y autorización en los distintos servicios.

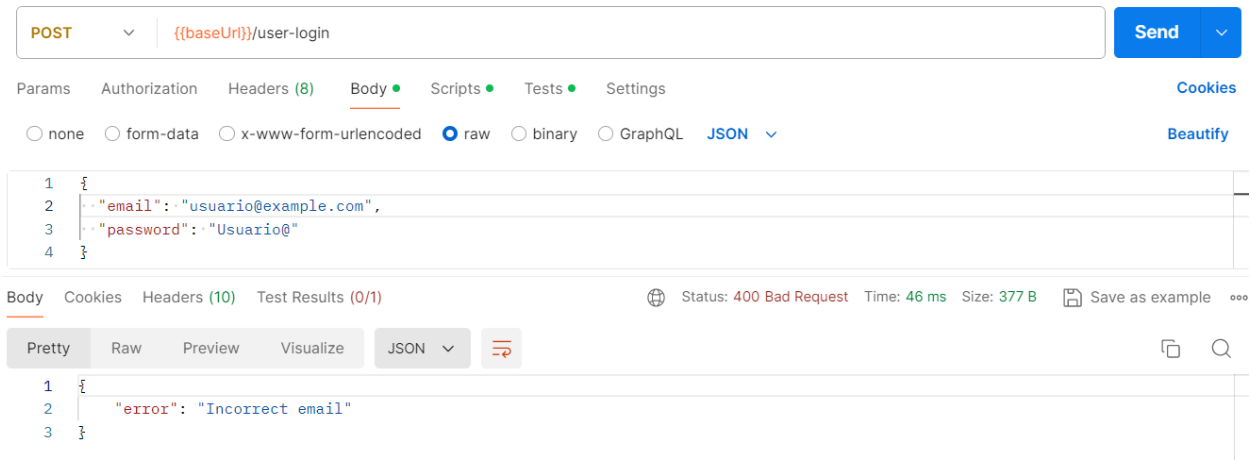


Figure-4-8. Autenticación no exitosa del usuario, correo incorrecto

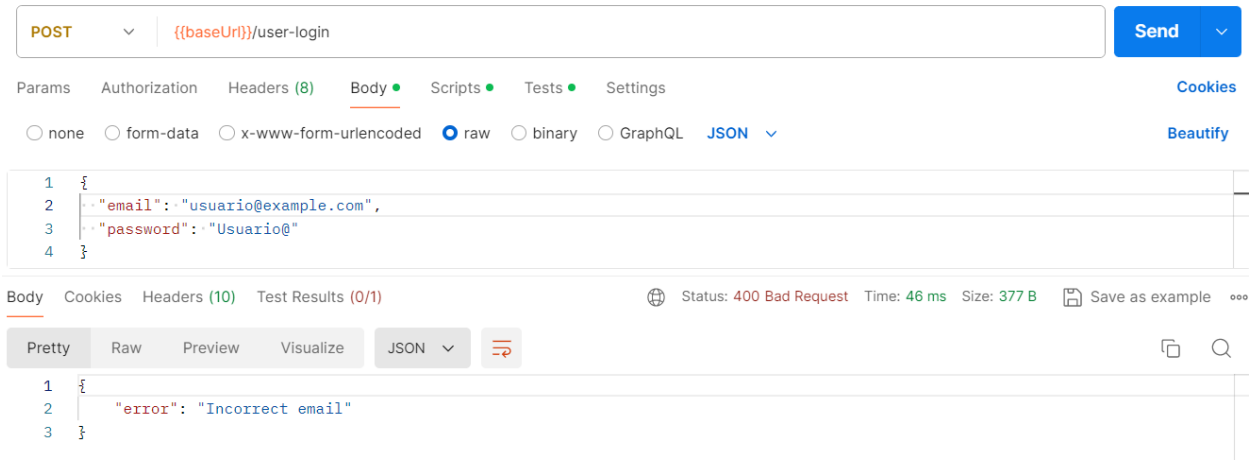


Figure-4-9. Autenticación no exitosa del usuario. Contraseña incorrecta.

Si se introduce un correo electrónico que no ha sido registrado anteriormente o se introduce una contraseña errónea, para un usuario que sí ha sido registrado anteriormente, se produce un error como el que aparece en las figuras 4-8 y 4-9 respectivamente.

4.8 Conclusión

El microservicio de autenticación y gestión de usuarios está diseñado para ser seguro, escalable y fácil de mantener. Utiliza tecnologías modernas y sigue buenas prácticas de desarrollo para asegurar la autenticación y protección de datos de los usuarios. Con la expansión futura para incluir la gestión de perfiles de pacientes y especialistas, este microservicio jugará un papel aún más crucial en la arquitectura del sistema SEHHA, facilitando una administración integral de los usuarios y sus roles.

5 MICROSERVICIO DE GESTIÓN DE CITAS

5.1 Introducción al Microservicio

5.1.1 Descripción General

El microservicio de gestión de citas es un componente esencial dentro del sistema de SEHHA. Su principal función es gestionar las citas entre usuarios (pacientes) y especialistas, permitiendo a los pacientes reservar y cancelar citas, y a los especialistas gestionar sus horarios mediante la creación de bloques de tiempo disponibles. Estas citas son fundamentales para la coordinación de videollamadas en las sesiones de teleconsulta, donde se identifican a los participantes de cada cita. Actualmente, el sistema maneja dos tipos de usuarios: pacientes y especialistas que participan en las videoconsultas. Las citas son incluyen un paciente y un especialista con la posibilidad de incluir más de dos participantes en una cita en el futuro.

Las funcionalidades principales incluyen:

- Gestión de Horarios de Especialistas: Los especialistas pueden crear y gestionar bloques de tiempo (time slots) en sus calendarios.
- Reserva y Cancelación de Citas: Los pacientes pueden reservar y cancelar citas dentro de los bloques de tiempo disponibles creados por los especialistas.
- Integración para Videollamadas: Las citas gestionadas por este microservicio son utilizadas para identificar a los participantes de las sesiones de videollamadas.

5.2 Arquitectura del Microservicio

5.2.1 Introducción a la Arquitectura CLEAN

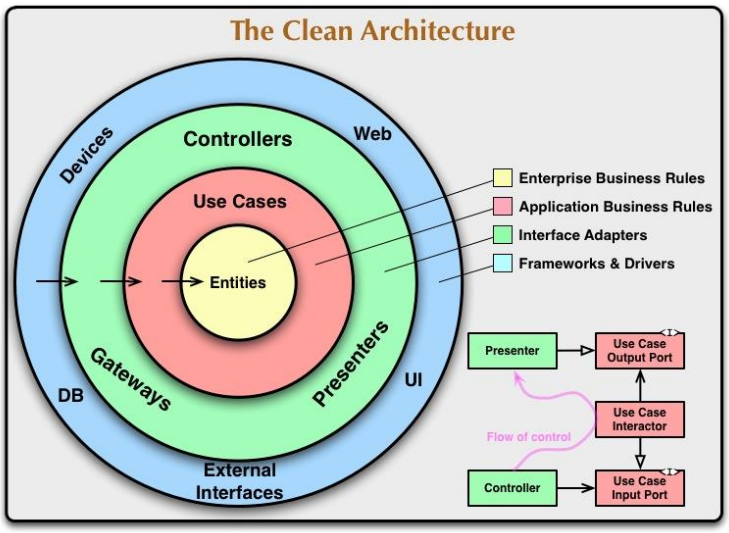


Figure 5-1. Arquitectura CLEAN por Robert C. Martin

La arquitectura CLEAN [21], propuesta por Robert C. Martin (también conocido como "Uncle Bob"), es un enfoque de diseño de software que busca producir sistemas que sean mantenibles, testeables y de alta calidad. Esta arquitectura organiza el código en niveles claramente definidos, cada uno con responsabilidades distintas y bien separadas. La idea principal detrás de CLEAN es que el diseño del software debe permitir el cambio fácil y la evolución del sistema sin comprometer su integridad.

Al ser el microservicio de gestión de citas un servicio clave para el negocio de SEHHA, esta arquitectura se usa solamente para este servicio. El resto se de servicio tienen un rol de soporte a este servicio, por lo tanto no se usa esta arquitectura para dichos servicios por simplicidad.

5.2.2 Componentes de la Arquitectura CLEAN

- Entidades: Son las clases que representan las reglas de negocio de más alto nivel. Estas reglas son universales para el sistema y no cambian a menudo.
- Casos de Uso: Contienen la lógica específica de la aplicación y orquestan la interacción entre las entidades y los repositorios. Los casos de uso encapsulan y coordinan el flujo de datos hacia y desde las entidades.
- Controladores: Reciben las entradas del usuario y dirigen estas entradas a los casos de uso apropiados. Los controladores son responsables de gestionar las solicitudes y respuestas del sistema, y de llamar a los casos de uso para ejecutar la lógica de negocio.
- Repositorios: Manejan la lógica de acceso a datos. Los repositorios proporcionan una abstracción sobre la forma en que los datos son almacenados y recuperados.
- Gateways y Drivers: Son los mecanismos mediante los cuales el sistema interactúa con agentes externos. Estos componentes incluyen el acceso a bases de datos, sistemas de mensajería, servicios externos, etc

5.2.3 Principios de la Arquitectura CLEAN

- Independencia de Frameworks: Los frameworks son herramientas útiles, pero no deberían dictar el diseño del sistema. El sistema debe ser independiente de los detalles de implementación que los frameworks proporcionan.

- Testabilidad: El sistema debe ser fácil de probar. Esto se logra separando la lógica de negocio de la infraestructura de tal manera que las pruebas unitarias y de integración puedan llevarse a cabo de forma aislada.
- Independencia de la UI: La lógica del negocio debe ser independiente de la interfaz de usuario. Los cambios en la UI no deberían afectar el núcleo del sistema.
- Independencia de la base de datos: La lógica del negocio no debe depender de la base de datos. El acceso a los datos debe estar abstraído de tal manera que pueda cambiarse fácilmente sin afectar el resto del sistema.
- Independencia de agentes externos: El sistema debe ser independiente de cualquier sistema externo con el que se comunique. Esto facilita el mantenimiento y la evolución del sistema sin depender de estos agentes.

5.2.4 Componentes Internos

Para la implementación del código se sigue la arquitectura CLEAN, que garantiza un código mantenible, testeable y de alta calidad, por lo tanto la estructura de este servicio sigue una estructura similar a la mostrada en la mostrada en la figura 5-1. Los componentes de esta arquitectura incluyen:

5.2.4.1 Server

El componente del servidor es el punto de inicio del microservicio. Inicializa el servidor Express, que sirve la API utilizada para interactuar con el servicio. Este servidor establece y gestiona las conexiones con dos elementos críticos: la base de datos MongoDB en MongoDB Atlas y el sistema de mensajería RabbitMQ. Mongoose se utiliza para facilitar la conexión y gestión de la base de datos MongoDB, permitiendo la manipulación de datos de manera eficiente y segura. RabbitMQ maneja la mensajería asíncrona, facilitando la comunicación entre diferentes microservicios dentro del sistema.

5.2.4.2 Entidades

Las entidades representan los objetos de negocio y contienen la lógica empresarial esencial. Las principales entidades en el microservicio de gestión de citas son:

- Appointment: Representa las citas programadas entre usuarios y especialistas.
- Schedule: Representa los horarios de los especialistas.
- ScheduleSlot: Representa los bloques de tiempo disponibles en los horarios de los especialistas.
- Specialist: Representa a los especialistas que ofrecen consultas médicas.
- Speciality: Representa las especialidades médicas disponibles.
- User: Representa a los usuarios (pacientes) que utilizan la plataforma.

5.2.4.3 Casos de Uso

Los casos de uso contienen la lógica de aplicación específica y coordinan la interacción entre las entidades y los repositorios. Ejemplos de casos de uso son:

- createNewAppointment: Gestiona la lógica para la creación de nuevas citas.
- addNewSpeciality: Maneja la adición de nuevas especialidades médicas.
- updateSlot: Coordina la actualización de bloques de tiempo en los horarios de los especialistas.
- cancelAppointment: Maneja la lógica para la cancelación de citas.
- retrieveAppointment: Recupera la información detallada de una cita específica.

5.2.4.4 Controladores

Los controladores son responsables de gestionar las operaciones de reserva y cancelación de citas, así como la gestión de horarios de los especialistas. Hacen uso de los casos de uso para implementar la lógica de negocio. Existen varios controladores principales:

- Controladores de citas: Manejan la creación, cancelación y obtención de citas.
- Controladores de especialistas: Gestiona las operaciones relacionadas con los especialistas, incluyendo la creación y gestión de sus horarios.
- Controladores de usuarios: Gestiona las operaciones relacionadas con los usuarios (pacientes).
- Controladores de horarios: Maneja la creación, actualización y eliminación de bloques de tiempo en los horarios de los especialistas.
- Controladores de especialidades: Gestiona la creación, actualización y eliminación de especialidades médicas.

5.2.4.5 Repositorios

Son responsables de la comunicación con la base de datos. Cada entidad tiene su repositorio correspondiente que realiza las operaciones CRUD. Los modelos de datos están definidos utilizando Mongoose y representan los esquemas de citas, horarios, bloques de tiempo, especialistas, especialidades y usuarios. Cada modelo incluye métodos estáticos para las operaciones CRUD, validando los datos de entrada y asegurando la integridad de los datos almacenados en MongoDB:

- appointmentModel: Representa las citas entre usuarios y especialistas.
- scheduleModel: Representa los horarios de los especialistas.
- scheduleSlotModel: Representa los bloques de tiempo disponibles en los horarios de los especialistas.
- specialistModel: Representa a los especialistas.
- specialityModel: Representa las especialidades médicas.
- userModel: Representa a los usuarios (pacientes).

5.2.4.6 Utilidades

Las utilidades incluyen funciones para gestionar la conexión con RabbitMQ y la publicación de eventos. Las colas RabbitMQ gestionan los eventos de creación de citas, enviando información al microservicio de gestión de videollamadas, lo que permite identificar a los participantes en las sesiones de teleconsulta. Esto asegura un acoplamiento débil entre los microservicios y una comunicación eficiente.

5.2.4.7 Frameworks y Drivers

Los frameworks y drivers proporcionan implementaciones concretas para la infraestructura, como el acceso a la base de datos y la mensajería. En este microservicio, se utilizan los siguientes:

- Mongoose: Para la gestión de la base de datos MongoDB, proporcionando un esquema para modelar los datos y métodos para interactuar con la base de datos.
- Express: Como framework para construir el servidor y las APIs RESTful.
- RabbitMQ: Para la mensajería asíncrona, facilitando la comunicación entre los microservicios mediante colas de mensajes.

5.3 Implementación

5.3.1 Lenguajes y Tecnologías Utilizadas

El microservicio de gestión de citas está desarrollado en Node.js, utilizando Express.js como framework principal para construir la API REST.

5.3.2 Endpoints y APIs

El microservicio expone varias rutas API que permiten la interacción con el sistema de gestión de citas. Los endpoints principales incluyen:

Ruta para la gestión de citas:

- POST /appointments: Permite la creación de nuevas citas por parte de los usuarios (pacientes) en los bloques de tiempo disponibles.
- DELETE /appointments/:id: Permite la cancelación de citas existentes por parte de los usuarios (pacientes).
- GET /appointments: Devuelve una lista de todas las citas existentes, filtradas por criterios específicos.
- GET /appointments/:id: Devuelve los detalles de una cita específica.

Rutas para la gestión de especialidades:

- GET /specialities: Devuelve una lista de especialidades médicas disponibles.
- POST /specialities: Permite la creación de nuevas especialidades médicas.
- GET /specielities/:id: Devuelve los detalles de una especialidad usando su ID.
- PUT /specialities/:id: Permite la actualización de especialidades médicas existentes.
- DELETE /specialities/:id: Permite la eliminación de especialidades médicas existentes.

Rutas para la gestión de usuarios:

- GET /users/:id: Devuelve los detalles de un usuario específico.
- GET /users: Devuelve una lista de todos los usuarios existentes.

Rutas para la gestión de especialistas:

- GET /specialists: Devuelve una lista de todos los especialistas existentes.
- GET /specialists/:id: Devuelve los detalles de un especialista específico.
- GET /specialists/:id/schedule: Devuelve el horario de un especialista específico.
- POST /specialists/:id/schedule: Permite a los especialistas crear nuevos bloques de tiempo en sus horarios.
- PUT /specialists/:id/schedule/slots/:slotId: Permite la actualización de bloques de tiempo específicos.
- DELETE /specialists/:id/schedule/slots/:slotId: Permite la eliminación de bloques de tiempo específicos.

5.3.3 Bases de Datos y Almacenamiento

La información de citas, horarios, bloques de tiempo, especialistas, especialidades y usuarios se almacena en una base de datos MongoDB. Los esquemas de la base de datos están definidos utilizando Mongoose, una librería de modelado de datos para MongoDB en Node.js.

Esquema de Citas (appointmentModel):

- user: Referencia al usuario que reservó la cita.
- specialist: Referencia al especialista con quien se reservó la cita.
- speciality: Referencia a la especialidad médica de la cita.
- status: Estado de la cita (programada, completada, cancelada).
- description: Descripción de la cita.
- slot: Referencia al bloque de tiempo reservado para la cita.
- createdAt: Fecha de creación de la cita.
- updatedAt: Fecha de última actualización de la cita.

Esquema de Horarios (scheduleModel):

- specialist: Referencia al especialista propietario del horario.
- slots: Lista de bloques de tiempo disponibles.
- createdAt: Fecha de creación del horario.
- updatedAt: Fecha de última actualización del horario.

Esquema de Bloques de Tiempo (scheduleSlotModel):

- startTime: Hora de inicio del bloque de tiempo.
- endTime: Hora de fin del bloque de tiempo.
- appointment: Referencia a la cita reservada en este bloque de tiempo.
- createdAt: Fecha de creación del bloque de tiempo.
- updatedAt: Fecha de última actualización del bloque de tiempo.

Esquema de Especialistas (specialistModel):

- firstName: Nombre del especialista.
- lastName: Apellido del especialista.
- contactInfo: Información de contacto del especialista (correo electrónico, teléfono).
- specialities: Referencias a las especialidades del especialista.
- schedule: Referencia al horario del especialista.
- createdAt: Fecha de creación del registro del especialista.
- updatedAt: Fecha de última actualización del registro del especialista.

Esquema de Especialidades (specialityModel):

- name: Nombre de la especialidad.
- description: Descripción de la especialidad.
- createdAt: Fecha de creación de la especialidad.
- updatedAt: Fecha de última actualización de la especialidad.

Esquema de Usuarios (userModel):

- firstName: Nombre del usuario.
- lastName: Apellido del usuario.
- dateOfBirth: Fecha de nacimiento del usuario.
- email: Correo electrónico del usuario.
- phone: Número de teléfono del usuario.
- gender: Género del usuario.
- appointments: Lista de citas del usuario.
- createdAt: Fecha de creación del registro del usuario.
- updatedAt: Fecha de última actualización del registro del usuario.

5.4 Comunicación con Otros Microservicios

5.4.1 Métodos de Comunicación

La comunicación entre este microservicio y otros componentes del sistema se realiza principalmente a través de APIs REST para operaciones síncronas y RabbitMQ para la mensajería asíncrona. Esto permite una interacción eficiente y fiable, manejando tanto solicitudes directas como eventos asincrónicos.

5.4.2 Flujos de Datos

Creación de Citas: Cuando se crea una nueva cita, se envía un mensaje a RabbitMQ mediante un evento cuyo nombre es “appountment_created” para notificar al microservicio de gestión de videollamadas, facilitando la integración de la cita en las sesiones de teleconsulta.

Creación de usuarios y especialistas: Cuando se crean usuarios y especialistas en el microservicio de gestión de usuarios, se crean los eventos “user_created” y “specialist_created” que notifican al microservicio de la creación de estos usuarios. Se envían mensajes a través de colas RabbitMQ para comunicar esta información al microservicio de gestión de citas.

5.5 Seguridad

5.5.1 Mecanismos de Autenticación y Autorización

Se utiliza JWT para autenticar a los usuarios y especialistas. Tras la autenticación exitosa, se genera un token JWT en el microservicio de gestión de usuarios que se envía al cliente. Este token se incluye en la cabecera "Authorization" de las solicitudes como Bearer Token para acceder a recursos protegidos del microservicio de gestión de citas. El token se valida mediante la clave pública simétrica a la clave privada que se ha utilizado para firmar el token a la hora de su creación en el microservicio de gestión de usuarios. El token asegura que solo los usuarios autorizados puedan interactuar con el sistema. Los roles y permisos se manejan mediante RBAC (Role-Based Access Control), verificando los roles específicos de los usuarios en las rutas protegidas. El sistema utiliza middleware para verificar los roles de los usuarios en rutas específicas. Este middleware se asegura de que solo los usuarios con los roles adecuados puedan acceder a ciertas funcionalidades. Por ejemplo, para crear una nueva especialidad médica, solo los usuarios con el rol de "especialista" tienen permiso. Para ver la lista de especialidades, tanto "usuarios" como "especialistas" pueden tener acceso.

Por lo tanto, el middleware de autenticación (authenticateToken) se asegura de que cada solicitud incluya un token válido. Si no se proporciona un token, o si el token es inválido, la solicitud es rechazada con un código de estado 401 (No autorizado). Mientras tanto, el middleware de autorización (authorizeRoles) verifica que el usuario tenga los permisos necesarios para acceder a la ruta solicitada. Si el usuario no tiene el rol requerido, la solicitud es rechazada con un código de estado 403 (Prohibido).

5.5.2 Protección de Datos

Toda la comunicación entre el frontend y los microservicios se realiza a través de HTTPS, utilizando certificados SSL/TLS generados por Amazon Certificate Manager. Esto garantiza que la información sensible de los usuarios se transmita de manera segura. El dominio utilizado para acceder al microservicio es book.sehha-telemedicine.com, y todas las peticiones se gestionan de manera segura a través de este dominio.

5.6 Escalabilidad y Tolerancia a Fallos

El microservicio de gestión de citas se despliega en contenedores Docker y se orquesta mediante Kubernetes en un clúster de Amazon EKS (Elastic Kubernetes Service). Kubernetes permite la escalabilidad automática del microservicio basándose en la carga de trabajo, asegurando que el sistema pueda responder eficientemente a aumentos en la demanda. Kubernetes también se encarga de reiniciar automáticamente los contenedores que fallan, asegurando la disponibilidad continua del servicio. Para exponer el servicio al exterior, se utiliza un balanceador de carga de AWS, configurado desde Kubernetes, que distribuye el tráfico entrante de manera uniforme y asegura que las solicitudes lleguen a los pods disponibles. RabbitMQ facilita la entrega de mensajes incluso si algún microservicio no está disponible temporalmente, contribuyendo a la resiliencia del sistema y manteniendo la comunicación efectiva entre los diferentes microservicios.

5.7 Pruebas

Las pruebas se enfocarán en verificar la funcionalidad completa del sistema mediante la realización de pruebas contra la API, para eso es necesario generar los tokens de autenticación para un usuario y un especialista. El flujo de pruebas incluye la creación de una cita por parte del especialista y la reserva de dicha cita por parte del usuario, este flujo es solo una demostración de como se han realizado las pruebas.

5.7.1 Autenticación de usuario y especialista

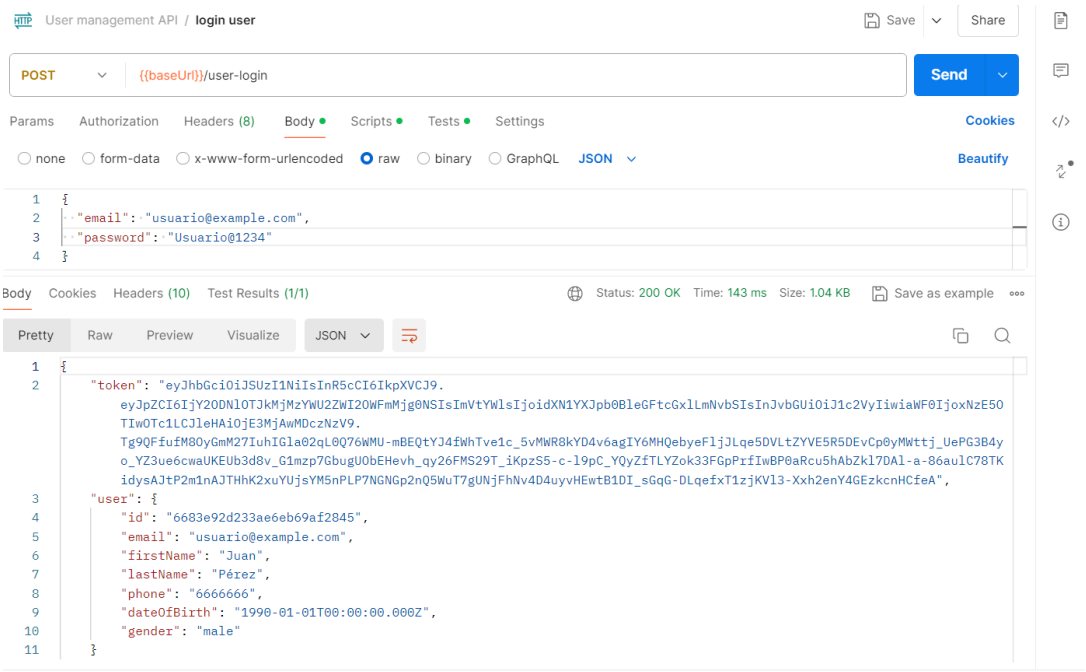


Figura 5-2. Autenticación del usuario y obtención del token JWT

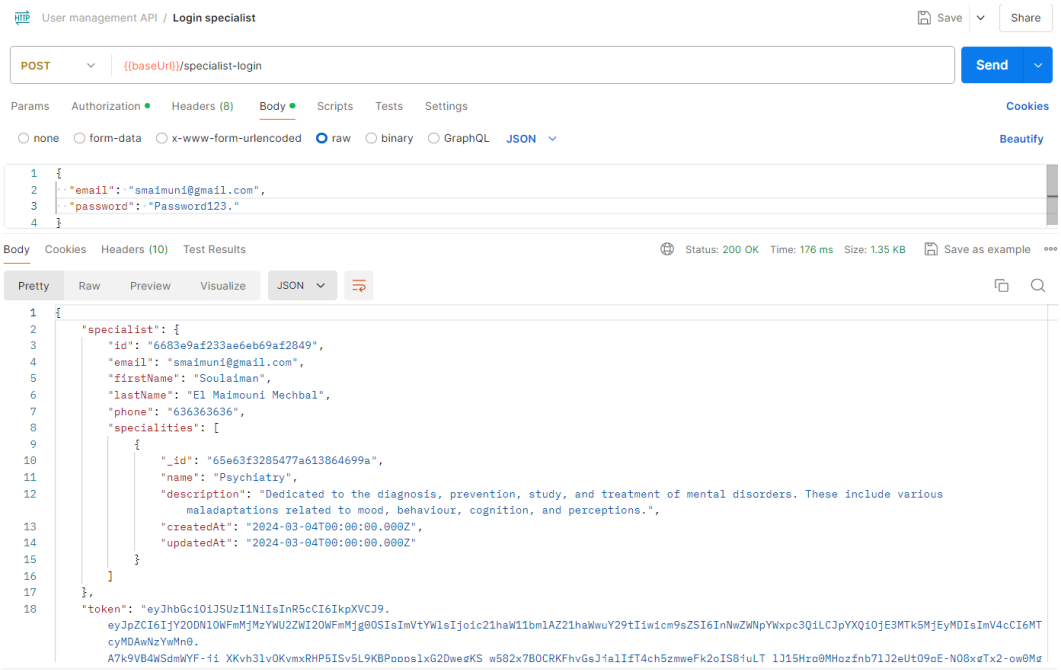


Figura 5-3. Autenticación del especialista y obtención del token JWT

A la hora de autenticarse, se reciben los tokens que sirven para la autenticación y autorización de los usuarios de SEHHA. Estos tokens irán la cabecera "Authorization" como "Bearer tokens" en las subsiguientes peticiones al microservicio de citas para la creación de citas por parte de los especialistas y la reserva de citas por parte de los usuarios

5.7.2 Creación de bloques de tiempo y reserva de citas

Para la creación de los bloques de tiempo que pueden ser reservados por usuarios, los especialistas envían una petición de tipo POST como se muestra en la figura

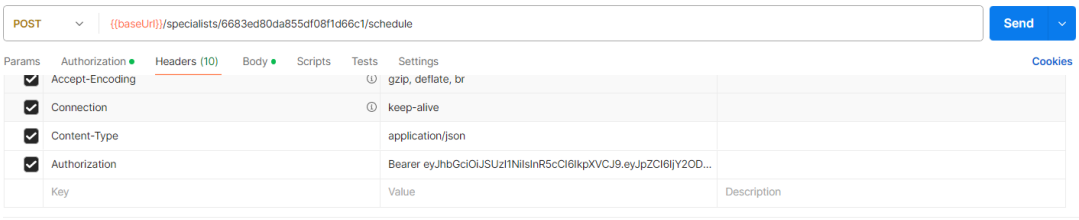


Figure 5-4. Bearer token generado por el especialista enviado en una cabecera Authorization

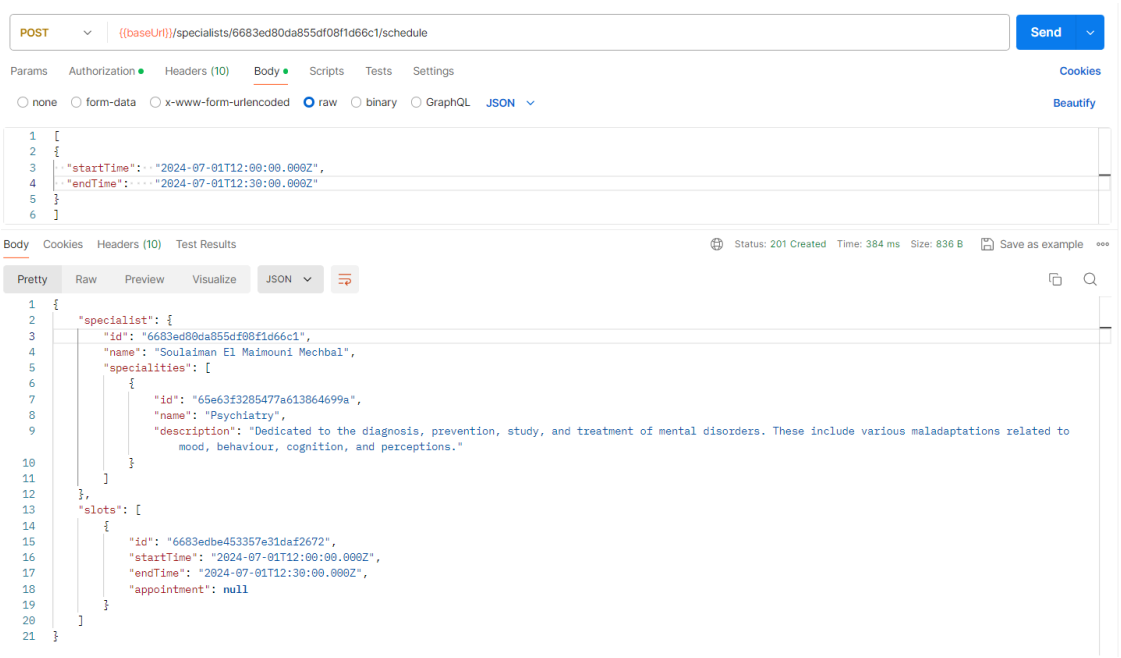


Figure 5-5. Creación exitosa del bloque de tiempo

Una vez creado el bloque de tiempo, se puede reservar la cita por parte de usuario. En caso de que se envíe una petición sin el token generado por el usuario, se produce error “401 Unauthorized” como se muestra en la figura siguiente:

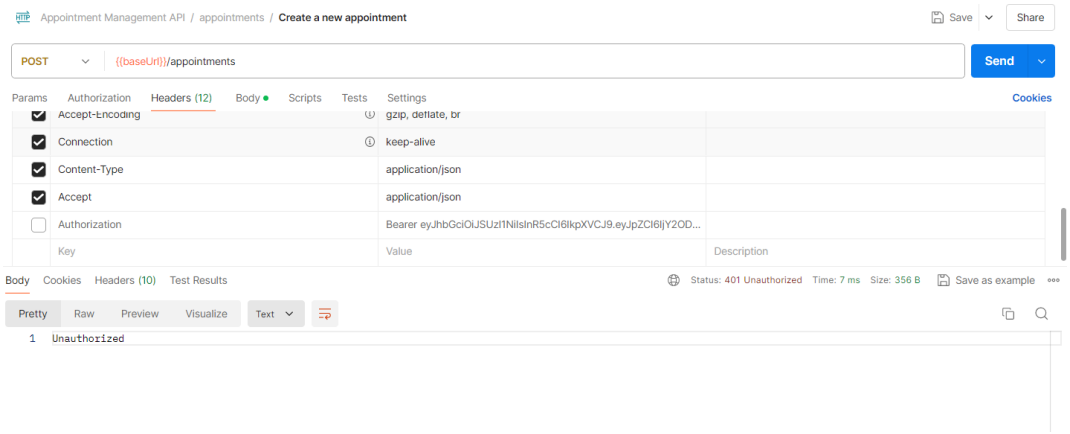


Figure 5-6. Error de autorización si no se envía el token

Cuando se habilita la cabecera Authorization con el Bearer token generado en la autenticación del usuario y se envía la petición para la reserva de la cita, se observa que la cita se crea correctamente, devolviendo un “201 Created” como respuesta.

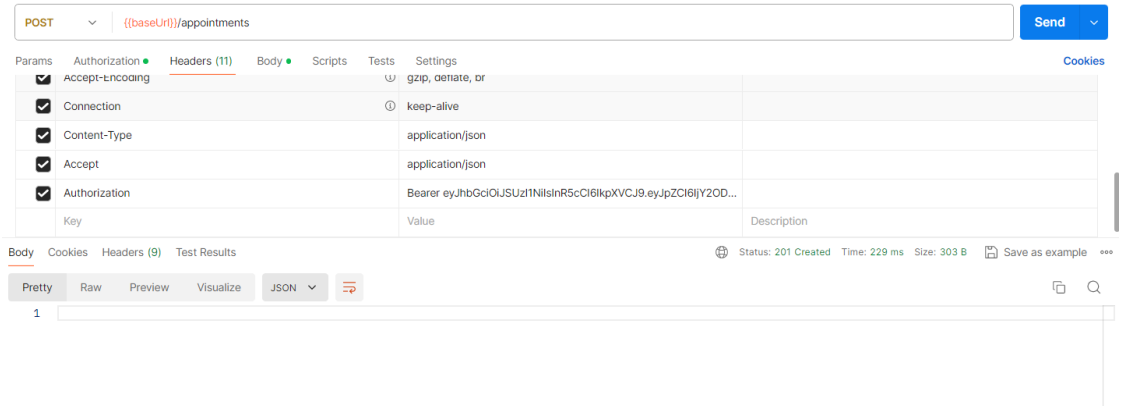


Figure 5-7. Cita reservada correctamente

Recuperamos las citas del usuario que ha reservado la cita y observamos que la cita se ha creado correctamente en el bloque de tiempo que se ha creado por el especialista

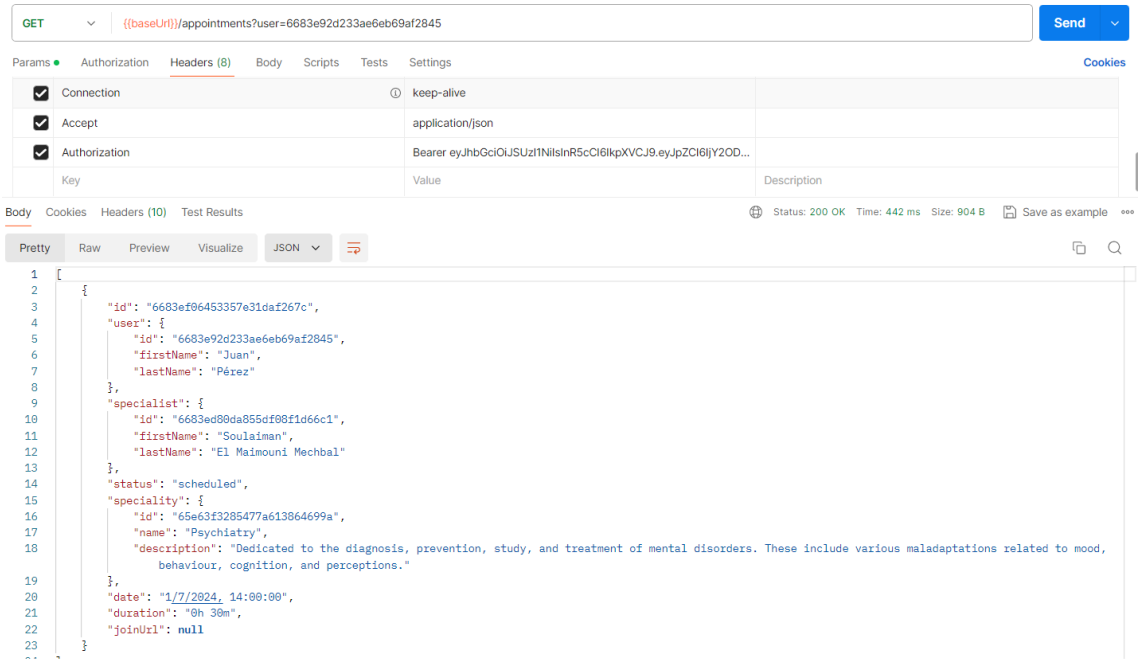


Figure 5-8. Citas reservadas por parte del usuario

5.8 Conclusión

El microservicio de gestión de citas está diseñado para ser seguro, escalable y fácil de mantener. Utiliza tecnologías modernas y sigue buenas prácticas de desarrollo para asegurar la gestión eficiente y segura de las citas entre usuarios y especialistas. Con la integración de este microservicio en el sistema de teleconsulta, se facilita una administración integral de las citas y los horarios de los especialistas, contribuyendo a la eficiencia y efectividad del sistema en su conjunto.

6 VISIÓN GENERAL DE WEBRTC

6.1 Introducción a WebRTC

El contenido de esta sección está basado en el libro "WebRTC for the Curious" [22], escrito por los propios implementadores de WebRTC. Este libro proporciona una visión teórica detallada de WebRTC y sus tecnologías subyacentes. Esta introducción servirá como base teórica para entender la implementación de WebRTC en el proyecto SEHHA, que se describirá en las secciones siguientes.

Web Real-Time Communication (WebRTC) es una tecnología avanzada que permite la comunicación en tiempo real entre navegadores y aplicaciones móviles sin la necesidad de instalar plugins o software adicional. Desarrollado y estandarizado por el World Wide Web Consortium (W3C) y el Internet Engineering Task Force (IETF), WebRTC permite la transmisión de audio, vídeo y datos a través de la web, lo que facilita la creación de aplicaciones como videoconferencias, transmisión en vivo, juegos en línea y más.

6.2 APIs de WebRTC

Las APIs de WebRTC proporcionan las herramientas necesarias para construir aplicaciones de comunicación en tiempo real. Estas APIs son:

- MediaStream API: Permite el acceso a las entradas multimedia del dispositivo del usuario, como cámaras y micrófonos. Esta API es fundamental para capturar y transmitir audio y vídeo en tiempo real.
- RTCPeerConnection: Es la API central que permite establecer una conexión de red entre dos pares. Maneja la configuración de la conexión, el intercambio de flujos multimedia y la gestión de la conectividad de red.
- RTCDataChannel: Permite la transmisión de datos arbitrarios entre pares, proporcionando una forma eficiente de enviar mensajes, archivos y otros datos en tiempo real.

6.3 Protocolos de WebRTC

WebRTC se basa en varios protocolos y tecnologías clave para su funcionamiento:

- STUN (Session Traversal Utilities for NAT) y TURN (Traversal Using Relays around NAT): Estos protocolos ayudan a los clientes de WebRTC a superar las barreras de NAT y firewall, permitiendo la conectividad directa en la mayoría de los casos.
- ICE (Interactive Connectivity Establishment): Proporciona un marco para descubrir la mejor ruta para la transmisión de medios a través de servidores STUN y TURN.
- DTLS (Datagram Transport Layer Security) y SRTP (Secure Real-time Transport Protocol): DTLS se utiliza para cifrar los datos y SRTP para proteger los flujos de medios, garantizando que todas las comunicaciones sean seguras y privadas.
- SCTP (Stream Control Transmission Protocol): Se utiliza para la transmisión de datos a través de los canales de datos de WebRTC, proporcionando características como la entrega ordenada y no ordenada de mensajes y la retransmisión parcial en caso de pérdida de paquetes.

6.4 Fases de la Comunicación en WebRTC

El proceso de comunicación en WebRTC se puede dividir en cuatro fases principales:

- Signalling: Esta fase se refiere al intercambio inicial de información necesario para establecer una conexión WebRTC. Involucra el intercambio de mensajes entre los pares para negociar capacidades, como códecs de audio y vídeo, y direcciones de red. Aunque WebRTC no define un protocolo específico de señalización, se pueden usar varios métodos como WebSockets, HTTP, y otros.
- Connecting: Una vez que se ha realizado el intercambio de señalización, los pares inician el proceso de establecimiento de la conexión. Aquí, los candidatos ICE se intercambian con el fin de descubrir las mejores rutas para la comunicación directa entre los pares. Esto puede incluir el uso de servidores STUN y TURN para atravesar NATs y firewalls.
- Securing: La seguridad es una parte integral de WebRTC. Durante esta fase, se establece una conexión segura utilizando DTLS para cifrar todos los datos que se intercambian entre los pares. Esto asegura que la comunicación es privada y no puede ser interceptada por terceros.
- Communicating: Una vez establecida y asegurada la conexión, los pares pueden comenzar a comunicarse. Esta fase incluye la transmisión de audio, vídeo y datos. Los flujos multimedia se gestionan utilizando RTP/RTCP, mientras que los datos se transmiten a través de SCTP sobre DTLS.

6.5 Topologías de Red en WebRTC

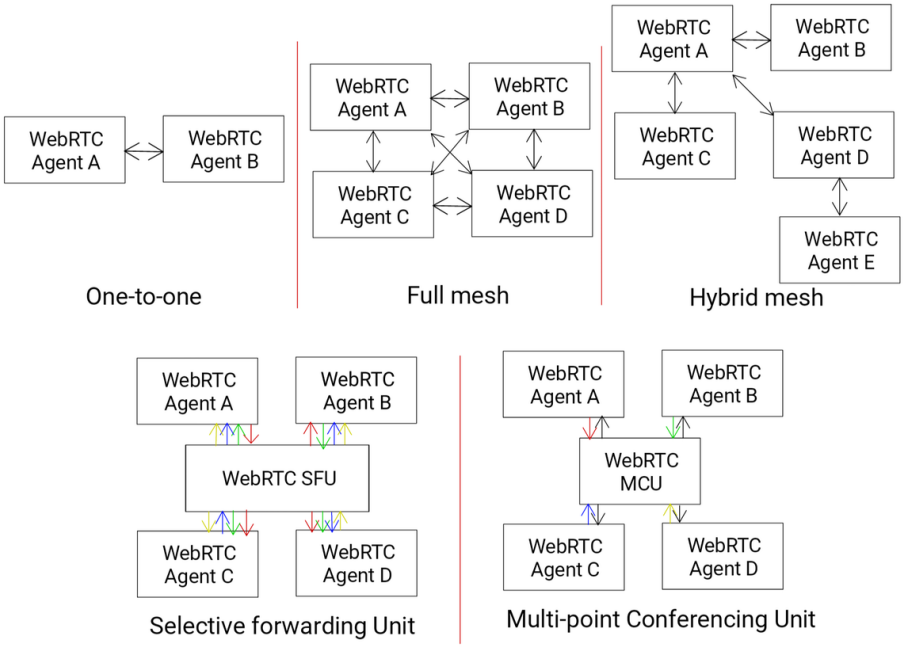


Figure 6-1. Topologías de red en WebRTC

WebRTC permite varias topologías de red para diferentes casos de uso:

- Uno a Uno (One-to-One): Es la conexión más básica donde dos pares se comunican directamente. Es ideal para videollamadas simples y transferencias de archivos entre dos usuarios.
- Malla Completa (Full Mesh): En esta topología, cada participante establece una conexión directa con todos los demás participantes, lo que es adecuado para pequeñas conferencias, pero no escala bien debido al aumento exponencial de conexiones y uso de ancho de banda.
- Malla Híbrida (Hybrid Mesh): A diferencia de la malla completa, las conexiones no se establecen entre todos los usuarios. Los usuarios pueden retransmitir los datos entre los usuarios, de esta manera, el agente que emite los datos no usa tanto ancho de banda como en el caso de la malla completa. En este caso la topología es parecida a la de un árbol.
- Unidad de Reenvío Selectiva (Selective Forwarding Unit): Una SFU recibe flujos de medios de todos los participantes y reenvía selectivamente estos flujos a los demás participantes. Esto reduce la carga de

ancho de banda en los clientes y es adecuado para conferencias más grandes.

- Unidad de Conferencia Multipunto (Multi-point Conferencing Unit): Una MCU recibe los flujos de medios de todos los participantes, los mezcla en un solo flujo y lo envía de vuelta a cada participante, simplificando la recepción y el procesamiento en los clientes, aunque aumenta la carga de procesamiento en el servidor.

6.6 Control de Congestión

WebRTC utiliza varias implementaciones de control de congestión para optimizar la transmisión de medios en tiempo real:

- GCC (Google Congestion Control): Ajusta dinámicamente el ancho de banda de la transmisión basándose en las condiciones de la red, utilizando algoritmos que miden el retardo de la red y las pérdidas de paquetes.
- NADA (Network-Assisted Dynamic Adaptation): Otra alternativa a GCC que también busca optimizar la transmisión de medios en tiempo real mediante el ajuste del ancho de banda en función de la retroalimentación de la red.
- SCReAM (Self-Clocked Rate Adaptation for Multimedia): Algoritmo de control de congestión que ajusta la tasa de transmisión basándose en las condiciones de la red y el rendimiento de los medios.

6.7 Casos de Uso de WebRTC

- Videoconferencias: WebRTC facilita la creación de aplicaciones de videoconferencia directamente en el navegador, sin necesidad de descargar software adicional.
- Transmisión en Vivo: Permite la transmisión en vivo de eventos, clases y otras actividades, con baja latencia y alta calidad.
- Teleoperación: Utilizado para controlar remotamente dispositivos y máquinas, como drones y robots, proporcionando una transmisión de vídeo en tiempo real y canales de datos para comandos.
- Juegos Multijugador: Proporciona una plataforma para la comunicación de baja latencia en juegos multijugador en tiempo real.
- Intercambio de Archivos: Permite la transferencia directa de archivos entre pares sin necesidad de servidores intermedios.

6.8 Seguridad en WebRTC

WebRTC incluye varias características de seguridad integradas, como el cifrado obligatorio de todos los flujos de medios y datos utilizando DTLS y SRTP. Además, WebRTC requiere el uso de HTTPS para todas las aplicaciones web que accedan a dispositivos multimedia o establezcan conexiones de red, garantizando la privacidad y seguridad de las comunicaciones.

6.9 Conclusión

WebRTC es una tecnología robusta y flexible que permite la comunicación en tiempo real en la web. Con su conjunto de APIs y su capacidad para manejar audio, vídeo y datos, WebRTC abre un amplio abanico de posibilidades para el desarrollo de aplicaciones interactivas y colaborativas, proporcionando una base sólida para la próxima generación de comunicaciones web.

7 MICROSERVICIO DE SEÑALIZACIÓN DE VÍDEO

7.1 Introducción al Microservicio

7.1.1 Descripción General

El microservicio de señalización de vídeo es un componente esencial dentro del sistema de teleconsulta SEHHA. Su principal función es gestionar la señalización entre los pares WebRTC que participan en una cita, permitiendo la comunicación en tiempo real entre usuarios y especialistas. Además, se integra con el microservicio de gestión de citas mediante RabbitMQ para recibir información de las citas y gestionar los participantes en las videollamadas.

Las funcionalidades principales incluyen:

- Gestión de Señalización: Facilita el intercambio de mensajes de señalización WebRTC entre pares.
- Integración con el Microservicio de Citas: Recibe información de citas a través de RabbitMQ para gestionar los participantes en las videollamadas.

7.2 Arquitectura del Microservicio

7.2.1 Diagrama de Arquitectura

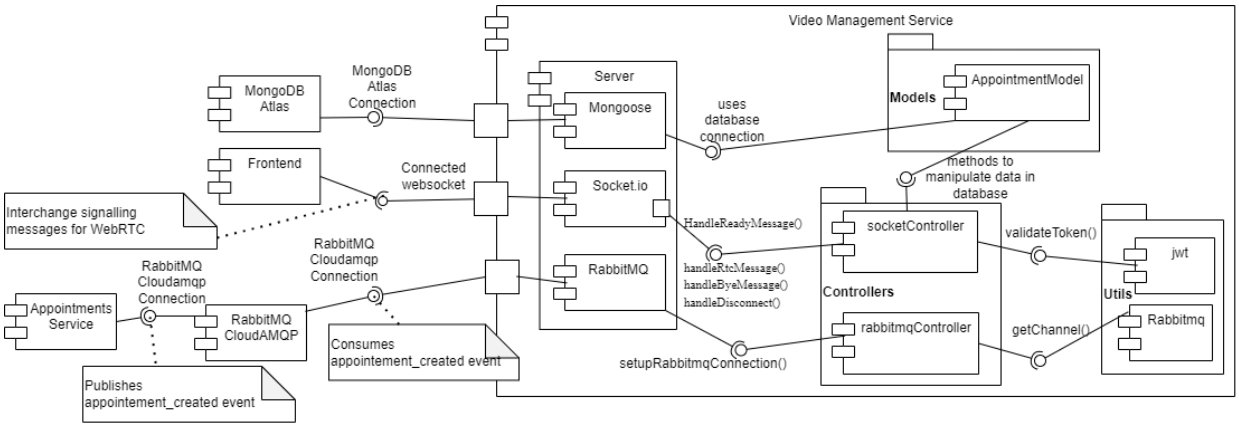


Figura 7-1. Arquitectura del microservicio de señalización de vídeo

El diagrama de componentes ilustra la arquitectura del microservicio de señalización de vídeo, destacando los principales componentes y sus interacciones. El Server es el núcleo del microservicio, compuesto por tres subcomponentes principales: Mongoose para la conexión y gestión de la base de datos MongoDB Atlas; Express

Server que expone las rutas necesarias para las operaciones de señalización; y RabbitMQ que maneja la mensajería asíncrona para la comunicación entre microservicios.

Los Controladores gestionan las operaciones de señalización en tiempo real, utilizando Socket.io para la transmisión de mensajes WebRTC. Los modelos de datos definidos en Mongoose (appointmentModel) representan las citas y eventos de señalización. Además, se utilizan utilidades específicas para la gestión de conexiones y mensajes mediante Socket.io y RabbitMQ.

7.3 Componentes Internos

7.3.1 Server

El componente del servidor es el núcleo del microservicio. Inicializa el servidor Express, que sirve la API utilizada para interactuar con el servicio y manejar la señalización WebRTC. Este servidor establece y gestiona las conexiones con dos elementos críticos: la base de datos MongoDB en MongoDB Atlas y el sistema de mensajería RabbitMQ.

- Express Server: Expone las rutas necesarias para las operaciones de señalización.
- Socket.io: Maneja la comunicación en tiempo real entre los clientes y el servidor.
- RabbitMQ: Facilita la comunicación entre microservicios mediante mensajería asíncrona.

7.3.2 Controladores

Los controladores son responsables de gestionar las operaciones de señalización en tiempo real, incluyendo la gestión de mensajes de WebRTC y la comunicación con otros microservicios a través de RabbitMQ.

- Controlador de Conexiones: Gestiona las conexiones de los clientes, asegurando que se autentiquen correctamente y se unan a la sala de videollamada correspondiente.
- Controlador de Mensajes RTC: Maneja los mensajes de oferta, respuesta y candidatos ICE necesarios para establecer y mantener la conexión WebRTC.
- Controlador de Desconexiones: Gestiona la desconexión de los clientes, asegurando que los estados de las citas se actualicen correctamente.
- Controlador de gestión de eventos de RabbitMQ: Maneja la recepción de mensajes de citas creadas por el servicio de gestión de citas a través de la cola de RabbitMQ

7.3.3 Modelos de Datos

Se utilizan modelos de datos definidos con Mongoose para representar y gestionar la información de las citas en MongoDB Atlas.

- Appointment Model: Representa las citas y almacena información sobre los participantes, el estado de la cita y los eventos de señalización.

El esquema del modelo de datos appointmentModel incluye los siguientes campos:

- user: Referencia al usuario que reservó la cita.
- specialist: Referencia al especialista con quien se reservó la cita.
- appointmentId: Identificador único de la cita.
- status: Estado de la cita (pendiente, activa, completada, cancelada).
- userConnected: Booleano que indica si el usuario está conectado.
- specialistConnected: Booleano que indica si el especialista está conectado.

- userSocketId: Identificador del socket del usuario.
- specialistSocketId: Identificador del socket del especialista.
- events: Array de eventos relacionados con la cita.
- createdAt: Fecha de creación de la cita.
- updatedAt: Fecha de última actualización de la cita.

7.3.4 Utilidades

Las utilidades incluyen funciones para gestionar la conexión con RabbitMQ y la publicación de eventos, así como la verificación de tokens JWT para la autenticación de los usuarios.

- Gestión de Conexiones: Funciones para conectar y desconectar clientes de las salas de videollamadas.
- Publicación de Eventos: Funciones para publicar y consumir eventos en RabbitMQ.
- Verificación de Tokens: Funciones para verificar la autenticidad de los tokens JWT.

7.4 Implementación

7.4.1 Lenguajes y Tecnologías Utilizadas

El microservicio de señalización de video está desarrollado en Node.js, utilizando Express.js como framework principal para construir el servidor. Se utiliza Socket.io para la gestión de conexiones en tiempo real y Mongoose para la interacción con MongoDB Atlas. RabbitMQ se utiliza para la mensajería asíncrona entre microservicios.

7.4.2 Flujos de Datos y Comunicación

7.4.2.1 Flujos de Datos

El flujo de datos principal en este microservicio se centra en la gestión de las sesiones de videollamada. Los datos de las citas se sincronizan a través de RabbitMQ y se almacenan en MongoDB. Los mensajes de señalización se transmiten en tiempo real entre los clientes a través de Socket.io.

7.4.2.2 Comunicación entre Microservicios

El microservicio de señalización se comunica con el microservicio de gestión de citas utilizando RabbitMQ para recibir eventos de creación de citas. Se consume el evento “appointment_created”, lo que permite sincronizar las citas y gestionar los participantes en las videollamadas.

7.5 Mensajería y Secuencia de Eventos

7.5.1 Mensajería de WebSocket

La comunicación entre los clientes y el servidor de señalización se realiza mediante WebSocket, gestionado por Socket.io. Los tipos de mensajes manejados incluyen:

- Ready: Indica que un cliente está listo para establecer la comunicación y unirse a la sala de videollamada.
- Offer/Answer: Mensajes de oferta y respuesta de WebRTC para establecer la conexión WebRTC. Estos mensajes contienen las ofertas/respuestas SDP (Session Description Protocol), que describe las capacidades de los peers (clientes). Un cliente (normalmente el que inicia la llamada) envía un mensaje offer al servidor. El servidor reenvía este mensaje al otro cliente en la sala. El otro cliente responde con

un mensaje answer. El servidor reenvía este mensaje al cliente que envió el offer.

- Candidate: Los mensajes de candidatos ICE (Interactive Connectivity Establishment) se utilizan para intercambiar información sobre las posibles rutas de red para establecer la conexión P2P (peer-to-peer).
- Bye: Indica que un cliente ha dejado la sala de videollamada.

7.5.2 Secuencia de Eventos

La siguiente secuencia de eventos describe el flujo de mensajes entre los clientes y el servidor de señalización:

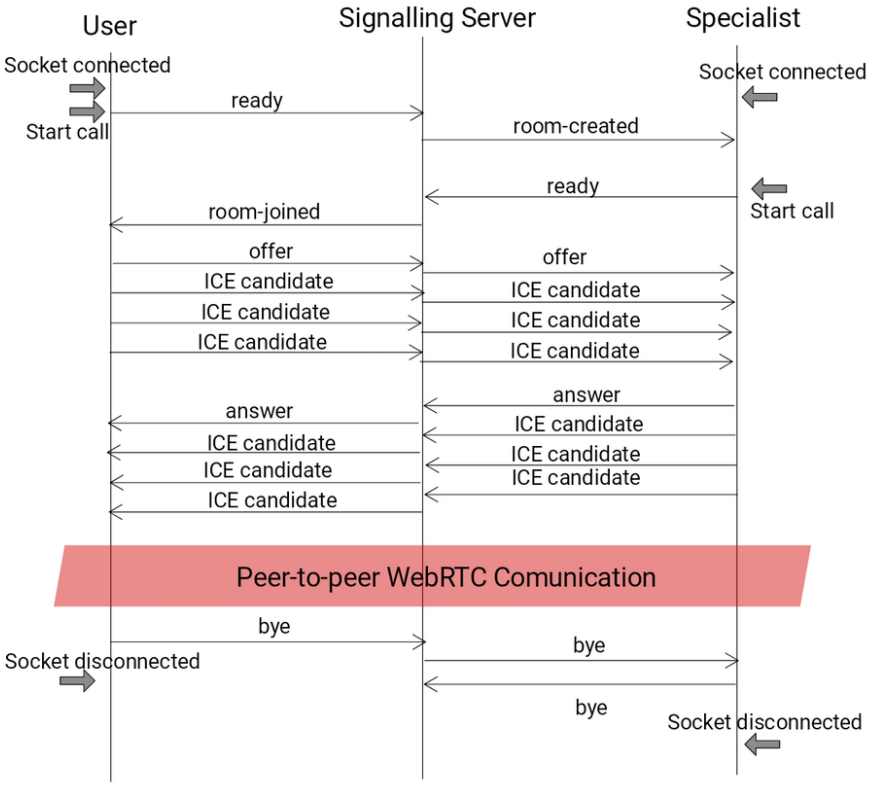


Figura 7-2. Diagrama de Secuencia de Mensajes de Señalización WebRTC

El primer paso en el proceso de señalización WebRTC es la conexión inicial de los clientes (usuario y especialista) al servidor de señalización. Esta conexión se establece utilizando WebSockets, gestionados por Socket.io.

Una vez establecida la conexión, el cliente (usuario o especialista) envía un mensaje ready al servidor de señalización para indicar que está listo para unirse a una videollamada. Suponemos que el usuario inicia la llamada:

- El usuario envía un mensaje ready al servidor de señalización con su token JWT y el identificador de la sala.
- El servidor verifica la autenticidad del token JWT y comprueba si el usuario tiene los permisos necesarios para unirse a la sala especificada.
- Si la verificación es exitosa y la sala aún no ha sido creada, el servidor crea la sala y envía un mensaje room-created de vuelta al usuario.
- Si la sala ya ha sido creada, el servidor simplemente añade al usuario a la sala existente y envía un mensaje room-joined.

Un tiempo después el especialista se une también a la llamada:

- El especialista también envía un mensaje ready al servidor con su token JWT y el identificador de la sala.
- El servidor sigue el mismo proceso de verificación que para el usuario.
- Si la sala ya existe, el servidor añade al especialista a la sala y envía un mensaje room-joined.

Después de que ambos clientes (usuario y especialista) se hayan unido a la sala, se lleva a cabo el intercambio de mensajes SDP (Session Description Protocol) para describir las capacidades multimedia de cada cliente. El usuario envía una oferta:

- El usuario inicia el proceso de establecimiento de la conexión enviando un mensaje offer al servidor de señalización.
- Este mensaje contiene una descripción SDP que incluye información sobre los codecs y formatos de medios que el usuario soporta.
- El servidor reenvía este mensaje offer al especialista.

Especialista responde con respuesta a la oferta enviada por el usuario:

- Al recibir el offer, el especialista responde con un mensaje answer.
- Este mensaje también contiene una descripción SDP con las capacidades multimedia del especialista.
- El servidor reenvía el answer al usuario.

Para establecer la conectividad de red, ambos clientes intercambian información sobre posibles rutas de red mediante mensajes de candidatos ICE (Interactive Connectivity Establishment). Tanto el usuario como el especialista recopilan sus candidatos ICE y los envían al servidor de señalización, estos candidatos ICE contienen información sobre las posibles rutas de red que pueden usarse para la conexión directa entre los clientes. El servidor de señalización reenvía estos mensajes de candidatos ICE entre los clientes, facilitando así el establecimiento de la conexión P2P.

Cuando un cliente decide terminar la videollamada, se envía un mensaje bye al servidor de señalización. Suponiendo que el usuario cuelga la llamada:

- El usuario envía un mensaje bye al servidor, indicando que desea finalizar la llamada.
- El servidor reenvía este mensaje al especialista.
- El usuario se desconecta del socket.

La llamada del especialista a su vez se cuelga automáticamente:

- El especialista también puede enviar un mensaje bye para finalizar la llamada.
- El servidor reenvía este mensaje al usuario (si aún está conectado).
- El especialista se desconecta del socket.

7.6 Seguridad

7.6.1 Mecanismos de Autenticación y Autorización

Se utiliza JWT para autenticar a los usuarios y especialistas. Tras la autenticación exitosa en el microservicio de gestión de usuarios, se genera un token JWT que se envía al cliente. Este token se incluye en cada uno de los mensajes de señalización y se valida en el microservicio de señalización utilizando la clave pública. Esto asegura que solo los usuarios autorizados puedan interactuar con el sistema.

7.6.2 Protección de Datos

Toda la comunicación entre los clientes y el servidor se realiza a través de HTTPS, utilizando certificados SSL/TLS generados por Amazon Certificate Manager. Esto garantiza que la información sensible se transmita de manera segura. El dominio utilizado para acceder al microservicio es video.sehha-telemedicine.com, y todas las peticiones se gestionan de manera segura a través de este dominio.

7.7 Despliegue y Escalabilidad

7.7.1 Despliegue en EKS

El microservicio de señalización de vídeo se despliega en contenedores Docker y se orquesta mediante Kubernetes en un clúster de Amazon EKS (Elastic Kubernetes Service). Kubernetes permite la escalabilidad automática del microservicio basándose en la carga de trabajo, asegurando que el sistema pueda responder eficientemente a aumentos en la demanda.

7.7.2 Balanceo de Carga y Alta Disponibilidad

Kubernetes se encarga de reiniciar automáticamente los contenedores que fallan, asegurando la disponibilidad continua del servicio. Para exponer el servicio al exterior, se utiliza un balanceador de carga de AWS, configurado desde Kubernetes, que distribuye el tráfico entrante de manera uniforme y asegura que las solicitudes lleguen a los pods disponibles.

7.8 Pruebas

En esta sección se detallará el proceso de prueba del flujo de mensajes en WebRTC y la verificación de la correcta actualización de los estados en la base de datos. Las pruebas asegurarán que los mensajes se intercambian correctamente entre el usuario, el servidor de señalización y el especialista, y que los estados en la base de datos reflejan con precisión el estado actual de la videollamada. El objetivo de las pruebas es verificar que los mensajes `ready`, `offer`, `answer`, `ICE candidate` y `bye` se envían y reciben correctamente entre los clientes y el servidor de señalización y asegurar que los estados de conexión de los usuarios en la base de datos se actualizan correctamente en cada etapa del flujo de mensajes.

7.8.1 Conexión de los Clientes

Se inicia el servidor de señalización, el usuario se conecta al servidor de señalización y luego el especialista se conecta también, ambos a través de sus respectivos sockets:

```
Listening on port 5000
RabbitMQ consumer setup complete
MongoDB connected
Client connected: N0tsHRclF64JRrnnAAAC
Client connected: NAQ1rvI-pzgtjvFUAAAF
```

Figura 7-3. Inicio del servidor y de las conexiones a sockets

7.8.2 Mensajes “ready” y Creación/Unión de Sala

El usuario envía un mensaje de tipo “ready” al servidor de señalización, cuando lo recibe el servidor de señalización actualiza la base de datos con la información del socket del usuario conectado así como el estado de conexión que establece el usuario como conectado poniendo el valor “userConnected” a “true” para dicha cita tras validar el token y verificar que el usuario pertenece a la cita a la que se está conectando:

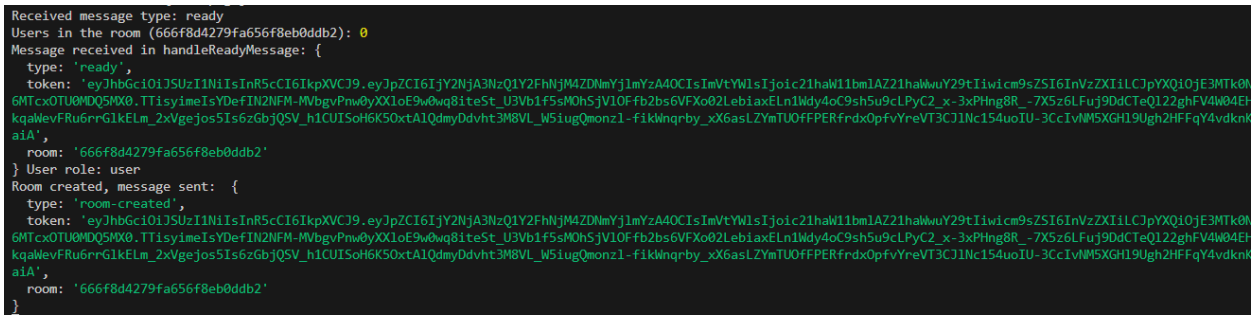


Figura 7-4. Logs del inicio de la llamada

Figura 7-5. Estado de la base de datos tras el inicio de la llamada por parte del usuario

A continuación, y de igual manera, inicia la llamada el especialista enviando un mesaje de tipo “ready”. Se valida el token del especialista y si es válido el servidor se emite un mensaje del tipo “room-joined” al existir ya un usuario conectado. En la base de datos el estado se actualiza el documento indicando que el especialista también se ha unido a la sesión:

Figura 7-6. Estado de la base de datos tras el inicio de la llamada por parte del especialista

7.8.3 Intercambio de Mensajes `offer` y los candidatos del usuario

El usuario envía un mensaje `offer` seguido de varios candidatos ICE al servidor de señalización. El servidor reenvía estos mensajes al especialista.

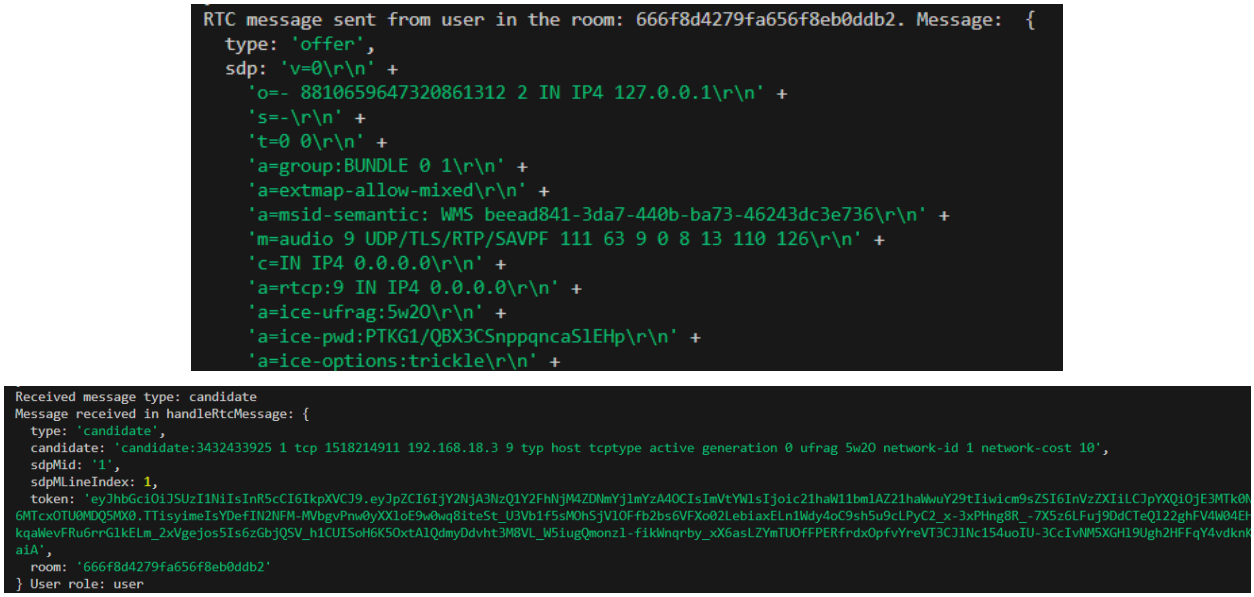


Figura 7-7. Logs del envío del mensaje de oferta y los ICE candidates por parte del usuario

7.8.4 Intercambio de Mensajes `answer` y los candidatos del especialista

El especialista al recibir el mensaje de oferta, envía un mensaje `answer` seguido de varios candidatos ICE al servidor de señalización. El servidor reenvía estos mensajes al usuario

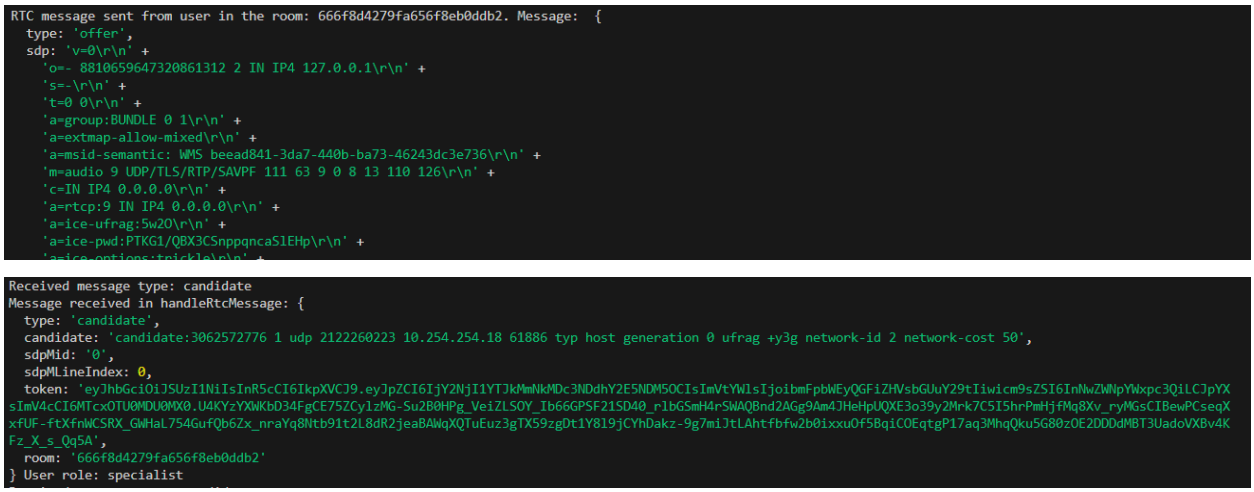


Figura 7-8. Logs del envío del mensaje de respuesta y los ICE candidates por parte del especialista

7.8.5 Finalización de la Llamada con Mensaje `bye`

El usuario al colgar la llamada envía un mensaje `bye` al servidor de señalización y se desconecta. El especialista también envía un mensaje `bye` al servidor de señalización y se desconecta. Cuando el servidor de señalización recibe un mensaje de tipo bye, lo reenvía.

Figura 7-9. Logs del envío del intercambio de mensajes de tipo “bye” para cerrar la sesión

En la base de datos se actualiza la información indicando que los participantes ya están fuera de la sesión

Figura 7-10. Estado de la base de datos tras el cierre de sesión

7.9 Conclusión

El microservicio de señalización de vídeo está diseñado para ser seguro, escalable y fácil de mantener. Utiliza tecnologías modernas y sigue buenas prácticas de desarrollo para asegurar la gestión eficiente y segura de las videollamadas entre usuarios y especialistas. Con la integración de este microservicio en el sistema de teleconsulta, se facilita una comunicación en tiempo real efectiva y confiable, mejorando la experiencia de teleconsulta para todos los participantes.

8 FRONTEND DE SEHHA Y SU INTERACCIÓN CON LOS MICROSERVICIOS

8.1 Introducción al Frontend

8.1.1 Descripción General

El frontend de SEHHA está desarrollado utilizando React y Vite, junto con React Bootstrap como framework de estilo. Este frontend es esencial para proporcionar una interfaz de usuario intuitiva y responsiva tanto para los usuarios (pacientes) como para los especialistas. La aplicación gestiona diferentes estados y contextos, asegurando una experiencia de usuario fluida y eficiente.

Las funcionalidades principales incluyen:

- Registro e inicio de sesión para usuarios y especialistas.
- Gestión de citas para usuarios.
- Gestión de calendarios para especialistas.
- Conexión a videollamadas para teleconsultas.

8.2 Arquitectura del Frontend

8.2.1 Estructura del Proyecto

El frontend de SEHHA está estructurado siguiendo las convenciones estándar de una aplicación creada con Vite. Esta estructura permite una clara organización del código, facilitando el mantenimiento y la escalabilidad. A continuación se describen las principales carpetas y su contenido:

- src/components: Contiene los componentes reutilizables de la interfaz de usuario, como botones, formularios y otros elementos UI comunes. Estos componentes son modulares y se pueden integrar fácilmente en diferentes partes de la aplicación.
- src/contexts: Define los contextos necesarios para la gestión del estado global de la aplicación. Los contextos permiten compartir información relevante, como la autenticación del usuario, sin necesidad de pasar props manualmente a través de la jerarquía de componentes.
- src/hooks: Contiene hooks personalizados que encapsulan la lógica específica de la aplicación, permitiendo reutilizar esta lógica en varios componentes. Ejemplos incluyen hooks para manejar la autenticación, la gestión de citas y las videollamadas.
- src/pages: Incluye las páginas principales de la aplicación, tales como la página de inicio, login, signup, gestión de citas y videollamadas. Cada página es un componente React independiente que puede tener su propia lógica y estado local.
- src/utils: Contiene funciones utilitarias que son usadas en diversas partes de la aplicación para tareas comunes como formateo de fechas, validación de formularios, y manejo de errores.

8.2.2 Componentes Principales

La aplicación frontend de SEHHA está compuesta por varios componentes principales que facilitan la interacción del usuario con las distintas funcionalidades del sistema:

- Homepage: Esta es la página de entrada de la aplicación, donde se muestra un mensaje de bienvenida y se proporcionan opciones para login y signup. Es la primera interacción del usuario con la plataforma.
- Login y Signup: Estas páginas permiten a los usuarios y especialistas registrarse y autenticarse en la plataforma. Utilizan formularios para la entrada de datos del usuario y gestionan la lógica de autenticación.
- Gestión de Citas: Dirigida a usuarios autenticados, esta página permite reservar, visualizar y cancelar citas médicas. Incluye un calendario interactivo y una lista de citas programadas.
- Gestión de Calendarios: Esta funcionalidad está destinada a los especialistas, permitiéndoles gestionar sus horarios disponibles. Los especialistas pueden definir, modificar y eliminar bloques de tiempo en los que están disponibles para consultas.
- Videollamadas: En esta página, los usuarios y especialistas pueden unirse a videollamadas para llevar a cabo consultas médicas virtuales. La página maneja la conexión y comunicación en tiempo real usando WebRTC.

8.2.3 Gestión de Estado

La gestión del estado en la aplicación frontend de SEHHA se maneja mediante useContext y useReducer de React, que permiten una gestión eficiente y estructurada del estado tanto a nivel global como local. A continuación se describe cómo se implementa la gestión del estado en la aplicación:

8.3 Implementación

8.3.1 Lenguajes y Tecnologías Utilizadas

El frontend de SEHHA está desarrollado utilizando JavaScript junto con el popular framework de desarrollo React. Para optimizar el proceso de construcción y ofrecer un entorno de desarrollo ágil, se emplea Vite, una herramienta moderna de build que mejora significativamente la velocidad de arranque y la experiencia del desarrollador.

8.3.2 Estilos y Responsividad

La gestión de estilos en el frontend de SEHHA se realiza utilizando una combinación de React Bootstrap y estilos CSS personalizados. Cada componente y página tiene sus propios estilos definidos en archivos .css, lo que asegura una separación clara entre la lógica y la presentación.

React Bootstrap proporciona componentes estilizados y responsivos que se adaptan automáticamente a diferentes tamaños de pantalla. Utilizar React Bootstrap facilita la implementación de una interfaz de usuario coherente y atractiva.

Aunque React Bootstrap cubre la mayoría de las necesidades estilísticas, también se utilizan estilos CSS personalizados para ajustar y personalizar la apariencia de componentes específicos. Estos estilos adicionales se importan directamente en los componentes relevantes.

La combinación de estas tecnologías asegura que la aplicación sea completamente responsiva, adaptándose a una amplia gama de dispositivos, desde teléfonos móviles hasta pantallas de escritorio, mejorando así la experiencia del usuario.

8.3.3 Rutas y Navegación

La navegación dentro de la aplicación SEHHA se gestiona mediante React Router, una biblioteca estándar para

el manejo de rutas en aplicaciones React. Esta herramienta permite definir rutas de manera declarativa y facilita la navegación entre diferentes vistas de la aplicación.

El siguiente fragmento de código del componente principal “App.jsx” ilustra cómo se gestionan las rutas y la autenticación en SEHHA:

```jsx
import React, { useEffect, useState } from 'react';
import { BrowserRouter as Router, Route, Routes, Navigate } from 'react-router-dom';
import HomePage from './pages/HomePage/HomePage';
import AppointmentBookingPage from './pages/UserPages/AppointmentBookingPage/AppointmentBookingPage';
import AppointmentsPage from './pages/UserPages/AppointmentsPage/AppointmentsPage';
import ConsultationPage from './pages/UserPages/ConsultationPage/ConsultationPage';
import SpecialistAppointmentsPage from './pages/SpecialistPages/SpecialistAppointmentsPage/SpecialistAppointmentsPage';
import SpecialistConsultationPage from './pages/SpecialistPages/SpecialistConsultationPage/SpecialistConsultationPage';
import Scheduler from './pages/SpecialistPages/Scheduler/Scheduler';
import LoginSelectionPage from './pages/Login/LoginSelectionPage';
import LoginPage from './pages/Login/LoginPage/LoginPage';
import SignUpPage from './pages/Login/SignUpPage/SignUpPage';
import 'bootstrap/dist/css/bootstrap.min.css';
import { useAuthContext } from './hooks/useAuthContext';
import validateToken from './utils/validateToken';
import { useLogout } from './hooks/useLogout';
const App = () => {
  const { user: authenticatedUser } = useAuthContext();
  const { logout } = useLogout();
  const token = authenticatedUser?.token;
  const [isTokenValid, setIsTokenValid] = useState(false);
  const [userRole, setUserRole] = useState(null);
  useEffect(() => {
    if (token) {
      const decodedToken = validateToken(token);
      console.log("decodedToken", decodedToken);
      const id = authenticatedUser.specialist ? authenticatedUser.specialist.id : authenticatedUser.user ? authenticatedUser.user.id : null;
      if (authenticatedUser && id !== decodedToken?.id) {
        console.log("id !== decodedToken?.id", id, decodedToken?.id);
        logout();
      } else {
        setUserRole(decodedToken?.role);
        setIsTokenValid(true);
      }
    }
  }, [token, authenticatedUser, logout]);

  if (!isTokenValid && authenticatedUser) {
    return <div>Loading...</div>;
  }
  return (
    <Router>
      <Routes>
        <Route
          path={import.meta.env.VITE_REACT_APP_HOMEPAGE}
          element={!authenticatedUser ? <HomePage /> : (userRole === 'user' ? <Navigate to={import.meta.env.VITE_REACT_APP_USER_APPOINTMENTS_URL} /> : userRole === 'specialist' ?
          <Navigate to={import.meta.env.VITE_REACT_APP_SPECIALIST_APPOINTMENTS_URL} /> : logout())}
        />
        <Route path={import.meta.env.VITE_REACT_APP_LOGIN_SIGNUP_SELECTION} element={!authenticatedUser ? <LoginSelectionPage /> : <Navigate to={import.meta.env.VITE_REACT_APP_HOMEPAGE} />} />
        <Route path={import.meta.env.VITE_REACT_APP_LOGIN} element={!authenticatedUser ? <LoginPage /> : <Navigate to={import.meta.env.VITE_REACT_APP_HOMEPAGE} />} />
        <Route path={import.meta.env.VITE_REACT_APP_SIGNUP} element={!authenticatedUser ? <SignUpPage /> : <Navigate to={import.meta.env.VITE_REACT_APP_HOMEPAGE} />} />

        {/* User Routes */}
        {authenticatedUser && userRole === 'user' && <Route path={import.meta.env.VITE_REACT_APP_USER_APPOINTMENTS_URL} element={<AppointmentsPage />} />}
        {authenticatedUser && userRole === 'user' && <Route path={import.meta.env.VITE_REACT_APP_USER_RESERVE_URL} element={<AppointmentBookingPage />} />}
        {authenticatedUser && userRole === 'user' && <Route path={import.meta.env.VITE_REACT_APP_USER_VIDEO_URL} element={<ConsultationPage />} />}

        {/* Specialist Routes */}
        {authenticatedUser && userRole === 'specialist' && <Route path={import.meta.env.VITE_REACT_APP_SPECIALIST_APPOINTMENTS_URL} element={<SpecialistAppointmentsPage />} />}
        {authenticatedUser && userRole === 'specialist' && <Route path={import.meta.env.VITE_REACT_APP_SPECIALIST_VIDEO_URL} element={<SpecialistConsultationPage />} />}
        {authenticatedUser && userRole === 'specialist' && <Route path={import.meta.env.VITE_REACT_APP_SPECIALIST_SCHEDULE_APPOINTMENT_URL} element={<Scheduler />} />}

        {/* Catch-all Route */}
        {/* <Route path="*" element={<Navigate to={import.meta.env.VITE_REACT_APP_HOMEPAGE} />} /> */}
      </Routes>
    </Router>
  );
};

export default App;
```

Figura 8-1. App.jsx, componente principal del frontend de SEHHA

El código verifica si el usuario está autenticado y valida el token utilizando la función validateToken. Si el token no es válido, se llama a la función logout para cerrar la sesión del usuario. Se definen rutas principales para diferentes componentes de la aplicación, como HomePage, LoginSelectionPage, LoginPage, y SignUpPage. Dependiendo del estado de autenticación y el rol del usuario, se redirige a las páginas correspondientes.

Las rutas de los usuarios incluyen rutas para la gestión de citas, reserva de citas y videollamadas, accesibles solo para usuarios autenticados con el rol de usuario. Y las rutas para los especialistas incluyen rutas para la gestión de citas de especialistas, videollamadas y programación de horarios, accesibles solo para usuarios autenticados con el rol de especialista. Una ruta de captura general que redirige a la página principal si la ruta especificada no coincide con ninguna de las rutas definidas.

Esta configuración de rutas asegura que solo los usuarios autenticados puedan acceder a las funciones específicas de la aplicación.

8.4 Integración con el Backend

8.4.1 Comunicación con Microservicios

El frontend se comunica con los microservicios de backend de las siguientes maneras:

- APIs RESTful: Para realizar peticiones HTTP a los microservicios de gestión de usuarios y citas, se utiliza Axios, una biblioteca de JavaScript popular para hacer solicitudes HTTP.
- WebSockets: Para la señalización de videollamadas, el frontend se comunica con el microservicio de señalización de vídeo utilizando Socket.io.

8.4.2 Autenticación y Autorización

8.4.2.1 Autenticación

Los usuarios y especialistas se autentican mediante el servicio de gestión de usuarios, que devuelve un token JWT (JSON Web Token). Este token se almacena en el localStorage del navegador y se incluye en las cabeceras de las solicitudes HTTP y WebSockets para la autorización. Esto asegura que todas las solicitudes al backend sean realizadas por usuarios autenticados. El siguiente código muestra cómo se maneja la autenticación en el frontend:

```
import { createContext, useReducer, useEffect } from 'react';

export const AuthContext = createContext();

export const authReducer = (state, action) => {
  switch (action.type) {
    case 'LOGIN':
      return { user: action.payload };
    case 'LOGOUT':
      return { user: null };
    default:
      return state;
  }
};

export const AuthContextProvider = ({ children }) => {
  const [state, dispatch] = useReducer(authReducer, {
    user: null
  });

  useEffect(() => {
    const user = JSON.parse(localStorage.getItem('user'));

    if (user) {
      dispatch({ type: 'LOGIN', payload: user });
    }
  }, []);

  console.log('AuthContext state:', state);

  return (
    <AuthContext.Provider value={{ ...state, dispatch }}>
      { children }
    </AuthContext.Provider>
  );
};
```

Figure 8-2. Componente de gestión de contexto AuthContextProvider

Para hacer uso de este contexto se debe envolver el componente principal de la aplicación como se observa en la siguiente figura.

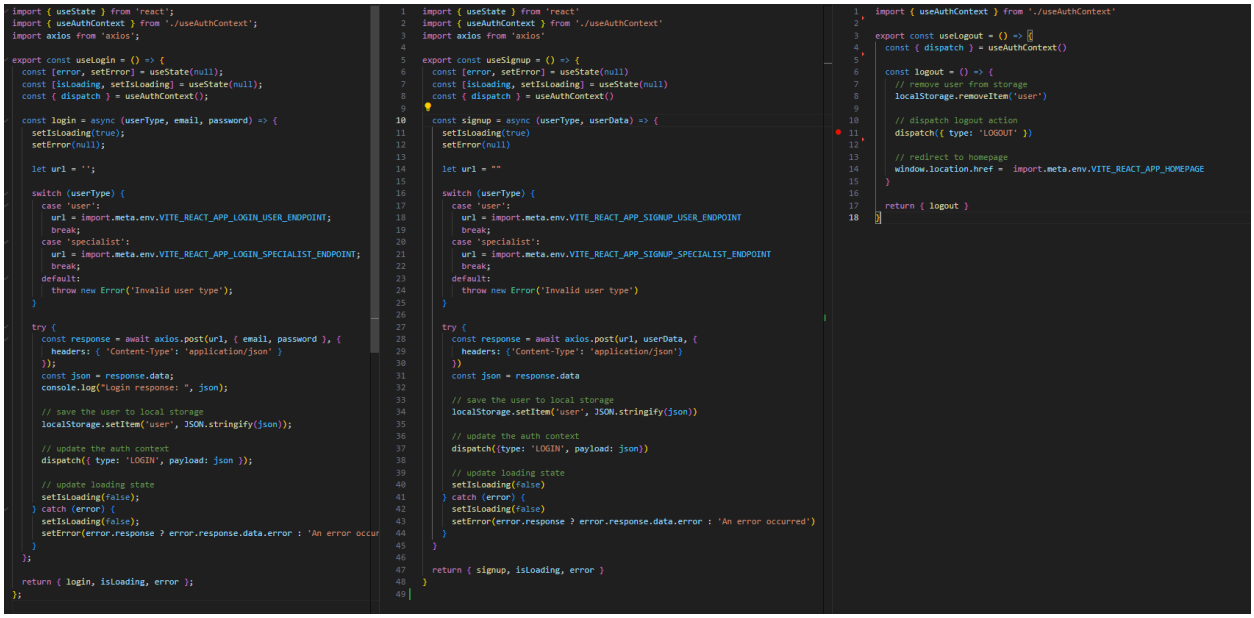
```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { AuthContextProvider } from './contexts/AuthContext';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <AuthContextProvider>
      <App />
    </AuthContextProvider>
  </React.StrictMode>
);
```

Figura 8-3. Index.jsx, el componente principal App dentro de AuthContextProvider

Este código asegura que el estado de autenticación se mantenga incluso después de que el usuario recargue la página o cierre y vuelva a abrir el navegador.

8.4.2.2 Hooks de Autenticación



```
import { useState } from 'react';
import { useAuthContext } from './useAuthContext';
import axios from 'axios';

export const useLogin = () => {
  const [error, setError] = useState(null);
  const [isLoading, setIsLoading] = useState(null);
  const { dispatch } = useAuthContext();

  const login = async (userType, email, password) => {
    setIsLoading(true);
    setError(null);

    let url = '';

    switch (userType) {
      case 'user':
        url = import.meta.env.VITE_REACT_APP_LOGIN_USER_ENDPOINT;
        break;
      case 'specialist':
        url = import.meta.env.VITE_REACT_APP_LOGIN_SPECIALIST_ENDPOINT;
        break;
      default:
        throw new Error('Invalid user type');
    }

    try {
      const response = await axios.post(url, { email, password }, {
        headers: { 'Content-Type': 'application/json' }
      });
      const json = response.data;
      console.log("Login response: ", json);

      // save the user to local storage
      localStorage.setItem('user', JSON.stringify(json));

      // update the auth context
      dispatch({ type: 'LOGIN', payload: json });

      // update loading state
      setIsLoading(false);
    } catch (error) {
      setIsLoading(false);
      setError(error.response ? error.response.data.error : 'An error occur
    }
  };

  return { login, isLoading, error };
};
```

```
import { useState } from 'react'
import { useAuthContext } from './useAuthContext'
import axios from 'axios'

export const useSignup = () => {
  const [error, setError] = useState(null)
  const [isLoading, setIsLoading] = useState(null)
  const { dispatch } = useAuthContext()

  const signup = async (userType, userData) => {
    setIsLoading(true)
    setError(null)

    let url = ""

    switch (userType) {
      case 'user':
        url = import.meta.env.VITE_REACT_APP_SIGNUP_USER_ENDPOINT
        break;
      case 'specialist':
        url = import.meta.env.VITE_REACT_APP_SIGNUP_SPECIALIST_ENDPOINT
        break;
      default:
        throw new Error('Invalid user type')
    }

    try {
      const response = await axios.post(url, userData, {
        headers: {'Content-Type': 'application/json'}
      })
      const json = response.data

      // save the user to local storage
      localStorage.setItem('user', JSON.stringify(json))

      // update the auth context
      dispatch({type: 'LOGIN', payload: json})

      // update loading state
      setIsLoading(false)
    } catch (error) {
      setIsLoading(false)
      setError(error.response ? error.response.data.error : 'An error occurred')
    }
  }

  return { signup, isLoading, error }
}
```

```
import { useAuthContext } from './useAuthContext'

export const useLogout = () => {
  const { dispatch } = useAuthContext()

  const logout = () => {
    // remove user from storage
    localStorage.removeItem('user')

    // dispatch logout action
    dispatch({ type: 'LOGOUT' })

    // redirect to homepage
    window.location.href =  import.meta.env.VITE_REACT_APP_HOMEPAGE
  }

  return { logout }
}
```

Figura 8-4. Hooks useLogin, useSignup y useLogout

Para manejar las operaciones de login, signup y logout, se han creado hooks personalizados que encapsulan la lógica de autenticación y actualizan el estado global de la aplicación:

- useLogin: Este hook define una función login que realiza una petición POST a la API de autenticación correspondiente (usuario o especialista) usando Axios. En función del tipo de usuario (user o specialist), se establece la URL de la API. Si la autenticación es exitosa, se guarda el token de usuario en localStorage y se actualiza el estado global mediante el dispatch de AuthContext.
- useSignUp: Similar a useLogin, este hook define una función signup que realiza una petición POST a la API de registro correspondiente. La URL se selecciona según el tipo de usuario. Tras un registro exitoso, se guarda el token en localStorage y se actualiza el estado global.
- useLogout: Este hook define una función logout que elimina el token de usuario de localStorage y actualiza el estado global para reflejar que no hay ningún usuario autenticado. También redirige al usuario a la página de inicio.

8.5 Páginas y Funcionalidades

8.5.1 Página de Inicio

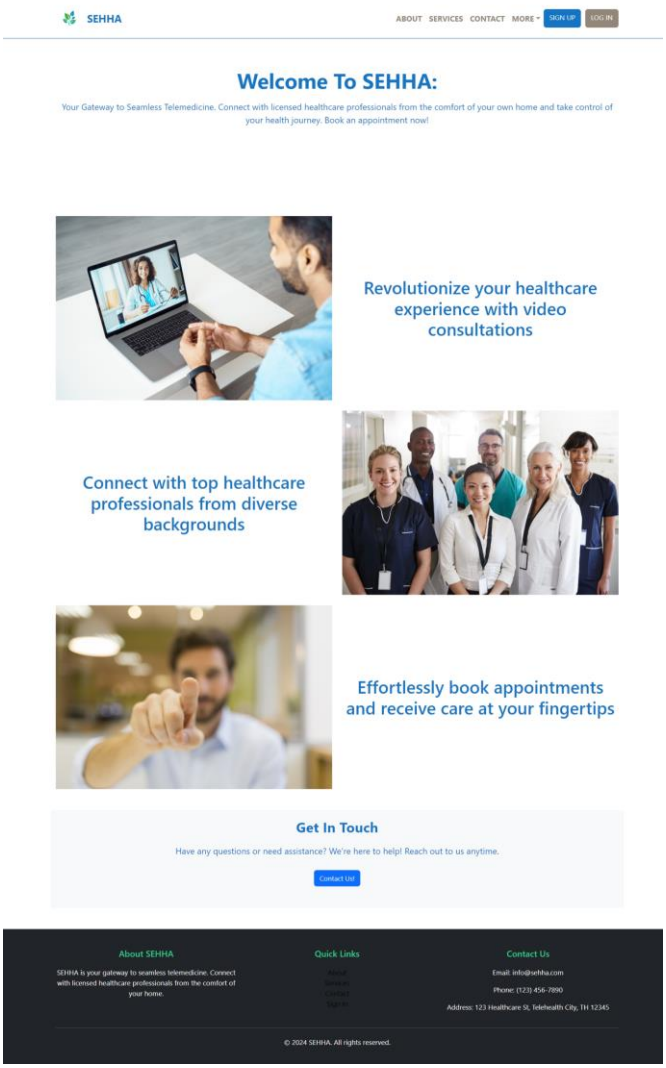


Figura 8-5. Página de inicio

La página de inicio (homepage) es la primera vista que los usuarios encuentran al acceder a la aplicación. Esta página muestra un mensaje de bienvenida, describe las principales funcionalidades de SEHHA y proporciona enlaces para que los usuarios y especialistas inicien sesión o se registren en el sistema.

8.5.2 Páginas de Login y Signup

Las páginas de login y signup permiten a los usuarios y especialistas registrarse e iniciar sesión en la aplicación SEHHA. A continuación, se detalla el proceso de navegación y las rutas asociadas. En la página principal, los usuarios son recibidos con opciones para login y signup, al seleccionar una opción, el usuario elige registrarse o iniciar sesión como usuario o especialista. Las rutas que se usan son las siguientes:

- Para la autenticaicón: https://sehha-telemedicine.com/login-signup-selection?action=signup
- Para la registración: https://sehha-telemedicine.com/login-signup-selection?action=login

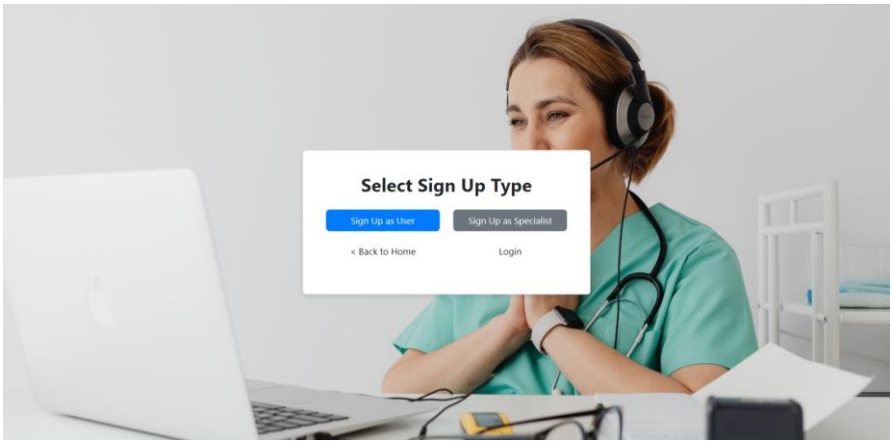


Figura 8-6. Selección del tipo de usuario para el registro

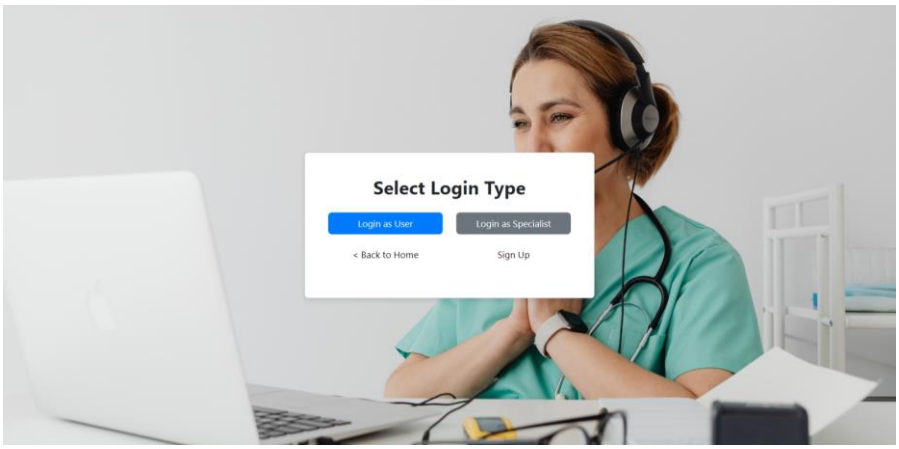


Figura 8-7. Selección del tipo de usuario para la autenticación

Los usuarios son redirigidos a los formularios específicos según su rol:

- Login Usuario: https://sehha-telemedicine.com/login?userType=user
- Login Especialista: https://sehha-telemedicine.com/login?userType=specialist
- Signup Usuario: https://sehha-telemedicine.com/signup?userType=user
- Signup Especialista: https://sehha-telemedicine.com/signup?userType=specialist

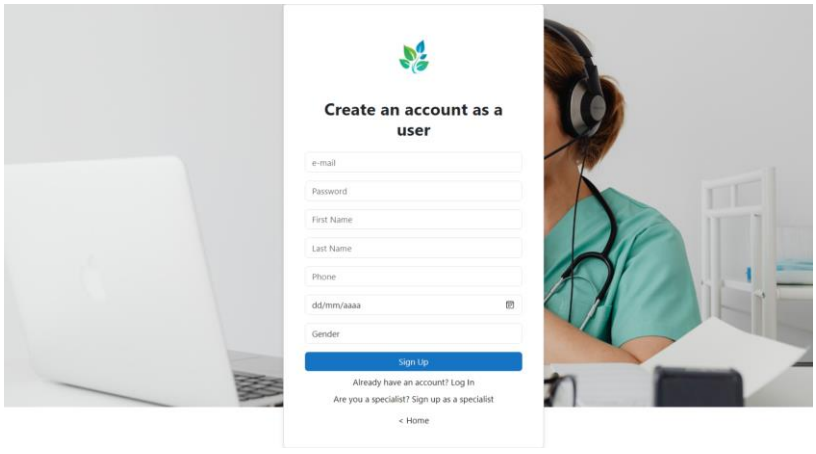


Figura 8-8. Signup de usuarios

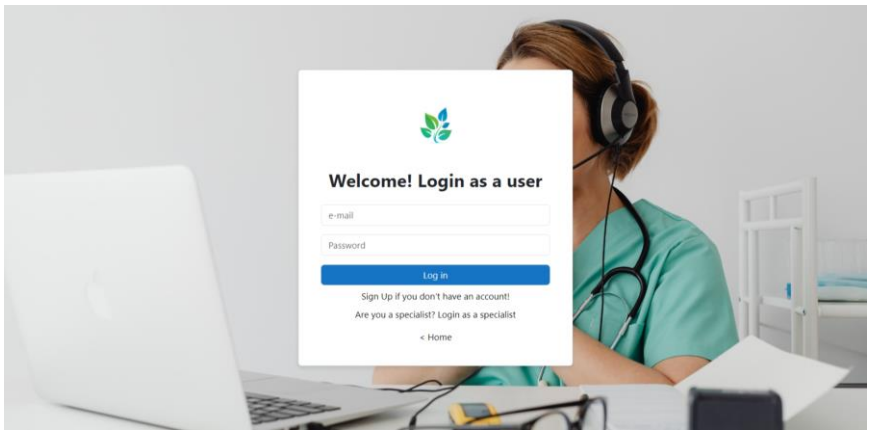


Figura 8-9. Login de usuarios

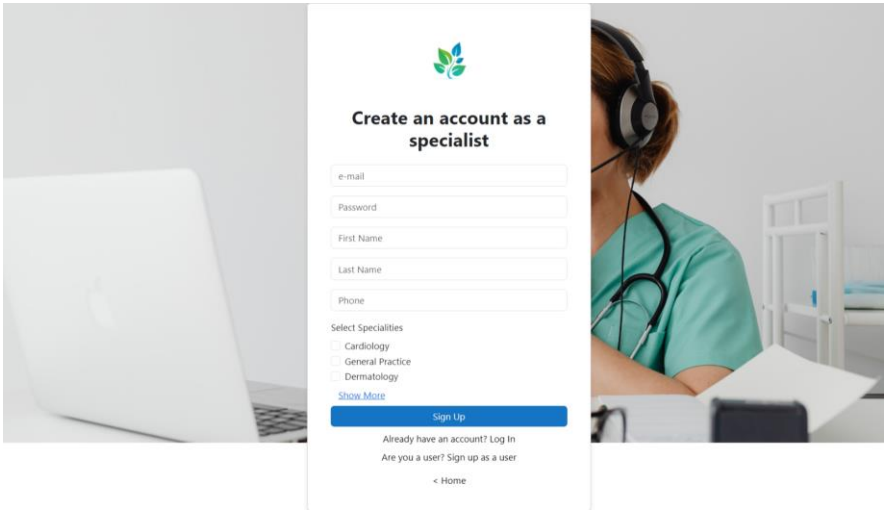


Figura 8-10. Signup de especialistas

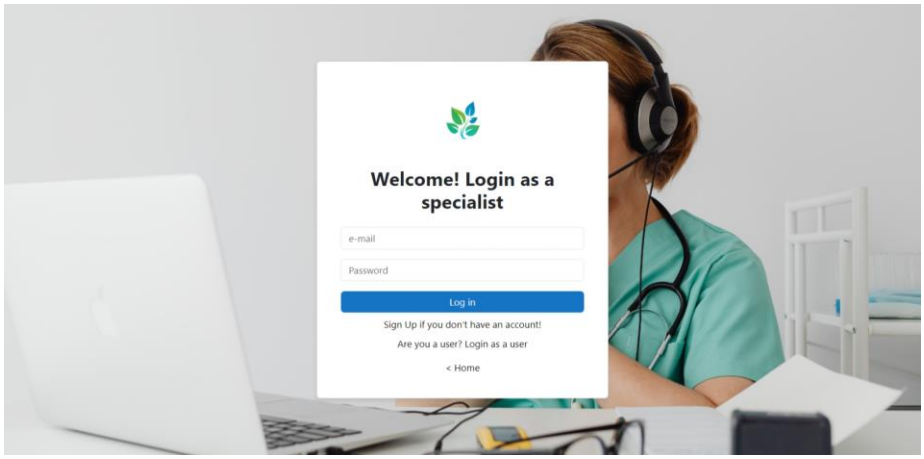


Figura 8-11. Login de especialistas

Al hacer clic en las opciones de login o signup, los usuarios son redirigidos a las rutas específicas según su elección. Los formularios de login y signup se validan al enviar las solicitudes al backend. La autenticación se maneja a través de los hooks `useLogin` y `useSignup`

8.5.3 Gestión de Citas para Usuarios

Una vez autenticados, los usuarios pueden acceder a una página donde pueden ver los calendarios de los especialistas disponibles y reservar citas. Esta página proporciona una interfaz intuitiva para seleccionar fechas y horas disponibles y confirmar las citas.

8.5.3.1 Pantalla de Selección de Citas

Los usuarios pueden seleccionar un especialista y ver sus horarios disponibles en un calendario. Esto se logra a través de una interfaz de usuario amigable que permite a los usuarios filtrar por especialidad y fechas. La pantalla también incluye un botón para reservar una cita una vez que se ha seleccionado una franja horaria disponible.

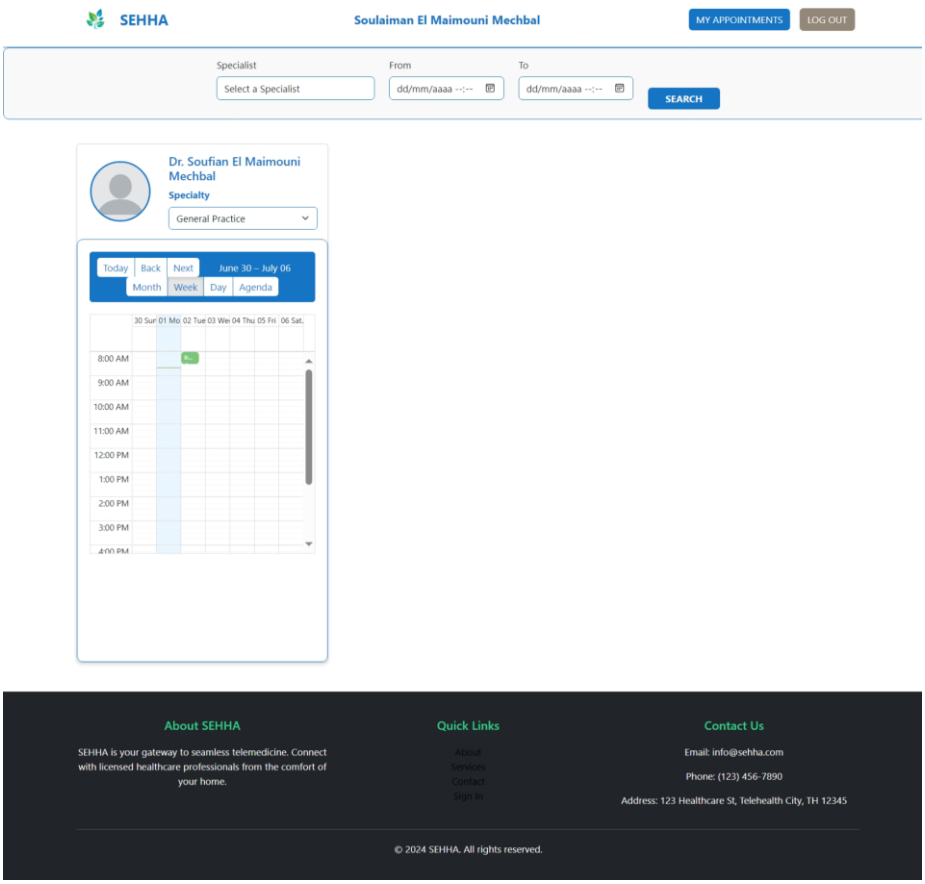


Figura 8-12. Pantalla de selección de citas para usuarios

8.5.3.2 Confirmación de la Cita

Después de seleccionar una franja horaria, se presenta una ventana emergente para confirmar la cita. Esta ventana muestra los detalles de la cita, incluyendo la fecha, hora y el especialista seleccionado. El usuario puede confirmar o cancelar la reserva desde esta ventana.

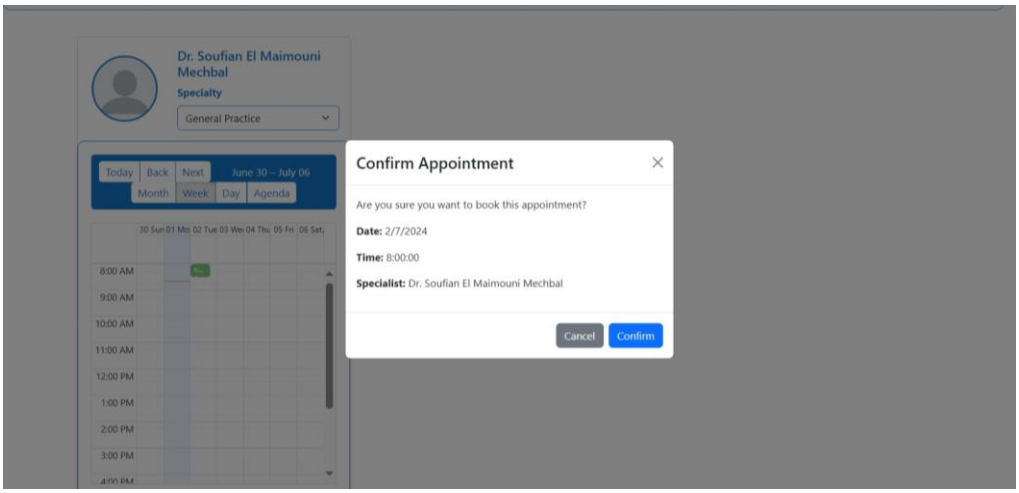


Figura 8-13. Ventana de confirmación de la cita

8.5.3.3 Cita Reservada

Una vez confirmada la cita, el usuario recibe una notificación indicando que la cita ha sido reservada con éxito.

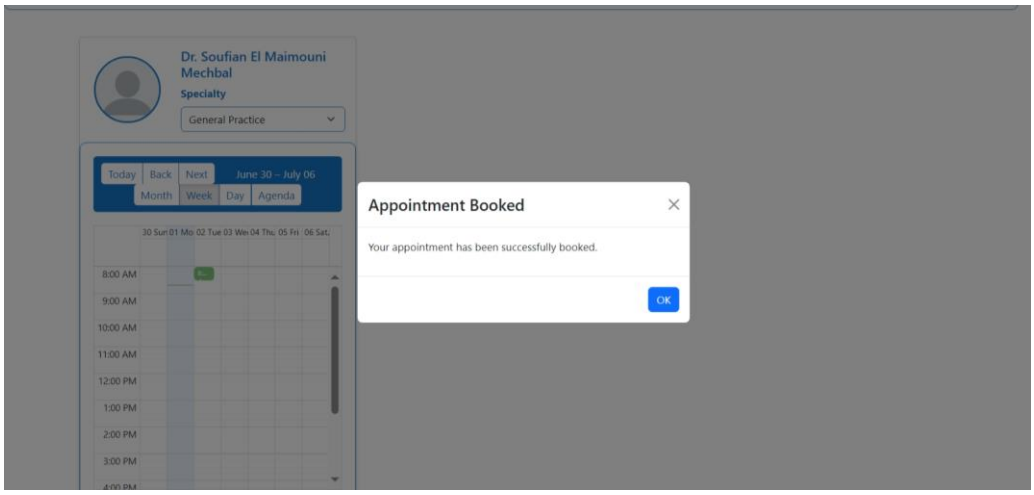


Figura 8-14. Notificación de cita reservada

8.5.3.4 Visualización de citas

Además, los usuarios pueden visualizar sus citas y cancelarlas si les es oportuno

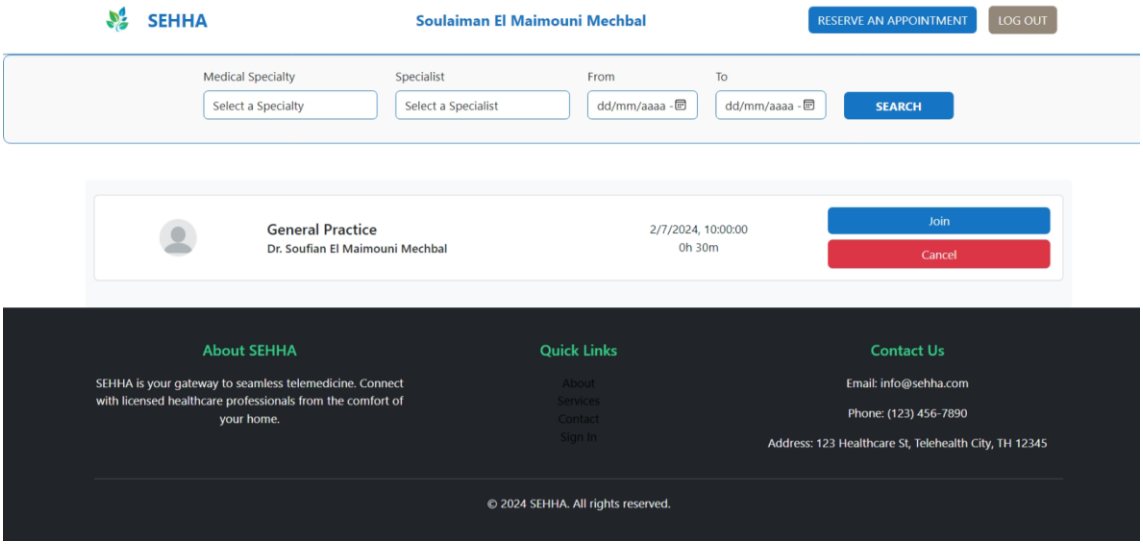


Figura 8-15. Visualización de citas reservadas para usuarios

8.5.4 Gestión de Calendarios para Especialistas

Los especialistas tienen acceso a una página donde pueden gestionar sus horarios, creando y modificando bloques de tiempo en los que están disponibles para citas. También pueden ver las citas reservadas por los usuarios.

8.5.4.1 Pantalla de Gestión de Horarios

La pantalla de gestión de horarios permite a los especialistas crear y gestionar sus bloques de tiempo disponibles. Los especialistas pueden especificar la fecha, hora y duración de las citas disponibles. También pueden configurar la recurrencia de las citas (por ejemplo, citas semanales).

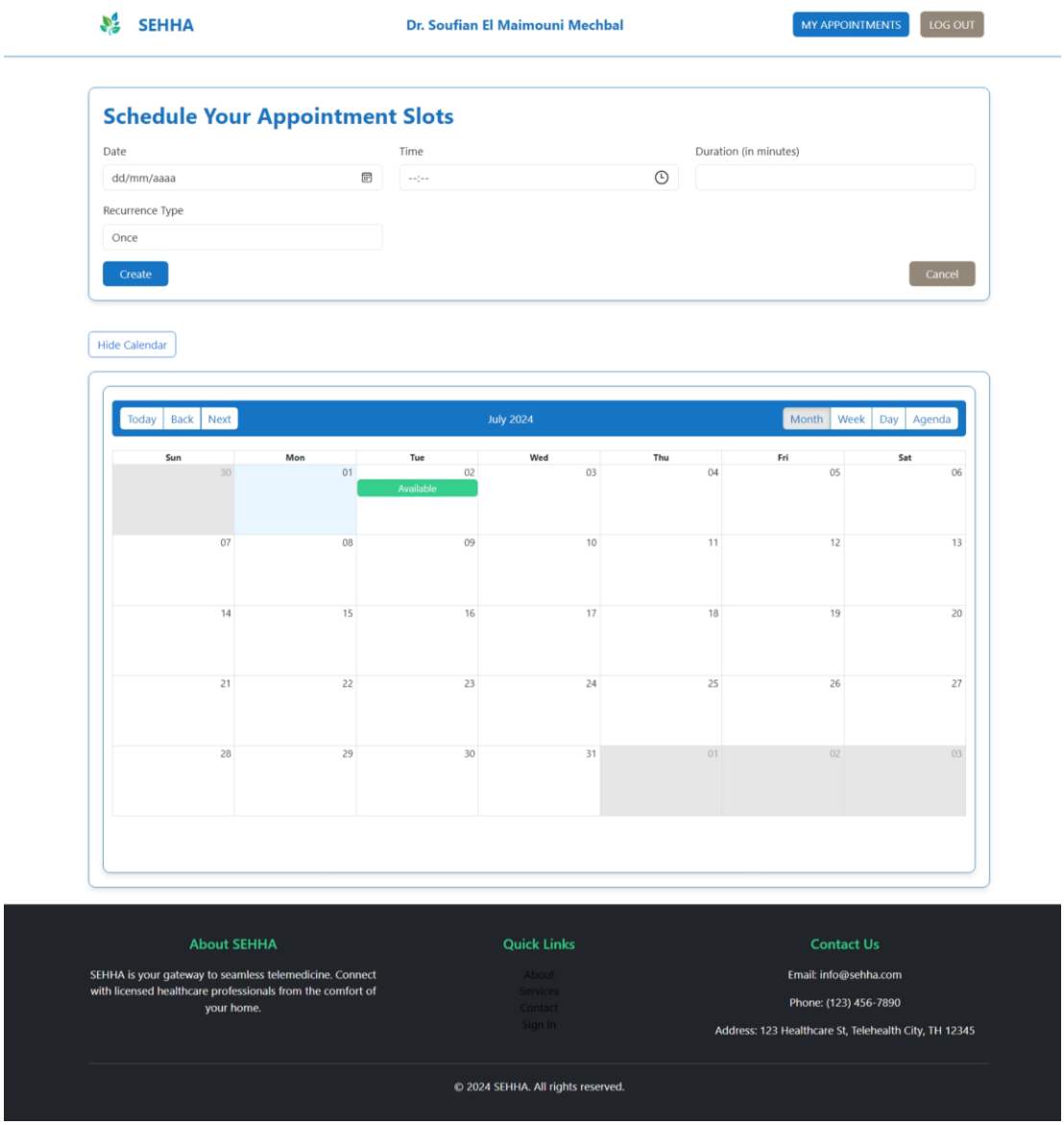


Figura 8-16. Pantalla de gestión de horarios para especialistas

8.5.4.2 Visualización de Citas

Además de gestionar sus horarios, los especialistas pueden ver las citas reservadas por los usuarios en un calendario. Esto les permite tener una visión general de su agenda y planificar sus actividades en consecuencia.

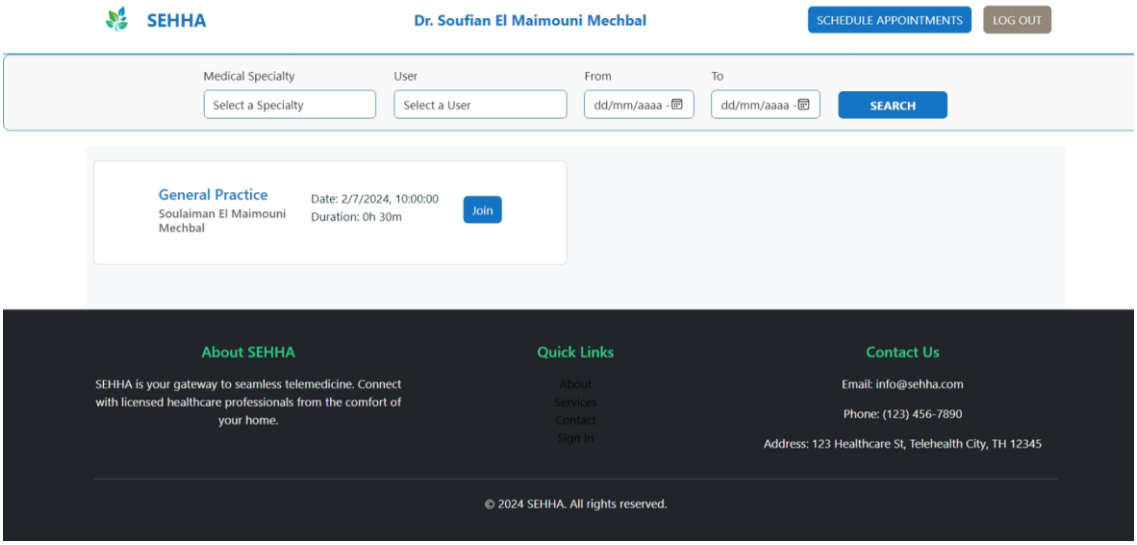


Figura 8-17. Visualización de citas reservadas para especialistas

8.5.5 Videollamadas

Tanto usuarios como especialistas pueden unirse a videollamadas desde sus respectivas páginas de citas pulsando el botón "Join". Esta funcionalidad se integra con el microservicio de señalización de vídeo para gestionar la señalización WebRTC necesaria para establecer las conexiones de videollamada uno a uno entre el usuario y el especialista.

8.5.5.1 Página de Videollamadas para Usuarios

La página de videollamadas permite a los usuarios unirse a una consulta de vídeo con el especialista. La interfaz incluye botones para iniciar la llamada, así como para activar o desactivar la cámara y el micrófono.

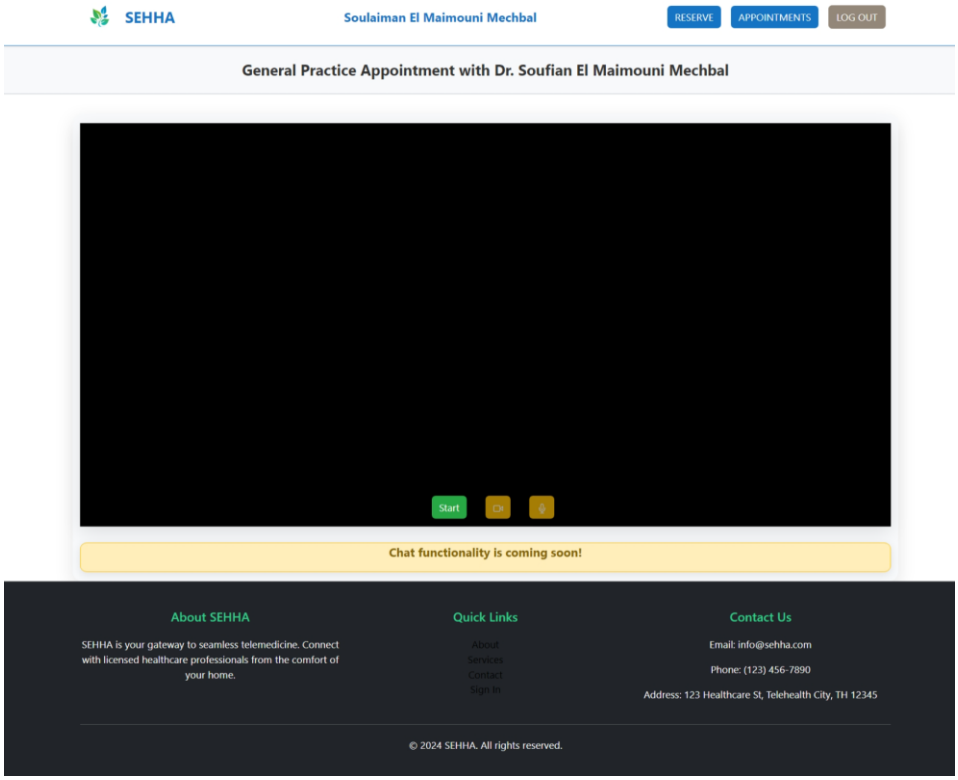


Figura 8-18. Página de videollamadas para usuarios

8.5.5.2 Página de Videollamadas para Especialistas

De manera similar, los especialistas pueden unirse a las videollamadas desde su propia interfaz. Esta página es funcionalmente idéntica a la de los usuarios, permitiendo una comunicación efectiva y fluida durante las consultas médicas.

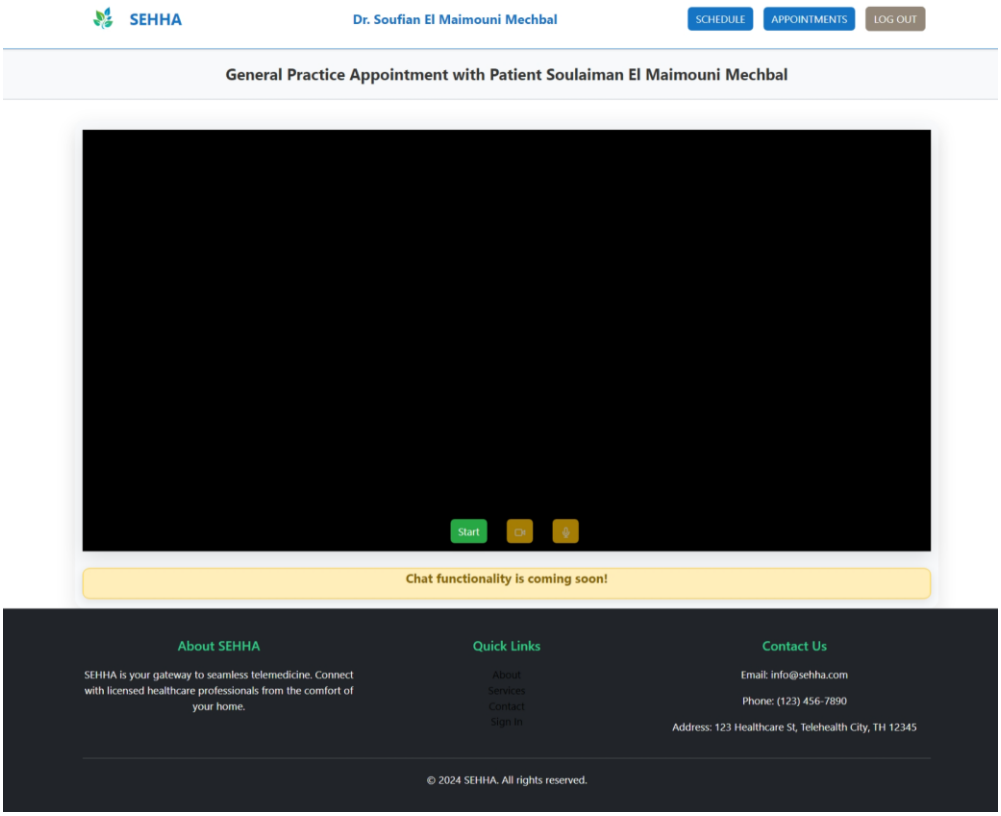


Figura 8-19. Página de videollamadas para especialistas

8.6 Implementación de WebRTC

El componente principal para la funcionalidad de videollamadas en el proyecto SEHHA se encuentra en el archivo `ConsultationOptions.jsx`. Este componente se encarga de gestionar las conexiones WebRTC entre los usuarios y los especialistas. A continuación, se explican los aspectos más importantes de esta implementación:

8.6.1 Configuración de ICE Servers

Para asegurar una conectividad robusta, especialmente en redes restringidas, se utilizan tanto servidores STUN como TURN. Los servidores STUN se encargan de descubrir la dirección pública y gestionar la conectividad NAT, mientras que los servidores TURN ayudan a enrutar el tráfico cuando la conectividad directa falla.

```
const servers = {
  iceServers: [
    {
      urls: [
        'stun:stun1.l.google.com:19302',
        'stun:stun2.l.google.com:19302',
      ],
    },
    {
      urls: 'turn:openrelay.metered.ca:80',
      username: 'openrelayproject',
      credential: 'openrelayproject'
    },
    {
      urls: 'turn:openrelay.metered.ca:443',
      username: 'openrelayproject',
      credential: 'openrelayproject'
    }
  ],
  iceCandidatePoolSize: 10,
};
```

Figura 8-20. Configuración de ICE Servers

8.6.2 Creación y Manejo de Conexiones Peer-to-Peer

El objeto `RTCPeerConnection` es el núcleo de cualquier aplicación WebRTC, permitiendo establecer, mantener y gestionar la conexión entre dos pares.

```
async function makeCall(appointmentId, socket, remoteVideo, token) {
  try {
    console.log('Making call with appointmentId:', appointmentId);
    pc = new RTCPeerConnection(servers);
    pc.onicecandidate = e => {
      console.log('ICE candidate event:', e);
      const message = {
        type: 'candidate',
        candidate: e.candidate ? e.candidate.candidate : null,
        sdpMid: e.candidate ? e.candidate.sdpMid : null,
        sdpMLineIndex: e.candidate ? e.candidate.sdpMLineIndex : null,
        token,
        room: appointmentId
      };
      console.log('Sending ICE candidate:', message);
      socket.emit('message', message);
    };
    pc.ontrack = e => {
      console.log('Track event:', e);
      remoteVideo.current.srcObject = e.streams[0];
    };
    localStream.getTracks().forEach(track => pc.addTrack(track, localStream));
    const offer = await pc.createOffer();
    console.log('Created offer:', offer);
    await pc.setLocalDescription(offer);
    const message = { type: 'offer', sdp: offer.sdp, token, room: appointmentId };
    console.log('Sending offer:', message);
    socket.emit('message', message);
  } catch (e) {
    console.error('Error in makeCall:', e);
  }
}
```

Figura 8-21. Función para iniciar una llamada

La función `makeCall` se ejecuta cuando un usuario o un especialista inicia una llamada.

- Se crea un nuevo objeto `RTCPeerConnection` utilizando la configuración de servidores ICE.
- Se define un manejador de eventos `onicecandidate` que se activa cuando el `RTCPeerConnection` genera nuevos candidatos ICE. Estos candidatos se envían al servidor de señalización junto con el token de autenticación y la identificación de la cita.
- El manejador de eventos `ontrack` actualiza la fuente del video remoto cuando se recibe una nueva

pista de media.

- Se capturan las pistas locales de video y audio y se añaden al `RTCPeerConnection`.
- Se crea una oferta utilizando `createOffer`, se establece como la descripción local y se envía al servidor de señalización.

8.6.3 Manejo de Mensajes de Señalización

WebRTC no define un protocolo específico para la señalización, por lo que se utilizan WebSockets para el intercambio de mensajes de señalización necesarios para establecer la conexión.

```
socket.on('message', e => {
  console.log('Received message:', e);
  if (!localStream) {
    console.log('Not ready yet');
    return;
  }
  console.log('Current appointmentId:', appointmentId);
  switch (e.type) {
    case 'offer':
      console.log('Handling offer');
      handleOffer(e, socket, remoteVideo, appointmentId, authenticatedUser.token);
      break;
    case 'answer':
      console.log('Handling answer');
      handleAnswer(e, appointmentId);
      break;
    case 'candidate':
      console.log('Handling candidate');
      handleCandidate(e, appointmentId);
      break;
    case 'room-created':
    case 'room-joined':
      console.log('Room created or joined');
      if (pc) {
        console.log('Already in call, ignoring');
        return;
      }
      console.log('Calling makeCall with appointmentId:', e.appointmentId);
      makeCall(appointmentId, socket, remoteVideo, authenticatedUser.token);  // Cor
      break;
    case 'bye':
      console.log('Handling bye');
      if (pc) {
        handleHangUpButton()
      }
      break;
    default:
      console.log('Unhandled message type:', e);
      break;
  }
```

Figura 8-22. Manejo de mensajes de señalización

Utilizando `useEffect`, se configura un listener para mensajes de señalización que llegan a través del socket. Dependiendo del tipo de mensaje recibido (`ready`, `room-created`, `room-joined`, `offer`, `answer`, `candidate`), se llaman a diferentes funciones para manejar cada caso. Estos mensajes incluyen las ofertas, respuestas y candidatos necesarios para establecer la conexión WebRTC.

```
async function handleOffer(offer, socket, remoteVideo, appointmentId, token) {
  console.log('Handling offer:', offer, 'for appointment', appointmentId);
  if (pc) {
    console.error('Existing peerconnection');
    return;
  }
  try {
    pc = new RTCPeerConnection(servers);
    pc.onicecandidate = e => {
      console.log('ICE candidate event:', e);
      const message = {
        type: 'candidate',
        candidate: e.candidate ? e.candidate.candidate : null,
        sdpMid: e.candidate ? e.candidate.sdpMid : null,
        sdpMLineIndex: e.candidate ? e.candidate.sdpMLineIndex : null,
        token,
        room: appointmentId
      };
      console.log('Sending ICE candidate:', message);
      socket.emit('message', message);
    };
    pc.ontrack = e => {
      console.log('Track event:', e);
      remoteVideo.current.srcObject = e.streams[0];
    };
    localStream.getTracks().forEach(track => pc.addTrack(track, localStream));
    await pc.setRemoteDescription(offer);
    const answer = await pc.createAnswer();
    console.log('Created answer:', answer);
    await pc.setLocalDescription(answer);
    const message = { type: 'answer', sdp: answer.sdp, token, room: appointmentId };
    console.log('Sending answer:', message);
    socket.emit('message', message);
  } catch (e) {
    console.error('Error in handleOffer:', e);
  }
}
```

Figura 8-23. Manejo de ofertas entrantes

La función `handleOffer` se ejecuta cuando se recibe una oferta de un par remoto.

- Se crea una nueva instancia de `RTCPeerConnection` si no existe una.
- Se configuran los manejadores de eventos `onicecandidate` y `ontrack` como se describió anteriormente.
- Se añaden las pistas locales a la conexión.
- Se establece la descripción remota utilizando `setRemoteDescription` con la oferta recibida.
- Se crea una respuesta utilizando `createAnswer`, se establece como la descripción local y se envía al servidor de señalización.

La función `handleAnswer` establece la descripción remota cuando se recibe una respuesta. La función `handleCandidate` añade los candidatos ICE recibidos a la conexión.

```
async function handleAnswer(answer, appointmentId) {
  console.log('Handling answer:', answer, 'for appointment', appointmentId);
  if (!pc) {
    console.error('No peerconnection');
    return;
  }
  try {
    await pc.setRemoteDescription(answer);
  } catch (e) {
    console.error('Error in handleAnswer:', e);
  }
}

async function handleCandidate(candidate, appointmentId) {
  console.log('Handling candidate:', candidate, 'for appointment', appointmentId);
  try {
    if (!pc) {
      console.error('No peerconnection');
      return;
    }
    await pc.addIceCandidate(candidate);
  } catch (e) {
    console.error('Error in handleCandidate:', e);
  }
}
```

Figura 8-24. Manejo de respuestas y candidatos ICE

La API `MediaStream` permite acceder a las cámaras y micrófonos del dispositivo del usuario. Con handleStartButton se accede a la cámara y le micrófono, y se envía un mensaje de tipo “ready” con el id de la cita y el token del usuario autenticado.

```
const handleStartButton = async () => {
  console.log('Starting call with appointmentId:', appointmentId);
  try {
    localStream = await navigator.mediaDevices.getUserMedia({ video: true, audio: { 'echoCancellation': true } });
    localVideo.current.srcObject = localStream;
  } catch (err) {
    console.error('Error while getting local media stream:', err);
  }
  const message = { type: 'ready', token: authenticatedUser.token, room: appointmentId };
  console.log('Sending ready:', message);
  socket.emit('message', message);
  muteAudButton.current.disabled = false;
  muteVidButton.current.disabled = false;
  setStarted(true)
};
```

Figura 8-25. Captura y transmisión de MediaStreams

La función `hangup` se encarga de finalizar la llamada, cerrando la conexión y liberando los recursos.

```
async function hangup() {
  console.log('Hanging up');
  if (pc) {
    pc.close();
    pc = null;
  }
  if (localStream) {
    localStream.getTracks().forEach(track => track.stop());
    localStream = null;
  }

  // Clear the video elements
  if (localVideo.current) {
    localVideo.current.srcObject = null;
  }
  if (remoteVideo.current) {
    remoteVideo.current.srcObject = null;
  }
}
```

Figura 8-26. Finalización de la llamada

8.6.4 Interacción del Usuario: Los botones de la interfaz

La interfaz de usuario para la gestión de videollamadas proporciona botones que permiten iniciar, finalizar, silenciar y alternar el vídeo durante la llamada. Estos botones interactúan con las funciones descritas anteriormente para controlar la llamada.

```
return (
  <Container className='consultation-options'>
    <Row className='video bg-main'>
      <div className="video-container">
        <video ref={remoteVideo} className='video-item big-video' autoPlay playsInline onClick={started ? toggleVideos : () => { console.log("not started") }}></video>
        <video ref={localVideo} className='video-item small-video' autoPlay playsInline muted onClick={started ? toggleVideos : () => { console.log("not started") }}></video>
        <div className="btns-container d-flex justify-content-center">

          {started ? <Button className='btn-end' ref={hangupButton} onClick={handleHangUpButton}>Hang</Button> :
          <Button className='btn-start' ref={startButton} onClick={handleStartButton}>Start</Button>}
          {videostate ?
            <Button className='btn-toggle' ref={muteVidButton} onClick={muteVideo}><FiVideo /></Button> :
            <Button className='btn-toggle' ref={muteVidButton} onClick={muteVideo}><FiVideoOff /></Button>}
          {audiostate ?
            <Button className='btn-toggle' ref={muteAudButton} onClick={muteAudio}><FiMic /></Button> :
            <Button className='btn-toggle' ref={muteAudButton} onClick={muteAudio}><FiMicOff /></Button>}
        </div>
      </div>
    </Row>
    <Row className="chat-coming-soon">
      <Col>
        <p>Chat functionality is coming soon!</p>
      </Col>
    </Row>
  </Container>
);
```

Figura 8-27. Interfaz de usuario para la gestión de videollamadas

- Botón de Iniciar Llamada: Llama a `handleStartButton`, que inicia la captura de los medios locales

y envía un mensaje "ready" al servidor de señalización.

- Botón de Finalizar Llamada: Llama a `handleHangUpButton`, que finaliza la llamada cerrando la conexión y liberando recursos.
- Botones de Silenciar Audio/Vídeo: Llaman a las funciones `muteAudio` y `muteVideo` para alternar el estado de las pistas de audio y vídeo.

Como prueba del funcionamiento de la comunicación por WebRTC en local, se inicia sesión con el especialista, a continuación, se inicia sesión con el usuario en una ventana de incógnito para tener sesiones separadas. Se une a una cita y se observa que la comunicación se establece correctamente.

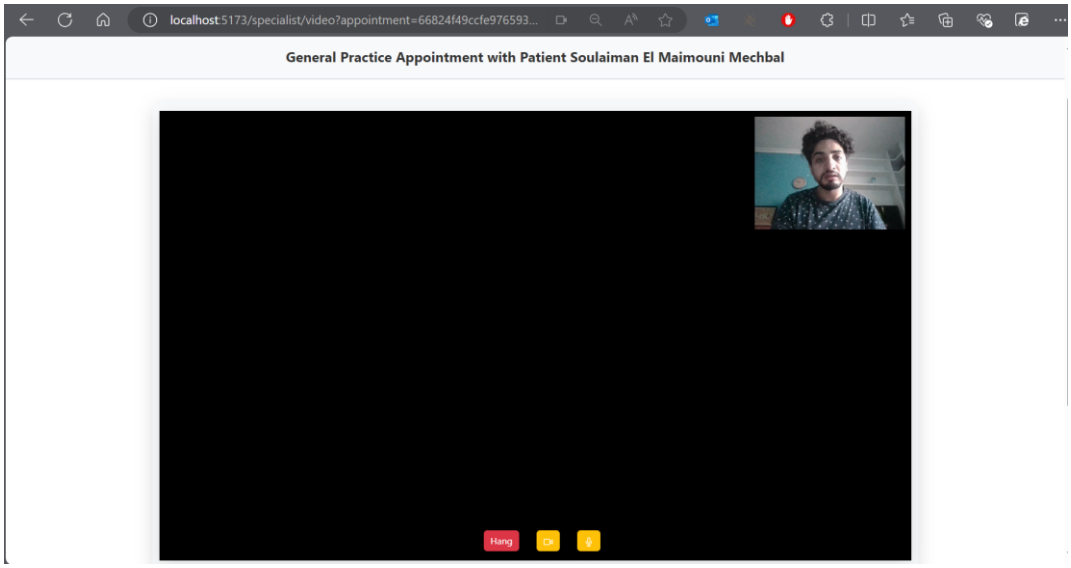


Figura 8-28. Inicialización de la videollamada por parte del especialista

Figura 8-29. Inicialización de la videollamada por parte del usuario

8.7 Seguridad

8.7.1 Autenticación y Autorización

En SEHHA, la autenticación y autorización se gestionan mediante tokens JSON Web Tokens (JWT). Cuando un usuario se registra o inicia sesión, el backend genera un token JWT que se almacena de forma segura en el localStorage del navegador. Este token se adjunta automáticamente a las cabeceras Authorization en todas las

solicitudes HTTP y en WebSocket como parte de la información que se envía al servidor. Esto permite que el backend valide no solo la identidad del usuario, sino también su rol, asegurando que solo los usuarios autorizados tengan acceso a ciertos recursos y funcionalidades.

Este enfoque garantiza una autenticación eficiente y segura, simplificando la gestión de sesiones y mejorando la experiencia del usuario al mantenerlo autenticado incluso después de recargar la página o cerrar y reabrir el navegador.

8.7.2 Protección de Datos

Para asegurar que todas las comunicaciones entre el frontend y los microservicios del backend sean seguras, SEHHA utiliza HTTPS, implementado con certificados SSL/TLS proporcionados por Amazon Certificate Manager (ACM). Esta medida garantiza que todos los datos sensibles transmitidos estén cifrados y protegidos contra posibles interceptaciones o ataques de intermediarios.

La utilización de HTTPS es fundamental para proteger la integridad y la confidencialidad de la información del usuario, especialmente en aplicaciones de telemedicina donde se manejan datos personales y médicos sensibles.

8.8 Despliegue

8.8.1 Dockerización

El frontend de SEHHA se construye y despliega utilizando Docker, lo que facilita la creación de entornos de desarrollo consistentes y escalables. El proceso de construcción de la imagen Docker se realiza en dos etapas. Primero, se construye la aplicación React con Vite, y luego se sirve con Nginx en un contenedor optimizado para producción.

8.8.2 Despliegue en Kubernetes

La imagen de Docker del frontend se despliega en un clúster de Kubernetes administrado por Amazon EKS. Utilizamos un Deployment para gestionar los pods y un LoadBalancer de AWS para manejar las conexiones HTTPS, lo que proporciona escalabilidad y alta disponibilidad. El dominio https://sehha-telemedicine.com se utiliza para acceder al frontend, y se configuran rutas seguras para cada microservicio del backend, asegurando una comunicación eficiente y segura.

8.9 Conclusión

El frontend de SEHHA está diseñado para ser intuitivo, responsivo y seguro. Utiliza tecnologías modernas como React y Vite, y se despliega en un entorno escalable y seguro en AWS EKS. La integración con los microservicios de backend se realiza mediante Axios y Socket.io, asegurando una comunicación eficiente y segura. Con un enfoque en la accesibilidad y la responsividad, el frontend proporciona una experiencia de usuario fluida y eficiente.

9 DESPLIEGUE DE LA APLICACIÓN SEHHA EN AWS

En esta sección se detalla el proceso completo de despliegue de la aplicación SEHHA en AWS. Esta explicación se basa en una arquitectura de microservicios y abarca desde la configuración inicial hasta la implementación y las medidas de seguridad adoptadas. Los archivos de configuración específicos utilizados en este proceso se encuentran detallados a lo largo de la sección.

9.1 Arquitectura General

El despliegue de la aplicación SEHHA en AWS se basa en una arquitectura de microservicios que proporciona una alta modularidad, escalabilidad y facilidad de mantenimiento. La infraestructura y los servicios de soporte se despliegan utilizando diversas tecnologías de AWS y servicios complementarios como MongoDB Atlas y CloudAMQP. A continuación, se presenta una visión general de los componentes involucrados en el despliegue:

9.1.1 Componentes Principales del Despliegue

9.1.1.1 Clúster Kubernetes en Amazon EKS:

EKS (Elastic Kubernetes Service) se utiliza para orquestar y gestionar los contenedores Docker que ejecutan los microservicios y el frontend. Los recursos dentro del clúster incluyen namespaces, deployments, services y secrets, configurados a través de archivos YAML.

9.1.1.2 Servicios de Gestión de Usuarios, Citas y Señalización de Vídeo:

Cada microservicio se despliega como un pod en el clúster EKS. Estos microservicios gestionan la autenticación y autorización de usuarios, la gestión de citas médicas y la señalización de vídeo para las videollamadas respectivamente.

9.1.1.3 Frontend de SEHHA:

Desarrollado en React y empaquetado como un contenedor Docker, el frontend proporciona la interfaz de usuario de la aplicación. Se sirve utilizando un servidor Nginx configurado dentro del contenedor.

9.1.1.4 MongoDB Atlas:

Cada microservicio utiliza una base de datos específica alojada en MongoDB Atlas, una plataforma de base de datos como servicio (DBaaS). Las conexiones a las bases de datos se gestionan mediante variables de entorno que contienen las URI de conexión.

9.1.1.5 CloudAMQP:

Se utiliza para la gestión de colas de mensajes con RabbitMQ, facilitando la comunicación asíncrona entre los microservicios.

9.1.1.6 Amazon Route 53 y AWS Certificate Manager (ACM):

Route 53 se utiliza para la gestión de DNS, proporcionando un dominio personalizado (sehha-telemedicine.com) y configurando las rutas necesarias para los servicios. ACM se encarga de la gestión de certificados SSL/TLS para asegurar la comunicación entre los usuarios y la aplicación a través de HTTPS.

9.2 Repositorios y Automatización con GitHub Actions

Cada microservicio y el frontend de SEHHA están gestionados en repositorios individuales en GitHub. A continuación, se detallan los repositorios utilizados y el proceso de automatización implementado mediante GitHub Actions para facilitar la integración continua y el despliegue continuo (CI/CD).

9.2.1 Repositorios de GitHub:

- Repositorio del Servicio de Gestión de Usuarios: https://github.com/souel22/tfg-sehha-user-managment
- Repositorio del Servicio de Gestión de Citas: https://github.com/souel22/tfg-sehha-appointment
- Repositorio del Servicio de Señalización de Vídeo: https://github.com/souel22/tfg-sehha-video
- Repositorio del Frontend de SEHHA: https://github.com/souel22/tfg-sehha-front

9.2.2 Workflows de GitHub Actions

Para cada uno de estos repositorios, se han configurado workflows de GitHub Actions que automatizan la creación y despliegue de contenedores Docker en Docker Hub. Los secretos necesarios para los workflows, como las credenciales de Docker Hub, se configuran a nivel de repositorio en GitHub. A continuación, se describe un ejemplo de workflow para el frontend de SEHHA, junto con su respectivo Dockerfile y configuración de Nginx. A continuación se muestra un ejemplo del workflow del frontend:

```yaml
name: Build, Test and Push Docker Image - React Vite App

on:
  push:
    branches:
      - main
      - feature/basic-frontend
  workflow_dispatch:

jobs:
  build-test-and-push:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build Docker image
        id: build-image
        uses: docker/build-push-action@v4
        with:
          context: .
          file: ./Dockerfile
          push: false
          platforms: linux/amd64, linux/arm64
          tags: ${{ secrets.DOCKER_USERNAME }}/sehha-frontend:latest
      - name: Push Docker image to Docker Hub
        if: success()
        uses: docker/build-push-action@v4
        with:
          context: .
          file: ./Dockerfile
          push: true
          platforms: linux/amd64, linux/arm64
          tags: ${{ secrets.DOCKER_USERNAME }}/sehha-frontend:latest
```

Figure 9-1. Workflow de Github Actions para automatizar la contenerización del frontend de SEHHA

El workflow se ejecuta en los eventos `push` a las ramas `main` y `feature/basic-frontend`, así como cuando se inicia manualmente mediante `workflow_dispatch`. Define un job llamado `build-test-and-push` que se ejecuta en un entorno `ubuntu-latest`.

- Primero realiza un checkout del código, se utiliza la acción `actions/checkout@v3` para clonar el repositorio en el entorno de ejecución.
- Se utiliza la acción `docker/setup-buildx-action@v2` para preparar el entorno para la construcción de imágenes Docker multiplataforma.
- Se autentica en Docker Hub utilizando las credenciales almacenadas en los secretos del repositorio.
- **Se** utiliza `docker/build-push-action@v4` para construir la imagen Docker sin subirla aún (`push: false`).
- Si la construcción es exitosa, utiliza nuevamente `docker/build-push-action@v4` para subir la imagen a Docker Hub.

9.2.3 Dockerfile para el Frontend:

```
# Stage 1: Build the React app
FROM node:20-alpine as build

WORKDIR /app

# Copy package.json and package-lock.json
COPY package.json package-lock.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Set environment variables
ARG NODE_ENV=production
ENV NODE_ENV=${NODE_ENV}

# Copy the production environment variables
COPY .env.production .env

# Build the application
RUN npm run build

# Stage 2: Serve the app with nginx
FROM nginx:alpine

COPY --from=build /app/dist /usr/share/nginx/html

COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Figure 9-2. Dokerfile del frontend de SEHHA

La dockerización del frontend de SEHHA se divide en dos etapas:

- Etapa 1 - Construcción de la App: Utiliza una imagen base de Node.js (`node:20-alpine`). Configura el directorio de trabajo, copia los archivos de configuración, instala dependencias y construye la aplicación React utilizando `npm run build`.
- Etapa 2 - Servir la App con Nginx: Utiliza una imagen base de Nginx (`nginx:alpine`). Copia los archivos construidos desde la etapa 1 al directorio de Nginx y configura el servidor para servir la aplicación.

9.2.4 Configuración de Nginx:

```
server {
    listen 80;
    server_name localhost;

    root /usr/share/nginx/html;
    index index.html;

    # Serve static files with caching
    location /static/ {
        alias /usr/share/nginx/html/static/;
        expires 1y;
        add_header Cache-Control "public";
    }

    # Fallback for client-side routing
    location / {
        try_files $uri /index.html;
    }

    # Logging configuration (optional)
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log warn;

    # Gzip compression settings (optional)
    gzip on;
    gzip_types text/plain text/css application/json application/javascript text/xml application/xml application/xml+rss text/javascript;
    gzip_min_length 256;
}
```

Figure 9-3. Configuración del servidor Nginx para servir el frontend de SEHHA

La configuración de Nginx establece el servidor para escuchar en el puerto 80, define "localhost" como el nombre del servidor y "/usr/share/nginx/html" como la raíz del documento. Configura la caché para archivos estáticos bajo "/static/" con una expiración de un año y añade un encabezado de control de caché. Utiliza "try_files" para servir "index.html" en todas las rutas, facilitando la navegación del lado del cliente. Además, define la ubicación de los archivos de logs de acceso y errores y habilita la compresión Gzip para varios tipos de archivos, mejorando el rendimiento de la transferencia de archivos.

Como demonstración del workflow, se realiza un commit a la rama `feature/basic-frontend` usando git bash y se muestra la ejecución del workflow en el Github Actions

```
[redacted] MINGW64 ~/git/sehha-defenitive/tfg-sehha-front (fe
ature/basic-frontend)
$ git add .

[redacted] MINGW64 ~/git/sehha-defenitive/tfg-sehha-front (fe
ature/basic-frontend)
$ git commit -m "workflow execution demo"
[feature/basic-frontend 1392bbb] workflow execution demo
 10 files changed, 27 insertions(+), 81 deletions(-)
 rename src/{pages/HomePage => components}/Footer/Footer.css (100%)
 rename src/{pages/HomePage => components}/Footer/Footer.jsx (100%)
 delete mode 100644 src/pages/SpecialistPages/Scheduler/Footer/Footer.css
 delete mode 100644 src/pages/SpecialistPages/Scheduler/Footer/Footer.jsx

[redacted] MINGW64 ~/git/sehha-defenitive/tfg-sehha-front (fe
ature/basic-frontend)
$ git push
Enumerating objects: 36, done.
Counting objects: 100% (36/36), done.
Delta compression using up to 8 threads
Compressing objects: 100% (19/19), done.
Writing objects: 100% (20/20), 1.65 KiB | 842.00 KiB/s, done.
Total 20 (delta 15), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (15/15), completed with 13 local objects.
To https://github.com/souel22/tfg-sehha-front.git
   d722b26..1392bbb  feature/basic-frontend -> feature/basic-frontend

[redacted] MINGW64 ~/git/sehha-defenitive/tfg-sehha-front (fe
ature/basic-frontend)
```

Figura 9-4. Commit a repositorio remoto souel22/tfg-sehha-front en la rama feature/basic-frontend

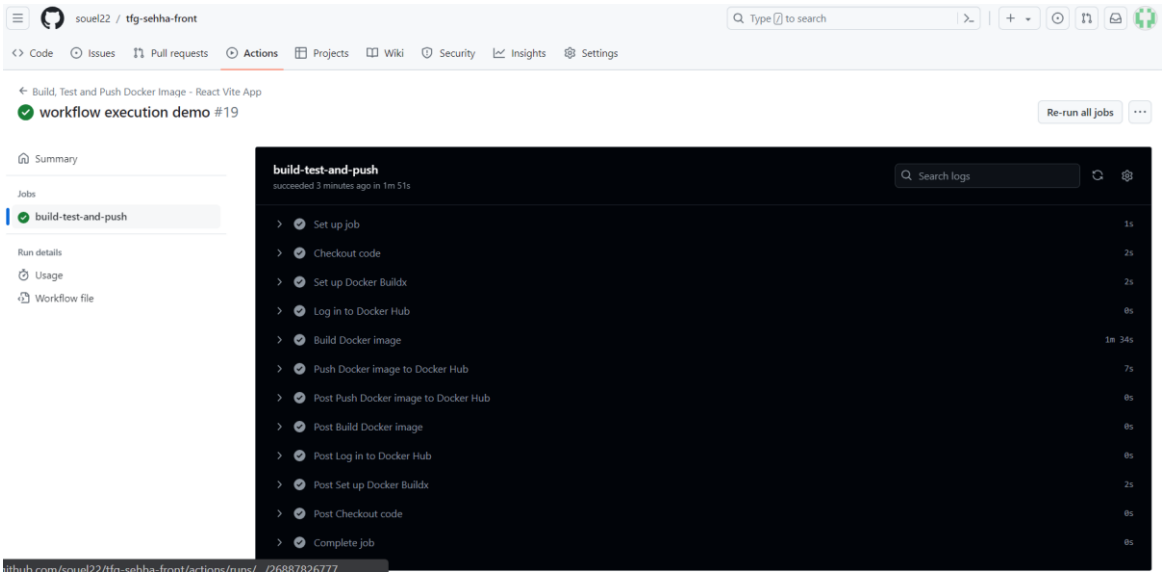


Figura 9-5. Ejecución del workflow en GitHub Actions.

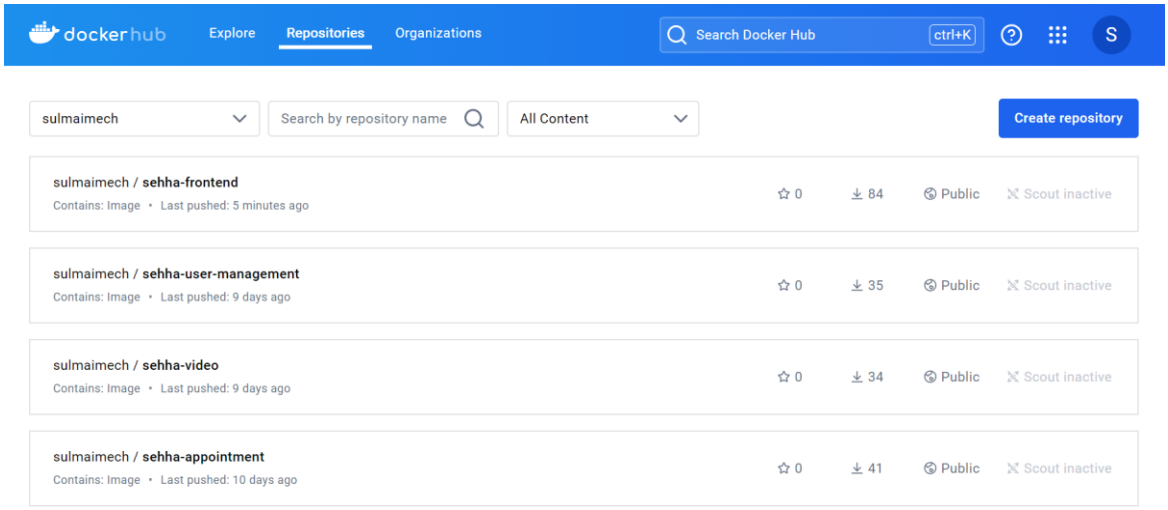


Figura 9-6. Contenedores en Docker Hub después del push y la ejecución de los workflows.

Esta configuración asegura un proceso de CI/CD robusto, automatizando la construcción, prueba y despliegue de las imágenes Docker para la aplicación SEHHA. En la siguiente sección, se detallará la configuración de la infraestructura en AWS.

9.3 Configuración de la Infraestructura

El despliegue de la infraestructura en AWS comienza con la creación de un clúster Kubernetes utilizando Amazon EKS (Elastic Kubernetes Service). Para facilitar esta tarea, se utiliza la herramienta eksctl, que permite automatizar y simplificar el proceso de configuración y gestión de clústeres de Kubernetes en AWS.

9.3.1 Creación del Clúster con eksctl

Para desplegar el clúster Kubernetes, se utiliza el siguiente comando en la AWS Cloud Shell:

```
eksctl create cluster --name sehha-cluster --version auto --region eu-west-2 --nodegroup-name sehha-nodes --node-type t4g.small --nodes 2 --nodes-min 2 --nodes-max 4 --managed --spot
```

Este comando crea un clúster EKS llamado sehha-cluster en la región eu-west-2 con un grupo de nodos gestionados y configurados para utilizar instancias t4g.small. El clúster puede escalar automáticamente entre 2 y 4 nodos según la carga, permitiendo una administración eficiente de los recursos.

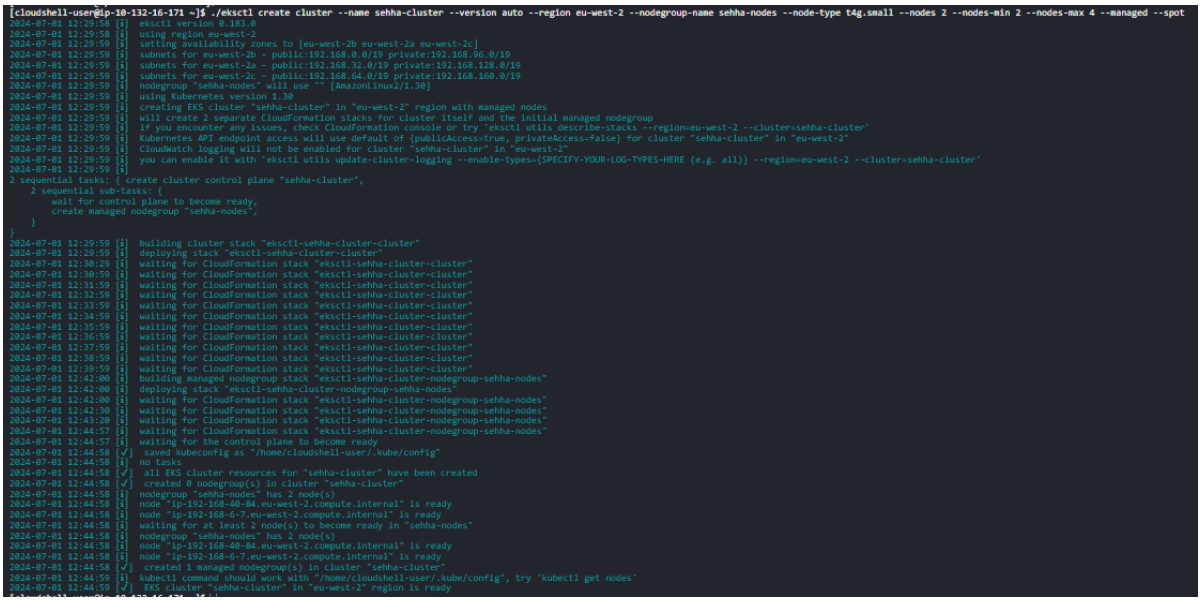


Figura 9-7. Ejecución del comando para crear el clúster EKS.

Para borrar el clúster, se puede usar el siguiente comando:

```
eksctl delete cluster --name sehha-cluster --region eu-west-2
```

9.3.2 Despliegue de Recursos con kubectl

Una vez creado el clúster, se utilizan archivos YAML y la herramienta kubectl para desplegar la infraestructura necesaria dentro del clúster. Esto incluye la creación de namespaces, secrets, deployments y load balancers. A continuación se detallan estos recursos con ejemplos específicos.Namespace: Define el espacio de nombres en el que se desplegarán los recursos de la aplicación.

9.3.2.1 Namespace

El namespace es un mecanismo de Kubernetes para organizar los recursos dentro del clúster. En este caso, se define un namespace específico para la aplicación SEHHA:

```
apiVersion: v1
kind: Namespace
metadata:
    name: sehha-app
```

Este archivo define un espacio de nombres llamado “`sehha-app`” donde se desplegarán todos los recursos relacionados con la aplicación.

9.3.2.2 Secrets

Los secrets en Kubernetes se utilizan para almacenar información sensible, como contraseñas y claves de API, de manera segura. A continuación, se muestra un ejemplo del archivo YAML utilizado para los secrets:

```
apiVersion: v1
kind: Secret
metadata:
    name: sehha-secrets
    namespace: sehha-app
```

```
  type: Opaque
  data:
      AUTH_DATABASE_PASSWORD: # variable para micro de gestion de
usuario
      AUTH_SECRET: # secreto que se usa para micro de gestion de
usuario
      PRIVATE_KEY: # clave privada que se usa para firmar los tokens
      PUBLIC_KEY: # clave publica que se usa para validar los tokens
      PASSPHRASE: # pass phrase que se usa para hashear las contraseñas
      CLOUDAMQP_URL: # url de cloudAMQP que se usa para conectarse a
rabbitMQ
      DATABASE_PASSWORD: # pass de la base de datos para el micro de
gestion de citas
      RABBITMQ_URL: # url de rabbitmq para el servicio de señalizacion
de video
      DB_PASSWORD: # pass de la base de datos para el servicio de
señalizacion de video
      JWT_PUBLIC_KEY: # clave publica que se usa para validar tokens
jwt en servicio de señalizacion de video
```

Este archivo almacena información sensible como contraseñas y claves de API de manera segura. Los datos se codifican en base64 para protegerlos.

9.3.2.3 Deployments

El archivo de Deployment define cómo se despliegan los contenedores de cada microservicio, especificando el número de réplicas y las imágenes de Docker a utilizar. A continuación se muestra un ejemplo del archivo YAML para el frontend:

```
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: sehha-frontend
    namespace: sehha-app
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: sehha-frontend
    template:
      metadata:
        labels:
          app: sehha-frontend
      spec:
        containers:
        - name: sehha-frontend
          image: sulmaimech/sehha-frontend:latest
          imagePullPolicy: Always
          ports:
          - containerPort: 80
```

Este archivo configura el despliegue del contenedor del frontend, utilizando la imagen sulmaimech/sehha-frontend:latest y especificando que se ejecute una réplica del contenedor.

9.3.2.4 Services

El archivo de Service define cómo se exponen los microservicios dentro del clúster y hacia el exterior. En este caso, se usa un servicio del tipo LoadBalancer para exponer el frontend al exterior:

```
  apiVersion: v1
  kind: Service
  metadata:
    name: sehha-frontend-service
   ```yaml
  apiVersion: v1
  kind: Service
  metadata:
    name: sehha-frontend-service
    namespace: sehha-app
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-ssl-cert:
#CERTIFICATE ARN
      service.beta.kubernetes.io/aws-load-balancer-backend-protocol:
"http"
      service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443"
      service.beta.kubernetes.io/aws-load-balancer-connection-idle-
timeout: "3600"
  spec:
    selector:
      app: sehha-frontend
    ports:
      - name: http
        protocol: TCP
        port: 80
        targetPort: 80
      - name: https
        protocol: TCP
        port: 443
        targetPort: 80
    type: LoadBalancer
```

Este archivo configura un servicio de tipo LoadBalancer para el frontend de la aplicación SEHHA. Las anotaciones indican el certificado SSL a utilizar, el protocolo backend, los puertos SSL y el tiempo de espera de la conexión. El servicio expone el frontend a través de los puertos 80 y 443, permitiendo acceso tanto HTTP como HTTPS.

Para facilitar el despliegue se usa un Shell script para crear los recursos en EKS:

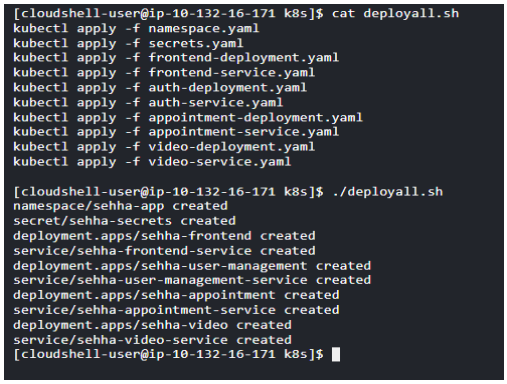


Figura 9-8. Despliegue de los recursos en EKS
```

En la siguiente captura se observa que se han desplegado todos los recursos en el cluster EKS, cabe destacar que los servicios que se han desplegados son de tipo LoadBalancer, y que muestran dominios externos que se pueden usar para interactuar con los servicios tanto con HTTP como con HTTPS. Posteriormente se usará el servicio de DNS de AWS Route53 para redirigir desde los dominios de SEHHA a los dominios creados por AWS para los balanceadores de carga.

```
[cloudshell-user@ip-10-132-16-171 k8s]$ kubectl -n sehha-app get all
NAME                                        READY   STATUS    RESTARTS   AGE
pod/sehha-appointment-658d687fc8-s6k2v      1/1     Running   0          3m15s
pod/sehha-frontend-86d998f69c-67pk5         1/1     Running   0          3m21s
pod/sehha-user-management-5b76d6bb8b-pl5tm  1/1     Running   0          3m18s
pod/sehha-video-c55d957c5-97s8g             1/1     Running   0          3m13s

NAME                                  TYPE           CLUSTER-IP       EXTERNAL-IP                                                                PORT(S)                      AGE
service/sehha-appointment-service     LoadBalancer   10.100.225.228   ad821582f9fec44f8988165dbd0cef97-1963966983.eu-west-2.elb.amazonaws.com     80:30987/TCP,443:31810/TCP   3m14s
service/sehha-frontend-service        LoadBalancer   10.100.8.117     a88046a9b0b7a425d992b51e73be3a6f-1635004787.eu-west-2.elb.amazonaws.com     80:31667/TCP,443:32284/TCP   3m19s
service/sehha-user-management-service LoadBalancer   10.100.162.175   aec5fccfc1f38402b9cc199b5ec2fd62-413579620.eu-west-2.elb.amazonaws.com      80:32453/TCP,443:31695/TCP   3m17s
service/sehha-video-service           LoadBalancer   10.100.62.55     ae0dc8b32d4074d2daab845d9a697831-1206057147.eu-west-2.elb.amazonaws.com     80:30524/TCP,443:32252/TCP   3m11s

NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/sehha-appointment      1/1     1            1           3m15s
deployment.apps/sehha-frontend         1/1     1            1           3m21s
deployment.apps/sehha-user-management  1/1     1            1           3m18s
deployment.apps/sehha-video            1/1     1            1           3m13s

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/sehha-appointment-658d687fc8      1         1         1       3m15s
replicaset.apps/sehha-frontend-86d998f69c         1         1         1       3m21s
replicaset.apps/sehha-user-management-5b76d6bb8b  1         1         1       3m18s
replicaset.apps/sehha-video-c55d957c5             1         1         1       3m13s
```

Figura 9-9. Recursos en el namepace sehha-app desplegados en EKS

Si accedemos la dirección correspondiente al balanceador de cargas del frontend, vemos que la página es accesible

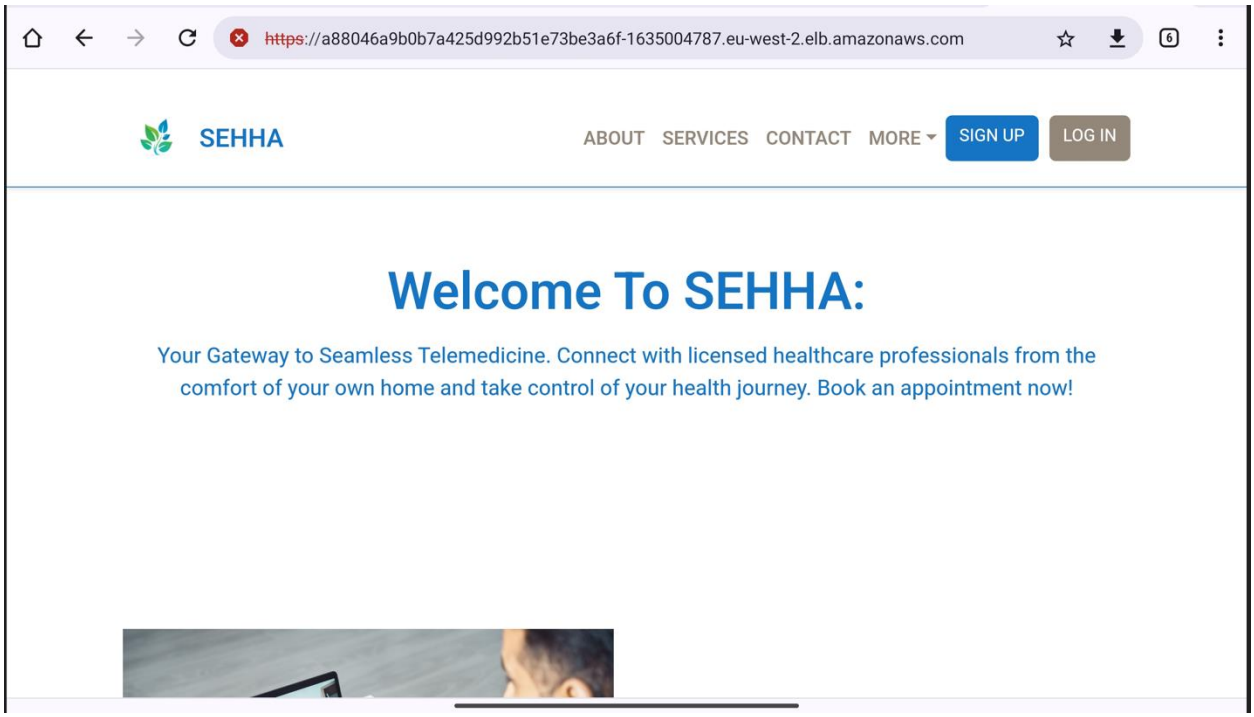


Figura 9-10. SEHHA accesible a través del balanceador de cargas

9.4 Configuración de DNS y Certificados con Route 53 y ACM

Para asegurar que los usuarios puedan acceder a la aplicación de manera segura y eficiente, se configura Route 53 y ACM como se describe a continuación:

9.4.1 Compra del Dominio y Configuración de DNS

El dominio `sehha-telemedicine.com` fue adquirido por un costo de 14 dólares anuales sin impuestos. Este dominio actúa como la dirección principal para acceder a la aplicación SEHHA.

9.4.1.1 Creación de Hosted Zone

En la consola de Route 53, se crea una Hosted Zone para `sehha-telemedicine.com`. Una Hosted Zone en Route 53 actúa como un contenedor para los registros DNS del dominio. Permite gestionar y configurar la resolución de nombres para el dominio y sus subdominios.

9.4.1.2 Actualización de Name Servers

Los name servers proporcionados por Route 53 se actualizan en el registrador del dominio. Estos name servers son responsables de la resolución de DNS para `sehha-telemedicine.com`.

9.4.1.3 Creación de Registros DNS

Se crean los registros DNS necesarios para dirigir el tráfico a los diferentes balanceadores de carga (Load Balancers) que manejan el frontend y los microservicios. Los registros se configuran de la siguiente manera:

- `sehha-telemedicine.com`: Este registro apunta al DNS del balanceador de carga del frontend.
- `auth.sehha-telemedicine.com`: Apunta al balanceador de carga del microservicio de gestión de usuarios.
- `book.sehha-telemedicine.com`: Dirige al balanceador de carga del microservicio de gestión de citas.
- `video.sehha-telemedicine.com`: Apunta al balanceador de carga del microservicio de señalización de vídeo.

Para simplificar la configuración, se utiliza un registro Alias si se emplean balanceadores de carga de AWS. Los registros Alias permiten direccionar a recursos específicos de AWS como balanceadores de carga, eliminando la necesidad de especificar direcciones IP estáticas.

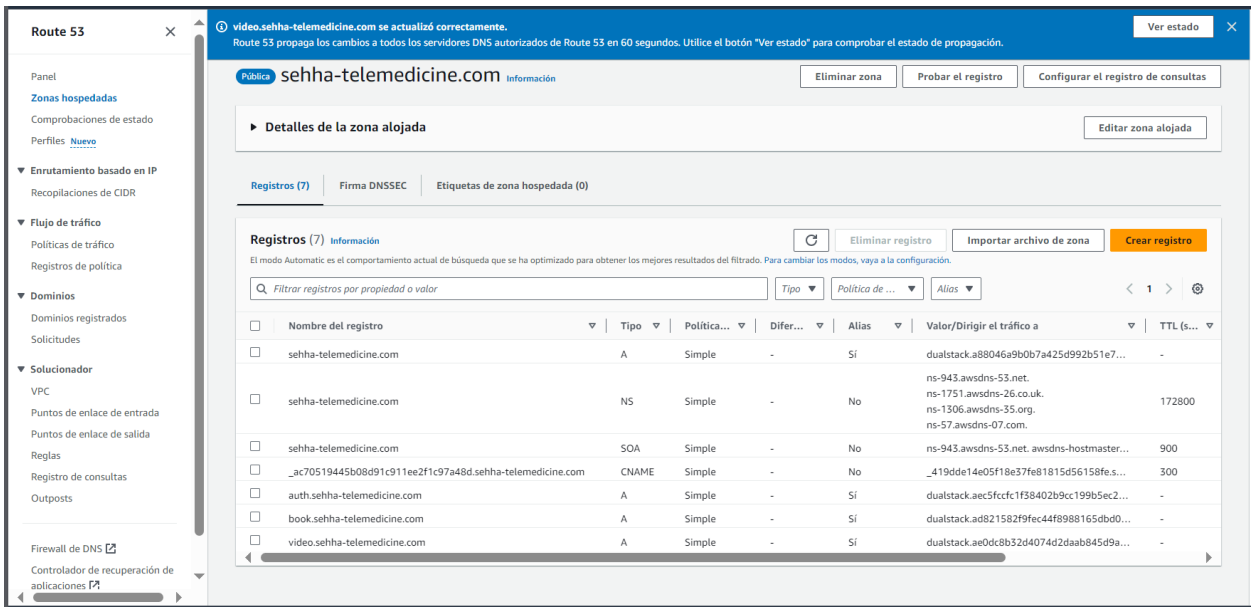


Figura 9-11. Configuración de registros DNS en Route 53.

9.4.1.4 9.4.2 Solicitud y Validación de Certificado en ACM

Para garantizar la seguridad de las comunicaciones, se configura un certificado SSL/TLS en AWS Certificate Manager (ACM).

9.4.1.5 Solicitud de Certificado

En la consola de ACM, se solicita un certificado público para `sehha-telemedicine.com` y `*.sehha-`

`telemedicine.com` (subdominios comodín). Este certificado asegura que todas las comunicaciones hacia el dominio y sus subdominios estén encriptadas y seguras.

9.4.1.6 Validación del Certificado

ACM proporciona registros CNAME que deben ser añadidos a Route 53 para validar el dominio. Estos registros demuestran que se tiene control sobre el dominio al validar la solicitud del certificado.

Una vez que los registros CNAME se añaden correctamente a Route 53 como se observa en la figura 9-11 y se propagan, ACM valida la propiedad del dominio y el estado del certificado cambia a "Issued" como se muestra en la siguiente figura. Este proceso asegura que el dominio y todos sus subdominios estén protegidos por el certificado SSL/TLS.

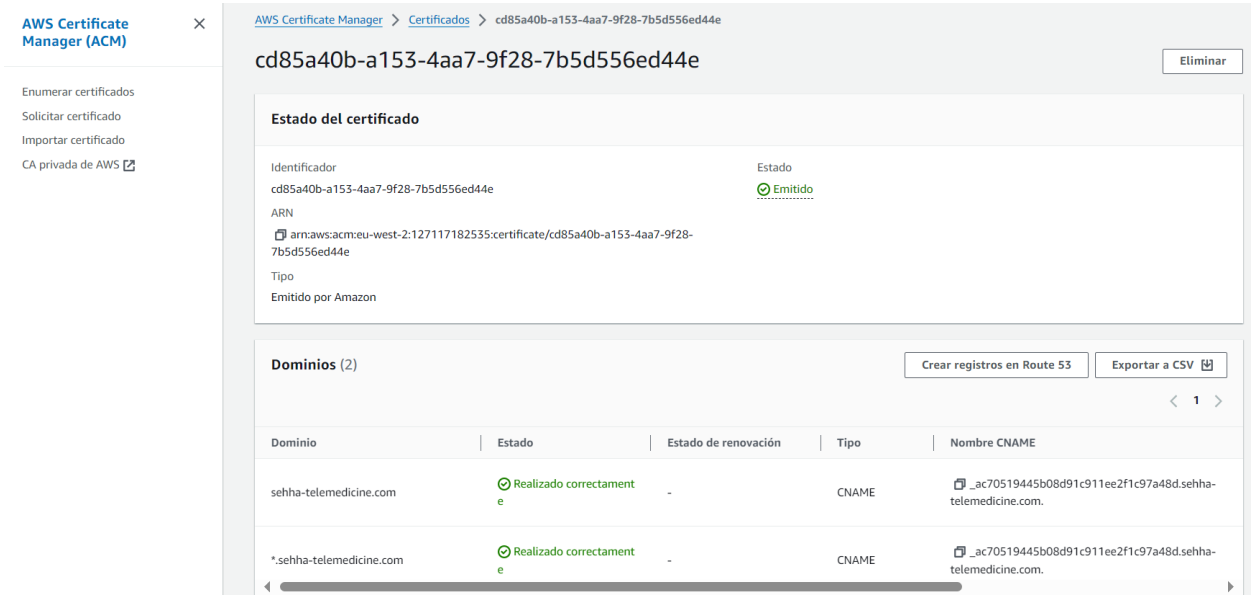


Figura 9-12. Solicitud y validación de certificado en ACM.

El ARN del certificado generado por ACM se utiliza en los Load Balancers para configurar el acceso HTTPS, asegurando así que todas las conexiones sean seguras y cumplan con los estándares de cifrado modernos.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: sehha-video-service
  namespace: sehha-app
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert: "arn:aws:acm:eu-west-2:127117182535:certificate/cd85a40b-a153-4aa7-9f28-7b5d556ed44e"
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "http"
    service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443"
    service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout: "3600"
```

Figura 9-13. Uso del ARN del certificado en la configuración del Load Balancer.

Esta configuración asegura que todas las interacciones con la aplicación SEHHA se realicen de manera segura, protegiendo la integridad y confidencialidad de los datos de los usuarios.

9.5 Gestión de Datos

La gestión de datos en SEHHA se realiza mediante el uso de bases de datos independientes para cada microservicio, alojadas en MongoDB Atlas. MongoDB Atlas es un servicio de base de datos como servicio (DBaaS) que facilita la gestión, escalabilidad y seguridad de los datos.

9.5.1 Arquitectura de la Base de Datos

Cada microservicio de SEHHA tiene su propia base de datos, lo que garantiza la separación de datos y permite una gestión más eficiente y segura. Esta arquitectura modular facilita la escalabilidad y el mantenimiento de la

aplicación, ya que los datos de cada microservicio se gestionan de manera independiente.

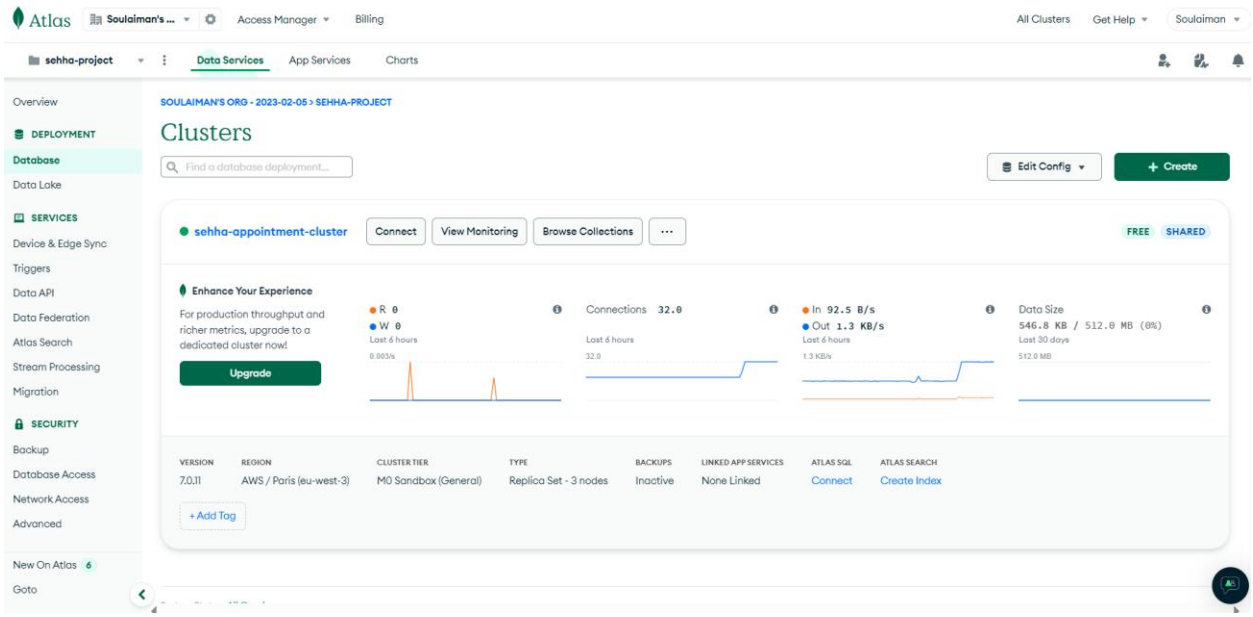


Figura 9-14. Cluster "sehha-appointement-cluster" en el que se alojan las bases de datos de SEHHA

En MongoDB Atlas, se crea un clúster que aloja las bases de datos para los diferentes microservicios:

- Base de Datos de Gestión de Usuarios (authentication)**:** Almacena información de usuarios, credenciales y tokens de autenticación.
- Base de Datos de Gestión de Citas (appointements)**:** Contiene información sobre las citas médicas, horarios y disponibilidad de los especialistas.
- Base de Datos de Señalización de Vídeo (videos)**:** Gestiona los datos relacionados con la señalización de vídeo y las sesiones de videollamada.

Cada una de estas bases de datos está configurada con sus propias credenciales de acceso y URI de conexión, las cuales se almacenan como variables de entorno en los microservicios correspondientes.

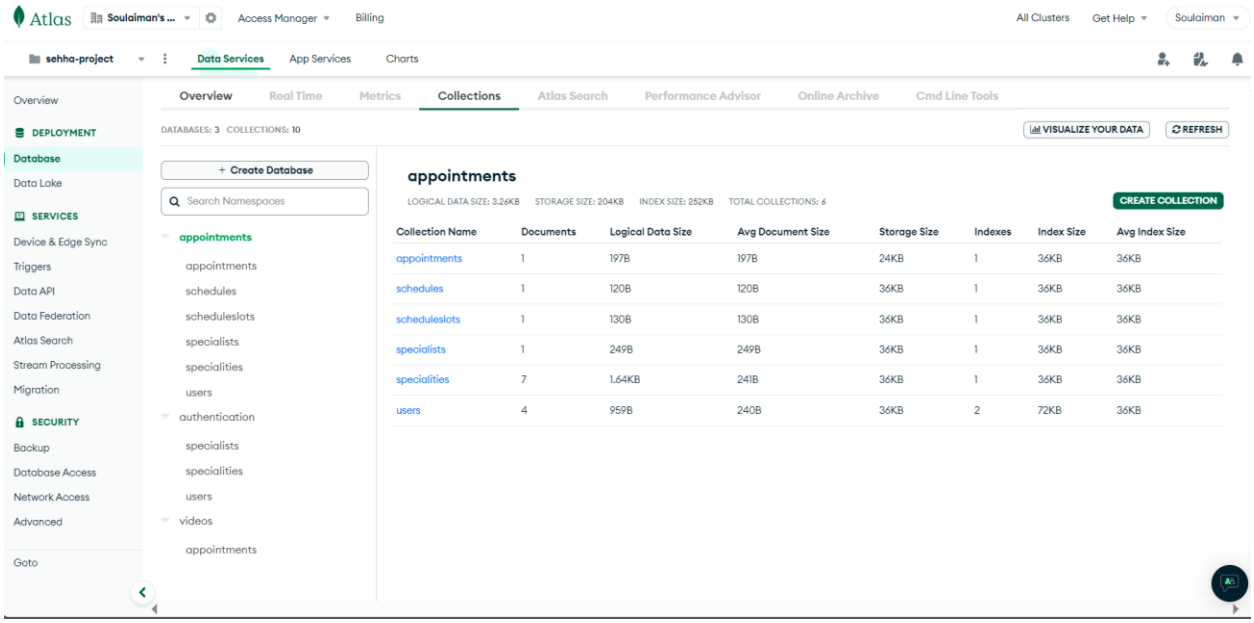


Figura 9-15. Estructura de Clúster en MongoDB Atlas mostrando las bases de datos independientes para cada microservicio.

9.5.2 Conexión y Seguridad de las Bases de Datos

Las conexiones a las bases de datos en MongoDB Atlas se gestionan mediante variables de entorno que contienen las URI de conexión. Estas variables se configuran en los archivos de configuración de los microservicios, asegurando que las credenciales sensibles no se exponen en el código fuente.

```
env:
    - name: DATABASE_URI
      valueFrom:
        secretKeyRef:
          name: sehha-secrets
          key: DATABASE_URI
```

9.5.3 Uso de RabbitMQ en CloudAMQP

Además de MongoDB Atlas, SEHHA utiliza RabbitMQ para la gestión de colas de mensajes, alojado en CloudAMQP. RabbitMQ facilita la comunicación entre microservicios, permitiendo el intercambio de mensajes de manera eficiente y confiable.

Cada microservicio se conecta a las colas de RabbitMQ utilizando las URI's proporcionadas por CloudAMQP, las cuales también se gestionan mediante variables de entorno.

```
env:
    - name: CLOUDAMQP_URL
      valueFrom:
        secretKeyRef:
          name: sehha-secrets
          key: CLOUDAMQP_URL
```

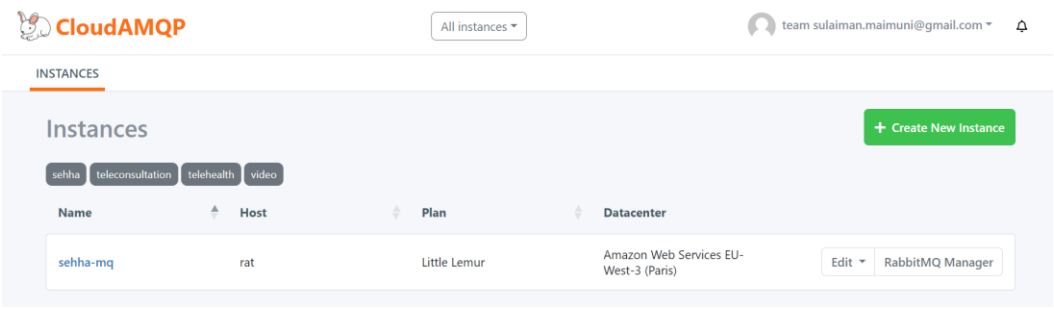


Figura 9-16. Instancia "sehha-mq" en la nube para alojar las colas de RabbitMQ de SEHHA

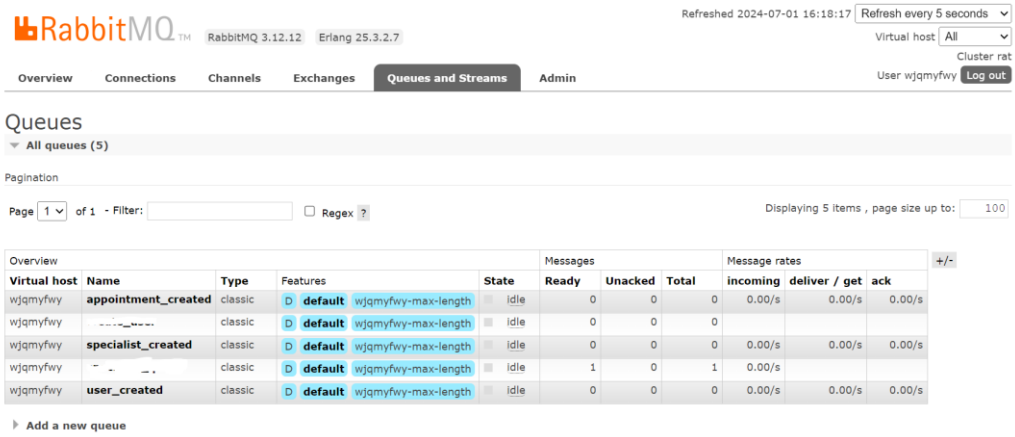


Figura 9-17. Colas de RabbitMQ de SEHHA en RabbitMQ Manager

9.6 Seguridad

La seguridad en SEHHA es primordial y se garantiza a través de varias medidas, incluyendo el uso de HTTPS para todas las comunicaciones y la gestión segura de certificados SSL/TLS.

9.6.1 Certificados SSL/TLS

Para asegurar la comunicación entre los componentes de la aplicación y con los usuarios finales, se utiliza HTTPS. Los certificados SSL/TLS se gestionan a través de AWS Certificate Manager (ACM), lo que asegura que todas las conexiones a los balanceadores de carga estén cifradas y protegidas.

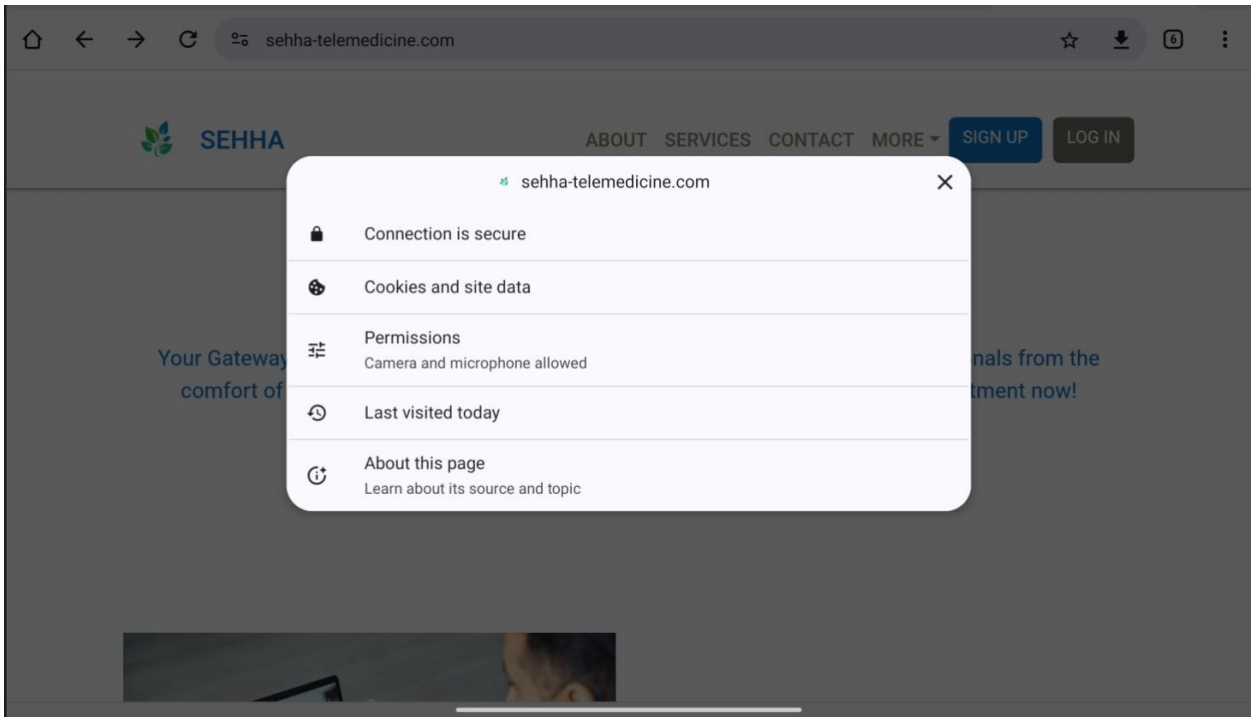


Figura 9-18. Certificado SSL/TLS activo y verificado en el navegador, indicando una conexión segura.

9.7 Pruebas

Para verificar que la infraestructura se ha desplegado correctamente, se inicia una llamada entre un usuario y un especialista y se confirma que la conexión se establece de manera exitosa.

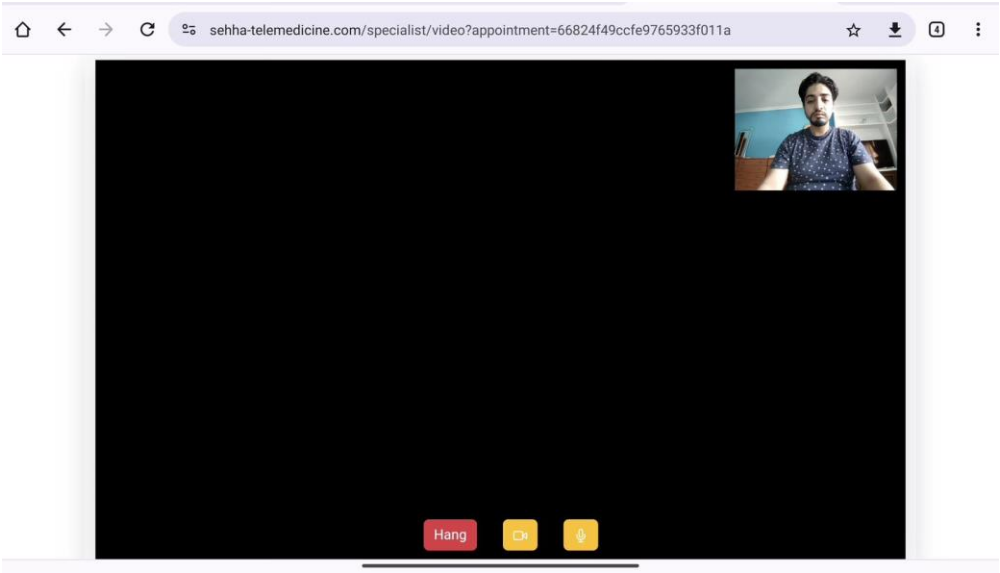


Figura 9-19. Inicio de llamada por parte de un especialista en un dispositivo de tipo Tablet

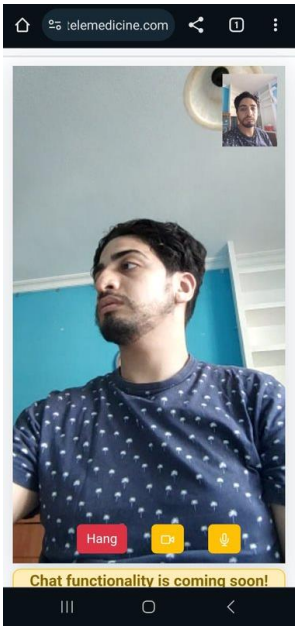


Figura 9-20. Inicio de llamada por parte de un usuario en un dispositivo de tipo Smartphone

9.8 Conclusión

El despliegue de la aplicación SEHHA en AWS se basa en una arquitectura de microservicios que garantiza modularidad, escalabilidad y seguridad. La configuración del clúster Kubernetes mediante eksctl, junto con la automatización de CI/CD a través de GitHub Actions, facilita un flujo de trabajo eficiente y robusto. Cada microservicio está encapsulado en contenedores Docker, gestionados en Docker Hub y orquestados en el clúster Kubernetes, asegurando una gestión eficiente de los datos mediante bases de datos en MongoDB Atlas y colas de mensajes en CloudAMQP. La seguridad se refuerza mediante el uso de certificados SSL/TLS de AWS Certificate Manager, garantizando la protección de todas las comunicaciones. Esta combinación de tecnologías y prácticas permite a SEHHA ofrecer una plataforma confiable y segura, capaz de manejar la complejidad de un sistema de telemedicina, asegurando la integridad y disponibilidad de los datos y proporcionando una experiencia de usuario óptima.

10 CONCLUSIONES Y LÍNEAS FUTURAS

La implementación del proyecto SEHHA ha establecido una base sólida para una plataforma de teleconsulta robusta y escalable, utilizando una arquitectura de microservicios en AWS. A lo largo del desarrollo, se han abordado los objetivos planteados inicialmente, logrando avances significativos en la creación de un sistema que facilita la interacción entre pacientes y profesionales de la salud de manera remota. No obstante, se han identificado varias áreas de mejora y expansión que pueden ser desarrolladas en el futuro.

10.1 Evaluación del Cumplimiento de Objetivos

10.1.1 Objetivo General

El objetivo principal era diseñar e implementar una aplicación de teleconsulta basada en una arquitectura de microservicios, permitiendo consultas médicas remotas, gestión de citas y administración de usuarios de manera segura y eficiente. Este objetivo se ha cumplido al desarrollar una plataforma funcional que responde a la necesidad de soluciones de teleconsulta, destacada durante la pandemia de COVID-19. La arquitectura de microservicios adoptada permite escalabilidad, flexibilidad y mantenibilidad del sistema, mejorando la accesibilidad a servicios médicos y la eficiencia en la gestión de citas y comunicaciones entre pacientes y médicos.

10.1.2 Objetivos Específicos

10.1.2.1 Desarrollo del Microservicio de Autenticación y Gestión de Usuarios

Se implementaron mecanismos de registro, inicio de sesión y gestión de perfiles.

La seguridad de los datos de los usuarios se garantizó mediante la implementación de JWT para la autenticación.

Este objetivo se cumplió, asegurando que solo usuarios autorizados puedan acceder al sistema, protegiendo datos sensibles y manteniendo la integridad del sistema.

10.1.2.2 Implementación del Microservicio para la Gestión de Citas

Se permitió a los usuarios reservar, modificar y cancelar citas.

Se gestionó la disponibilidad de los profesionales de la salud y los slots de tiempo.

Este objetivo se logró, facilitando la organización y administración de consultas médicas, mejorando la eficiencia y accesibilidad para los pacientes y médicos.

10.1.2.3 Desarrollo del Microservicio de Señalización WebRTC

Se facilitó la comunicación en tiempo real entre pacientes y médicos.

Se integraron funcionalidades de videollamadas y chat en la aplicación.

Este objetivo se alcanzó, permitiendo la comunicación directa y en tiempo real, una característica esencial para las teleconsultas.

10.1.2.4 Diseño e Implementación del Frontend Utilizando React y React Bootstrap

Se creó una interfaz de usuario intuitiva y accesible.

Se aseguró la responsividad y usabilidad de la aplicación en diferentes dispositivos usando Bootstrap.

Este objetivo se cumplió, asegurando una experiencia de usuario positiva y eficiente, fundamental para la adopción y uso continuo de la aplicación.

10.2 Líneas Futuras

10.2.1 Mejora de la Interfaz de Usuario

Aunque la interfaz actual es funcional, existe potencial para hacerla más atractiva y fácil de usar, reflejando mejor la marca SEHHA. Además, se planea desarrollar aplicaciones nativas para Android y iOS, reutilizando las APIs de los microservicios existentes. Establecer un canal de feedback una vez en producción permitirá recibir opiniones de los usuarios y mejorar continuamente la experiencia.

10.2.2 Integración con Otros Sistemas de Salud

Una extensión natural del servicio de gestión de usuarios es la implementación de un sistema de Gestión de Historia Clínica Electrónica (EHR). Esta funcionalidad podría integrarse con estándares como FHIR o HL7 para el intercambio de información con otros sistemas hospitalarios, facilitando una interoperabilidad eficiente y segura.

10.2.3 Escalabilidad Global

Para mejorar la escalabilidad global de SEHHA, se puede considerar la simplificación de la estructura de microservicios. Aunque el enfoque actual permite una gestión modular y eficiente, una estructura simplificada podría ser más manejable en una etapa inicial. La utilización de infraestructuras en la nube seguirá siendo un pilar fundamental para escalar la aplicación según la demanda.

10.2.4 Funcionalidades Avanzadas de Telemedicina

El potencial para implementar funcionalidades avanzadas como la monitorización remota de pacientes y el análisis de datos médicos mediante inteligencia artificial es vasto. Estas herramientas podrían proporcionar un valor añadido significativo, mejorando la calidad del servicio ofrecido por SEHHA.

10.2.5 Seguridad y Cumplimiento

Cumplir con las normativas GDPR y HIPAA es crucial para operar en los mercados europeo y estadounidense. En este sentido, se seguirán implementando y actualizando las medidas de seguridad para asegurar que la aplicación cumple con todas las regulaciones relevantes, protegiendo la información sensible de los usuarios.

10.2.6 Optimización del Rendimiento

La optimización del rendimiento de la aplicación es un área continua de desarrollo. Desplegar funciones en la nube y aprovechar la infraestructura cloud puede mejorar significativamente la escalabilidad y accesibilidad de la aplicación. Las estrategias específicas para optimizar el rendimiento y reducir la latencia en las comunicaciones WebRTC incluirán el uso de servidores STUN/TURN y la mejora de la infraestructura de red.

10.2.7 Ampliación del Soporte Multiplataforma

El soporte multiplataforma es una prioridad, con planes para desarrollar aplicaciones nativas para dispositivos móviles Android y iOS. Esto permitirá a los usuarios acceder a SEHHA de manera más conveniente y desde cualquier lugar, mejorando la accesibilidad y usabilidad de la plataforma.

10.2.8 Extensión del Alcance

SEHHA no solo se centrará en usuarios y especialistas individuales, sino que también buscará ampliar su infraestructura para incluir centros de salud y hospitales. Esto permitirá una mayor integración y coordinación entre diferentes tipos de profesionales de salud física, mental y social. Además, se añadirá un rol de administrador para gestionar usuarios y otras funcionalidades críticas del sistema.

REFERENCIAS

[1] Elendu, C., Egbunu, E. O., Opashola, K. A., Afuh, R. N., & Adebambo, S. A. (2023). The Role of Telemedicine in Improving Healthcare Outcome: A Review. Advances in Research, 24(5), 1-10.

Disponible en: https://journalair.com/index.php/AIR/article/download/958/1911

[2] Ford, D. W., & Valenta, S. R. (Eds.). (2021). Telemedicine: overview and application in pulmonary, critical care, and sleep medicine (1st ed.). Cham, Switzerland: Springer.

[3] Singh, J., Albertson, A., & Sillerud, B. (2022). Telemedicine during COVID-19 Crisis and in Post-Pandemic/Post-Vaccine World—Historical Overview, Current Utilization, and Innovative Practices to Increase Utilization. Healthcare, 10(6), 1041.

[4] Google Cloud. (s. f.). Portal Telemedicina Case Study. Disponible en: https://cloud.google.com/customers/portal-telemedicina-gcp

[5] SEC.gov. (s. f.). 10-K.

Disponible en: https://www.sec.gov/Archives/edgar/data/1393584/000095017023003945/amwl-20221231.htm

[6] Kalagiakos, P., & Karagiannaki, T. (2023). Architecting and Building the Future of Healthcare Informatics: Cloud, Containers, Big Data and CHIPS. International Journal of Environmental Research and Public Health, 20(5), 345.

Disponible en: https://saiconference.com/Downloads/FTC2017/Proceedings/33_Paper_468-Architecting_and_Building_the_Future_of_Healthcare.pdf

[7] Pierleoni, P., Pernini, L., Belli, A., Palma, L., & Valerio, A. (2020). An Innovative WebRTC Solution for e-Health Services. IEEE Access, 8, 225314-225323. Disponible en: https://ieeexplore.ieee.org/abstract/document/7749444/

[8] Padovilla, M. M., Gómez, J. M., & Rosales, F. M. (2022). An Ambient Assisted Living Architecture for Hospital at Home Coupled with a Process-Oriented Perspective. Journal of Ambient Intelligence and Humanized Computing, 13(2), 689-703.

Disponible en: https://link.springer.com/article/10.1007/s12652-022-04388-6

[9] Mantri, A., Kumar, V., Mantri, R., & Ratnam, K. (2021). Telemedicine for Emergency Care Management Using WebRTC. IEEE Transactions on Emerging Topics in Computing, 9(2), 794-805.

Disponible en: https://ieeexplore.ieee.org/abstract/document/7275865/

[10] Hasselbring, W. (2018). Software architecture: Past, present, future. The Essence of Software Engineering.

[11] Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. IEEE Access.

[12] Auer, F., Lenarduzzi, V., Felderer, M., & Taibi, D. (2021). From monolithic systems to microservices: An assessment framework. Information and Software Technology.

[13] Korotenko, A. (2024). Microservices Architecture: practical implementations, benefits, and nuances.

[14] Fedosejev, A. (2015). React.js essentials: a fast-paced guide to designing and building scalable and maintainable web apps with React.js (1st ed.). Birmingham, England: Packt Publishing.

[15] Matsinopoulos, P. (2020). Practical Bootstrap.

Disponible en: https://link.springer.com/content/pdf/10.1007/978-1-4842-6071-5.pdf

[16] Brown, E. (2019). Web Development with Node and Express: Leveraging the JavaScript Stack.

[17] Bradshaw, S., Brazil, E., & Chodorow, K. (2019). MongoDB: the definitive guide: powerful and scalable data storage (3rd ed.). Sebastopol, CA: O'Reilly Media, Inc.

[18] Videla, A., & Williams, J. J. W. (2012). RabbitMQ in action: distributed messaging for everyone (1st ed.). Shelter Island, New York: Manning Publications.

[19] Cadenhead, T. (2015). Socket.IO cookbook: over 40 recipes to help you create real-time JavaScript applications using the robust Socket.IO framework (1st ed.). Birmingham: Packt Publishing.

[20] WebRTC. (2024). WebRTC official documentation. WebRTC.org

[21] Martin, R. C. (2012). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall

[22] Implementers of WebRTC. WebRTC for the Curious. [Online].

Disponible en: https://webrtcforthecurious.com/