



CARRERA DE ESPECIALIZACIÓN EN INTERNET DE LAS COSAS

MEMORIA DEL TRABAJO FINAL

Desarrollo de Aplicación Hospitalaria con MQTT

Autor:

Ing. Gustavo Adrián Bastian

Director:

Mg. Ing. Ericson Joseph Estupiñan Pineda (Surix S.R.L)

Jurados:

Mg. Ing. Sergio Burgos (UTN, Facultad Regional Paraná)
Esp B. Sc. Daniel Iván Cruz Flores (UBA, Facultad de Ingeniería)
Esp. Ing. Luis Mariano Campos (Conicet)

*Este trabajo fue realizado en la Ciudad de Santo Tomé, Santa Fé,
entre Octubre de 2021 y Diciembre de 2022.*

Resumen

La presente memoria describe el desarrollo de un sistema para ser utilizado por enfermeros y médicos en el ámbito hospitalario implementado con un protocolo de bajo consumo de recursos. El trabajo se realizó para satisfacer una necesidad de la empresa Surix SRL.

La arquitectura del sistema está compuesta por cinco elementos principales que fueron abordados durante la carrera: un broker que gestiona los mensajes, una base de datos con información de los pacientes, un programa que genera acciones basado en distintos eventos, una aplicación web que permite la gestión de la base de datos y una aplicación móvil.

Agradecimientos

En primer lugar agradezco al director de este trabajo final, Mg. Ing. Ericson Estupiñán por la guía y los consejos brindados durante este tiempo.

Así mismo, agradezco a Sergio Starkoff por proporcionar la idea originaria del trabajo y por la confianza ofrecida desde un primer momento.

No puedo olvidar agradecer a los docentes, por el incalculable valor de realizar muy buenos contenidos para las clases y por la excelente predisposición para responder mis dudas.

Agradezco también a mis compañeros/as por demostrar mucho interés en las asignaturas y participar activamente en las clases.

Especialmente agradezco a mi familia, que comprendieron el tiempo que debía dedicarle al estudio, brindándome apoyo moral y humano. Y a Lucila y Alejo, quienes me brindaron mucho amor y ternura.

Finalmente, agradezco a Emilio y Alicia por toda la ayuda brindada.

Índice general

Resumen	I
Agradecimientos	III
1. Introducción general	1
1.1. Internet de las Cosas y las actividades hospitalarias	1
1.2. Motivación	2
1.3. Estado de arte	3
1.4. Objetivos y alcance	3
2. Introducción específica	5
2.1. Descripción de tecnologías web para IoT	5
2.1.1. Breve introducción a tecnologías de backend	5
2.1.2. Breve introducción a tecnologías frontend	6
2.2. Componentes del sistema	6
2.3. Descripción del protocolo MQTT	8
2.3.1. Uso de WebSockets en MQTT	10
2.3.2. Broker Mosquitto	10
2.4. Descripción del protocolo HTTP	10
2.5. Sistemas de contenedores Docker	11
2.5.1. Uso de Docker-compose	12
2.6. Introducción a las bases de datos	13
2.6.1. Descripción de MySQL	13
2.7. Frameworks/librerías para desarrollo web/móvil	15
2.7.1. Librerías Node.js, Express y JWT	15
2.7.2. Framework Ionic/Angular, Capacitor y Android Studio	16
3. Diseño e implementación	17
3.1. Generación del entorno base para el desarrollo del sistema de backend	17
3.2. Broker Mosquitto	17
3.3. Sistema Docker	18
3.4. Base de datos del sistema	18
3.5. Sistema de gestión	22
3.5.1. Descripción de las clases para monitoreo del sistema	23
3.5.2. Descripción de API MQTT para la mensajería de la aplicación móvil	25
3.5.3. Descripción de API Rest para aplicación Web	27
3.6. Página Web	31
3.6.1. Estructura y organización del software	32
3.6.2. Acceso de usuario	33
3.6.3. Monitoreo del sistema	35
3.6.4. Gestión de tareas programadas	35

3.6.5. Estadísticas del sistema	36
3.7. Aplicación Móvil	37
3.7.1. Estructura y organización del software	37
3.7.2. Configuración del broker y acceso de usuario	39
3.7.3. Modo administrador	40
3.7.4. Modo médico	41
3.7.5. Modo enfermera	42
4. Ensayos y resultados	47
4.1. Pruebas unitarias	47
4.1.1. Pruebas del servidor Mosquitto	47
4.1.2. Pruebas unitarias de la API Rest	48
4.2. Integración del sistema	50
4.2.1. Instalación del sistema en una instancia de AWS	50
4.2.2. Generación/instalación de la aplicación móvil en Android	52
4.2.3. Equipo simulador de llamadores	53
4.2.4. Resultados de las pruebas de integración	53
4.3. Contraste con los requerimientos	54
5. Conclusiones	57
5.1. Resultados obtenidos	57
5.1.1. Cumplimiento de planificación original	57
5.1.2. Gestión de riesgos	58
5.1.3. Recursos aprendidos durante la carrera:	58
5.2. Próximos pasos	58
A. Descripción Simulador de llamador	61
B. Configuración de servidor Nginx	63
C. Compilación para aplicaciones Android	65
Bibliografía	69

Índice de figuras

1.1. Infraestructura hospitalaria.	1
1.2. Componentes del sistema.	4
2.1. División del sistema.	7
2.2. Esquema de un sistema MQTT.	9
2.3. Uso de WebSockets con MQTT.	10
2.4. Pila de comunicaciones en un navegador.	11
2.5. Estructura general de Docker. ¹	12
2.6. Página de gestión phpMyAdmin.	14
3.1. Tabla Usuarios.	18
3.2. Tabla Camas.	19
3.3. Tabla pacientes.	19
3.4. Relación médicos-pacientes.	20
3.5. Relación notas-pacientes.	21
3.6. Tabla de eventos programados.	21
3.7. Tabla de registro de eventos.	22
3.8. Estructura de carpetas (backend).	23
3.9. Página web.	31
3.10. Estructura de carpetas de la aplicación web.	32
3.11. Monitoreo de camas.	35
3.12. Tareas programadas.	36
3.13. Relación paciente tratamiento.	36
3.14. Estructura de carpetas de la aplicación móvil.	37
3.15. Inicio de sistema.	39
3.16. Pantallas de administración.	40
3.17. Diagrama de estados en modo médico.	41
3.18. Modo médico.	42
3.19. Diagrama de estados en modo enfermera.	43
3.20. Recepción de tarea	44
3.21. Captura de QR en la aplicación.	44
3.22. Ejecución de tarea.	45
4.1. Imagen de MQTT Explorer.	47
4.2. Logueo en el sistema con Postman.	48
4.3. Rechazo de logueo.	48
4.4. Colección test de logueo.	49
4.5. Reporte en consola de Newman.	50
4.6. Simulador de llamadores.	53

Índice de tablas

1.1. Modelo de capas IoT	2
2.1. Comandos SQL	14
2.2. Tabla 1 Ejemplo SQL	15
2.3. Tabla 2 Ejemplo SQL	15
3.1. Tipos de mensajes MQTT	27

***Dedicado a todas las personas que participaron en mi
formación humana.***

Capítulo 1

Introducción general

En este capítulo se presentan las necesidades de los sistemas hospitalarios junto con nociones sobre el Internet de las cosas y el protocolo MQTT. Además, se mencionan las motivaciones, el estado de arte y el alcance del trabajo.

1.1. Internet de las Cosas y las actividades hospitalarias

En la actualidad, el avance de la Internet de las Cosas (IoT de *Internet of Things*) y la disminución de costos asociados a la tecnología hacen factible su incorporación a distintos contextos de la vida cotidiana. Un campo de mucho interés es el de infraestructuras hospitalarias inteligentes [1].

El término Internet de las Cosas fue utilizado por primera vez en 1990 para describir un sistema cuyos componentes del mundo físico se conectaban a la Internet mediante sensores. En la actualidad se utiliza el término para referirse a escenarios donde la conectividad en red y la capacidad de cómputo se extiende a objetos, sensores y elementos cotidianos no considerados computadoras, permitiéndoles generar, intercambiar y consumir datos con un mínimo de intervención humana [2].

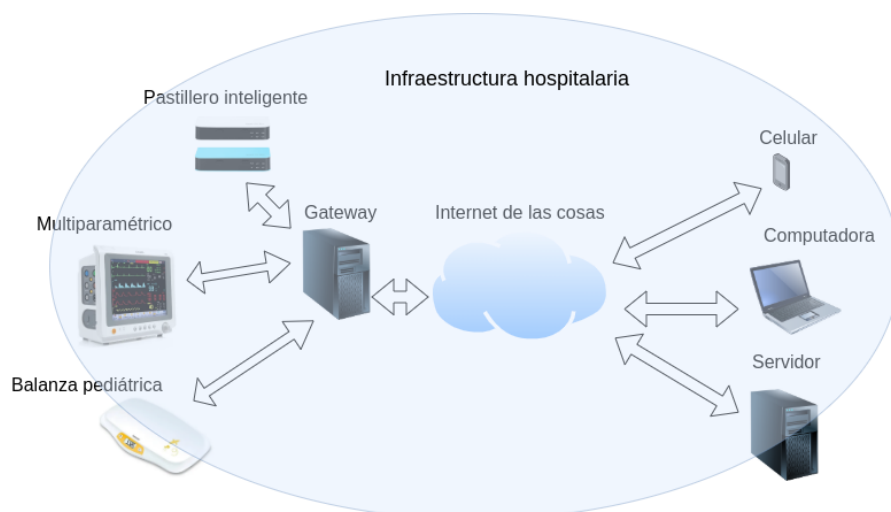


FIGURA 1.1. Infraestructura hospitalaria.

Si consideramos a las diferentes personas como un elemento más de un sistema, el ámbito hospitalario cuenta con numerosas entidades que interactúan entre sí como ser: médicos, pacientes, enfermeros, personal de limpieza, personal de

seguridad, personal administrativo, dispositivos de iluminación, dispositivos sonoros, herramientas médicas, puertas, ventanas, ventiladores, acondicionadores de aire, termómetros, etc. Un entorno de estas cualidades es ideal para gestionar con IoT, ya que permite mejorar la calidad del servicio de la salud [1].

Los principales beneficios de aplicar Internet de las Cosas son dos [1]:

- Cuidado clínico: el uso de sensores inteligentes permite monitorear el estado de los pacientes automáticamente. Como ejemplo se puede mencionar: cada quince minutos tomar la presión arterial del paciente y guardarla en una base de datos, cada una hora medir la temperatura y almacenar su valor en una base de datos, etc. Con los datos obtenidos se pueden generar mejores diagnósticos y tratamientos. El hecho de automatizar la gestión de los datos y la utilización de algoritmos de Inteligencia Artificial para los diagnósticos disminuye el tiempo de análisis y la posibilidad de errores. El objetivo final es efectuar tratamientos preventivos en momentos iniciales de la presencia de síntomas.
- Monitoreo remoto: otro beneficio muy importante radica en el hecho de que al estar almacenado en una red (o en la nube), la disponibilidad de la información facilita la consulta de la misma por parte de los médicos en todo momento, permitiendo actuar de manera eficiente y rápida ante eventos.

Un sistema de estas características posee un modelo de capas, presentado en [3] que separa los conocimientos en cinco categorías con responsabilidades bien definidas: negocio, aplicación, procesamiento, red y percepción. La tabla 1.1 presenta las funciones reducidas de cada modelo:

TABLA 1.1. Modelo de capas sistema IoT.

Capa	Función
Negocio	Establecer reglas y controlar sistema
Aplicación	Interactuar con el usuario
Procesamiento	Almacenar y analizar los datos obtenidos
Red	Transportar datos entre dispositivos
Percepción [3]	Realizar mediciones o acciones

1.2. Motivación

En Argentina muchos hospitales están retrasados en su progreso tecnológico. Por dicha razón, todos los avances en este campo son necesarios. Por lo explicado en la sección 1.1, el gestionar la institución con un sistema de IoT es de suma utilidad.

Surix S.R.L fabrica un sistema IP de llamado a enfermera que está basado en el protocolo SIP. Este consiste en un servidor central y terminales que se encuentran en las habitaciones del hospital. La aplicación principal se ejecuta en una computadora o bien en una tablet y monitorea el estado de las habitaciones. La principal motivación para migrar el protocolo radica en el alto costo de hardware que genera el agregar dispositivos a su sistema actual. En este contexto, se encargó la realización de este trabajo.

1.3. Estado de arte

En el mercado internacional se encontró un producto similar desarrollado por la firma TigerConnect (antes llamada TigerText), *TigerConnect Clinical Collaboration Platform* (Plataforma de Colaboración Clínica TigerConnect) [4], cuyas principales características se detallan a continuación:

- Aplicación de Mensajería para celulares y estaciones de trabajo.
- Solución en la nube asegurando disponibilidad en un 99.99 %.
- Mensajería por texto asegurada con encriptación.
- Homologado por HIPPA (del inglés, *Health Insurance Portability and Accountability Act*, ley de Portabilidad y Responsabilidad del Seguro Médico) [5].
- Certificado HITRUST (es un framework para gestionar riesgos utilizado por muchas redes de salud y hospitales [6]).
- Control administrativo total, permitiendo a los administradores gestionar usuarios, configuraciones y políticas de seguridad por medio de una consola. Los usuarios pueden ser cargados utilizando plantillas csv, y en caso de robo o extravío de dispositivo, prohibir el acceso.
- Posibilidad de incorporar mensajería de voz como servicio extra.
- Mensajes a grupos de personas.

Entre las diferencias que posee la solución presentada en este trabajo con respecto a la disponible en el mercado se encuentra el hecho de utilizar MQTT como protocolo base permite incorporar con muy poco esfuerzo dispositivos IoT de mediciones paramétricas de los pacientes ya que la estructura de la red así lo permite. Por otra parte, la solución desarrollada presenta de base la posibilidad de transmisión de audio.

1.4. Objetivos y alcance

El sistema resultante de este trabajo está orientado a gestionar las relaciones entre pacientes, enfermeras y médicos. Un diagrama reducido puede observarse en la figura 1.2.

Los componentes del sistema son:

- Broker MQTT: permite gestionar los mensajes entre los elementos del sistema.
- Base de datos: donde alojar información de reportes de habitaciones, personal y datos relevantes al paciente.
- Página web para configuración: permite gestionar usuarios, camas, pacientes, observar el estado del sistema.
- Aplicación móvil multiplataforma: permite a los usuarios enfermeros interactuar con el sistema y con los médicos. La aplicación es capaz de identificar la cama correspondiente (mediante lectura de símbolos QR) y de transmitir mensajes de voz en caso de ser necesario.

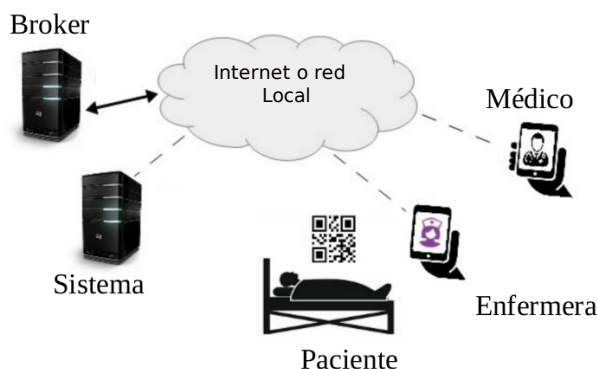


FIGURA 1.2. Componentes del sistema.

El alcance del trabajo consiste en:

- Confección de un plan de trabajo.
- Selección y configuración de un broker MQTT.
- Desarrollo local de una página web de configuración, que permite asignar médicos a pacientes, asignar tareas programadas a pacientes, asignar códigos QR a camas, asignar tratamientos a pacientes, asignar especialidades a los usuarios enfermeros.
- Desarrollo local de una aplicación móvil con 3 modos de funcionamiento: modo médico, con envío/recepción de audio/texto/alarmas, modo enfermera con envío/recepción de audio/texto/alarmas y escaneo de QR para identificar al paciente o cama, y modo sistema que permite el monitoreo de habitaciones.
- Código documentación de las aplicaciones realizadas.

El presente trabajo no incluye:

- Manuales de las distintas aplicaciones.
- Traducciones a distintos idiomas de las aplicaciones.
- Sistema llamador.
- Análisis de tráfico en la red.
- Análisis de seguridad (no se certifica).
- Contratación de base de datos remota.
- Contratación e instalación de servidores remotos.

Capítulo 2

Introducción específica

En este capítulo se presentan las bases teóricas que sustentan el trabajo realizado. Se describen distintas soluciones a cada una de las partes del sistema.

2.1. Descripción de tecnologías web para IoT

Los sistemas web utilizan (en su mayoría) una arquitectura cliente/servidor en donde se subdividen las tareas a desarrollar en dos partes. El cliente solicita una acción y el servidor la ejecuta. Las acciones pueden ser obtener información para mostrar en una pantalla, modificar un valor en una base de datos, obtener información para reproducir un sonido o un video, etc. En la programación web, se suele dividir el modelo en dos partes: backend y frontend.

2.1.1. Breve introducción a tecnologías de backend

El backend generalmente se encarga de almacenar y gestionar la información [3]. En el contexto de este trabajo, el backend se comunica con los clientes utilizando dos protocolos: *Message Queuing Telemetry Transport* (en adelante MQTT) y HTTP. El primero se utilizó para interactuar con los clientes móviles y el segundo se utilizó para brindar los servicios a la página web de configuración.

El protocolo MQTT [7] se ha probado como un protocolo confiable, altamente eficiente y ampliamente utilizado en sistemas IoT [8].

MQTT es un protocolo open source simple, liviano y orientado a dispositivos con pocos recursos y baja velocidad de transmisión. Está basado en la pila TCP/IP (del inglés *Transmission Control Protocol/Internet Protocol*, Protocolo de control de transmisión /Protocolo de Internet), se implementa en la capa de aplicación y utiliza mensajería bajo el patrón de publicación/subscripción. Las ventajas que provee es la posibilidad de agregar dispositivos rápidamente con bajo costo de software, hardware e implementación [8].

En la referencia [8] se realiza un estudio en profundidad del desempeño de varios protocolos de IoT: MQTT, CoAP (*Constrained Application Protocol*, Protocolo de Aplicaciones Restringido), AMQP (*Advanced Message Queuing Protocol*, Protocolo de Encolamiento de Mensajes Avanzado) y HTTP (*Hypertext Transfer Protocol*, Protocolo de Transferencia de Hipertexto). Se menciona que MQTT es ampliamente utilizado pero muy poco estandarizado en comparación con los demás y se resalta el hecho que tiene limitaciones en lo referido a seguridad.

En las subsecciones 2.6, 2.5, 2.5.1, 2.7.1 se explicará el funcionamiento del backend.

2.1.2. Breve introducción a tecnologías frontend

El frontend es la parte del programa o del dispositivo con la que interactúa el usuario. Son tecnologías que se ejecutan en un navegador o en una aplicación de teléfono celular (también llamado teléfono inteligente, o *smartphone*). El equipo que desarrolla el frontend consiste en programadores diseñadores UX/UI (del inglés *User Experience / User Interface*, Experiencia de Usuario/Interfaz de Usuario) y diseñadores gráficos. Las tecnologías más conocidas que se utilizan son: el lenguaje de programación Javascript, HTML (*HyperText Markup Language*, Lenguaje de Marcas de Hipertexto) y CSS (*Cascading Style Sheets*, Hojas de Estilo en Cascada) [3]. Si se desea profundizar en estas tecnologías, existen muchos sitios web donde se ofrecen referencias de las mismas (ver, por ejemplo [9]).

En los últimos años, se busca facilitar el diseño del frontend mediante librerías como React [10], JQuery [11] o Frameworks como Vue [12], Angular [13] e Ionic [14]. En este trabajo se seleccionó la combinación Ionic/ Angular para el desarrollo de la página Web y de la aplicación móvil. En la sección 2.7.2, se presenta una breve introducción.

2.2. Componentes del sistema

Al sistema desarrollado está compuesto por seis partes:

- Broker MQTT: servidor encargado de gestionar los mensajes.
- Base de datos: almacena información relevante de los pacientes, usuarios y sistema.
- Servidor de base de datos: proporciona una API (*Application Programming Interface*, interfaz de programación de aplicaciones) del tipo REST (*representational state transfer*, transferencia de estado representacional) que permite la modificación/consulta de la base de datos desde un dispositivo web. La API provee *endpoints* (puntos de acceso en español) que permiten realizar distintas operaciones como ser carga, consulta, alteración o eliminación de información.
- Servidor web: entrega información a un browser para que pueda mostrar una página web de administración del sistema y de carga de la base de datos.
- Backend: recibe los mensajes MQTT y responde/actúa acorde a las peticiones.
- Cliente browser: muestra en la pantalla de una computadora o dispositivo la página web de administración.
- Cliente teléfono: permite a médicos y enfermeras interactuar con el sistema.

La solución es un sistema distribuido cuyas partes se pueden ubicar en distintas ubicaciones.

Con el objetivo de facilitar el desarrollo se utilizó un sistema de contenedores que permite correr tres de las seis partes: la base de datos, el servidor de base de datos y el sistema propiamente dicho. Además, se agrupó los subsistemas que

modifican la base de datos en un solo módulo que provee ambos puntos de acceso, MQTT y API Rest. En la figura 2.1 se presenta un esquema general según esta división.

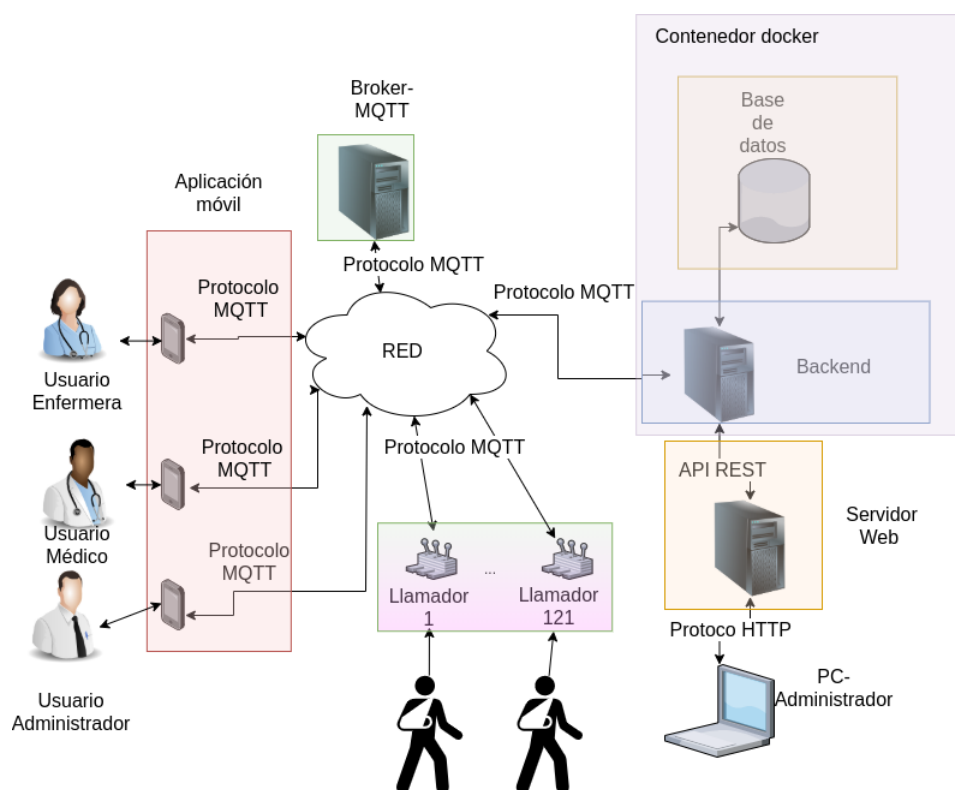


FIGURA 2.1. División del sistema.

2.3. Descripción del protocolo MQTT

MQTT fue introducido en 1999 por IBM y Eurotech. Es un protocolo de mensajería por suscripción/publicación especialmente diseñado para la comunicación M2M (*Machine to Machine*, Máquina a Máquina) en dispositivos con bajos recursos [7].

Para poder explicar el funcionamiento del protocolo es necesario incorporar conceptos definidos en la especificación [7]:

- Mensaje: datos transmitidos mediante el protocolo MQTT a través de la red por la aplicación. Cuando un mensaje es transmitido, el mismo contiene datos útiles ("*payload*"), un campo que identifica la Calidad de Servicio (*Quality of Service*, QoS), ciertas propiedades y un nombre de tópico.
- Cliente: programa o dispositivo que utiliza MQTT.
- Servidor o Broker: programa o dispositivo que actúa como intermediario entre clientes que publican mensajes y clientes que han realizado suscripciones.
- Sesión: una interacción entre cliente y servidor con estados bien definidos.
- suscripción: una suscripción implica un filtrado de tópicos y una máxima calidad de servicio. Una suscripción está asociada a una sola sesión y una sesión puede poseer varias suscripciones.
- Nombre de tópico: etiqueta que se adjunta a un mensaje la cual es comparada con las suscripciones conocidas en el servidor.
- Filtro de tópico: es una expresión contenida dentro de la suscripción para indicar interés, por parte del cliente, en uno o más tópicos.
- Suscripción con comodín o *Wildcard*: es una expresión contenida dentro de la suscripción que contiene un carácter especial o comodín, como ser '+' o '#', que representan un nivel único y nivel múltiple respectivamente.

En este protocolo, el funcionamiento es el siguiente:

Al iniciar la conexión, el cliente envía un mensaje CONNECT al broker y este contesta con un CONNECTACK y un código de estado. El broker mantiene abierta la conexión hasta que el cliente envía un comando de desconexión o la conexión se pierde por algún motivo. Los puertos estándar son 1883 para comunicación no encriptada y 8883 para comunicación encriptada utilizando TLS/SSL [3].

Luego el cliente MQTT publica mensajes en un broker MQTT, los cuales son recibidos por los clientes suscriptos o bien retenidos para una futura suscripción [7]. En la figura 2.2 se presentan todos los componentes de una red MQTT:

La conexión MQTT siempre es entre cliente y broker, nunca entre clientes directamente.

Cada mensaje es publicado en una dirección conocida como "tópico". Los clientes pueden suscribirse a múltiples tópicos y recibir todos los mensajes publicados en cada tópico. MQTT es un protocolo binario y normalmente requiere un encabezado fijo de dos *bytes* y un dato útil ("*payload*") de hasta 256 MB [8].

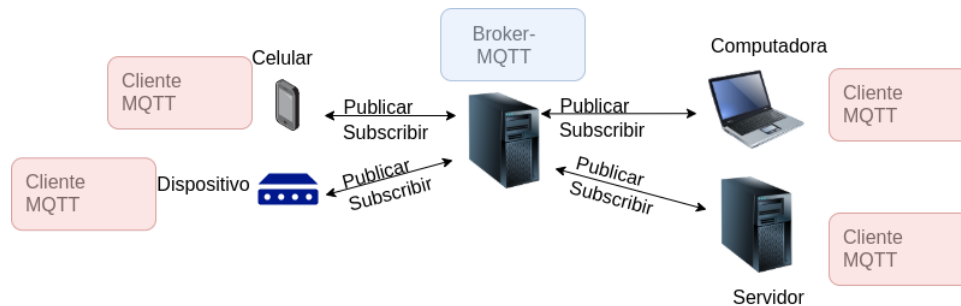


FIGURA 2.2. Esquema de un sistema MQTT.

Ejemplos de tópicos:

- 1: /user/1/question
- 2: /user/1/#
- 3: /user/+ /question

En el ejemplo 1, en caso de suscribirse a ese tópico, el cliente recibirá todos los mensajes publicados en el subtópico "question". En el ejemplo 2, el cliente recibirá todos los mensajes dirigidos al usuario 1. En el ejemplo 3, el cliente que se suscriba a ese tópico recibirá los mensajes enviados a "question" independientemente del número de usuario.

Utiliza usualmente TCP/IP como protocolo de transporte y puede implementar TLS/SSL para seguridad. Por lo que el servicio cliente-broker es del tipo orientado a la conexión. También puede utilizar otros protocolos de red con soporte bidireccional y sin pérdidas de datos [3][7].

Por otro lado, el protocolo posee tres niveles de calidad de servicio ("QoS", del inglés *Quality of Service*) que permite seleccionar la fiabilidad de la entrega de los mensajes [7][15]:

- Calidad 0: el mensaje se entrega como máximo una vez, sino no se entrega. Es el modo más rápido de entrega.
- Calidad 1: el mensaje se entrega como mínimo una vez. El mensaje se almacena localmente en el emisor y el receptor hasta que se procese. Si no se confirma la recepción se envía de nuevo con el distintivo "DUP" establecido hasta que se reciba acuse de recibo.
- Calidad 2: el mensaje se entrega exactamente una sola vez. El mensaje se almacena localmente en el emisor y el receptor hasta que se procese. El emisor recibe un mensaje acuse de recibo. Si el emisor no recibe un acuse de recibo, el mensaje se envía de nuevo con el distintivo DUP establecido hasta que se reciba un acuse de recibo.

El receptor puede procesar el mensaje proporcionado en la primera o segunda fase, no tiene que volver a procesarlo. Si el receptor es un intermediario, publica el mensaje a los suscriptores. Si el receptor es un cliente, el mensaje se entrega a la aplicación de suscriptor. El receptor devuelve un mensaje de finalización al emisor para comunicarle que ha terminado de procesar el mensaje.

2.3.1. Uso de WebSockets en MQTT

Actualmente no es posible utilizar MQTT natural en un *browser* o navegador debido a la imposibilidad de abrir una conexión TCP pura.

Una solución a este problema puede ser transmitir el mensaje MQTT sobre una conexión WebSocket lo que permite que un *browser* pueda utilizar todas las características del protocolo directamente. Esto es muy útil en el caso de las aplicaciones de teléfono móvil [16].

WebSocket es un protocolo de red que provee comunicación bidireccional entre un *browser* y un servidor web. El protocolo fue estandarizado en 2011 y es soportado por todos los navegadores modernos. Como MQTT, el protocolo WebSocket está basado en TCP. En la figura 2.3 se muestra conceptualmente como se encapsula la información del protocolo MQTT dentro de una trama WebSocket:

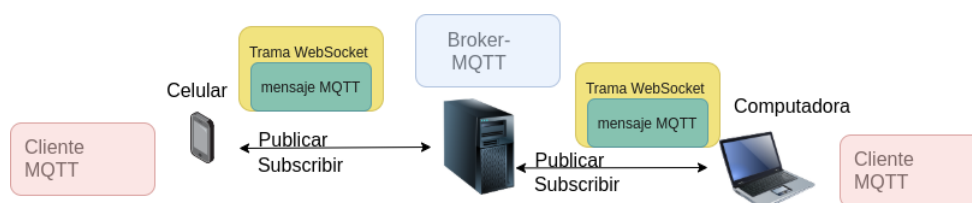


FIGURA 2.3. Uso de WebSockets con MQTT.

En MQTT sobre WebSockets, el mensaje MQTT se transfiere a través de la red y es encapsulado por una o más tramas WebSockets. La ventaja de este método de transmisión es que provee una comunicación bi-direccional, ordenada y sin pérdidas [16].

Para poder utilizar MQTT sobre WebSockets el Broker debe ser capaz de manejar WebSocket nativos.

Para mejorar la seguridad, se puede implementar TLS utilizando WebSockets seguros para encriptar toda la conexión.

2.3.2. Broker Mosquitto

Existen muchos brokers MQTT disponibles. Para este trabajo se seleccionó Eclipse Mosquitto como Broker ya que es de código abierto con licencia EPL/EDL que implementa MQTT en sus versiones 5.0, 3.1.1 y 3.1. [17].

Otra característica que lo hace muy conveniente es que se encuentra disponible en los repositorios de varias distribuciones de Linux.

2.4. Descripción del protocolo HTTP

HTTP es un protocolo que nos permite realizar una petición de datos y recursos como ser documentos HTML que pueden mostrarse en navegadores. Es la base de cualquier intercambio en la red [18]. En la figura 2.4 se observa donde se ubica el protocolo dentro del modelo de comunicación TCP/IP:

HTTP se basa en el modelo cliente-servidor: las peticiones son enviadas por una entidad al servidor, el cual gestiona y responde.

Entre las características principales figuran [18]:

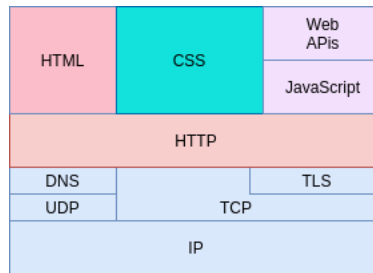


FIGURA 2.4. Pila de comunicaciones en un navegador.

- Es sencillo.
- Puede expandirse.
- Es un protocolo de sesiones pero no de estados: no guarda ningún dato entre dos peticiones en la misma sesión.
- Las conexiones se realizan en la capa de transporte por lo que quedan fuera del protocolo HTTP.

La secuencia de la comunicación entre el cliente y el servidor es la siguiente [18]:

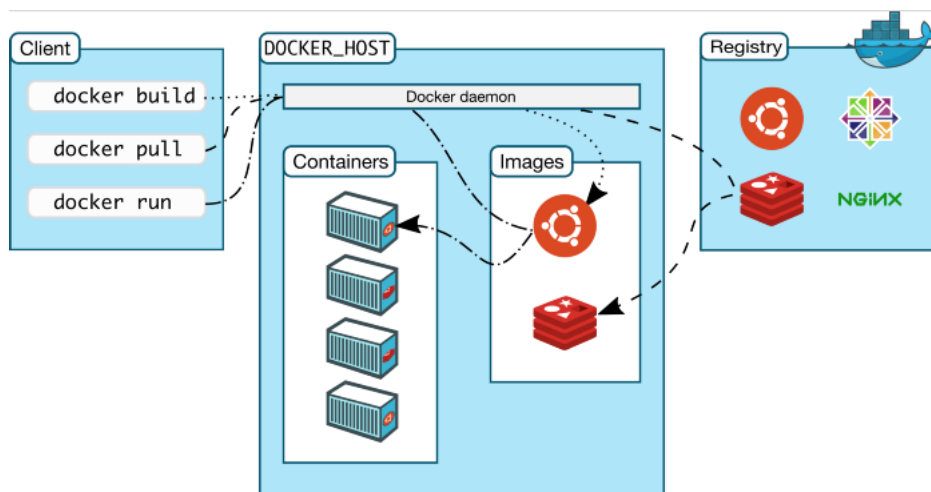
1. Se abre conexión TCP: la conexión TCP se utiliza para hacer una petición, o varias, y recibir la respuesta. El cliente puede abrir una conexión nueva, reusar la existente o abrir varias a la vez hacia el servidor.
2. Hacer una petición HTTP: los mensajes HTTP son legibles en texto plano.
3. Se lee la respuesta enviada por el servidor.
4. Se cierra o reutiliza la conexión.

En el encabezado del mensaje HTTP se puede introducir información de la sesión. Esto resulta muy importante en referencia a la seguridad.

2.5. Sistemas de contenedores Docker

Desde los orígenes de la programación, existió una necesidad de abstraer la aplicación desarrollada de la plataforma en la que se ejecuta. Múltiples avances se fueron dando desde ese entonces: sistemas operativos con interfaces de aplicaciones estándar, librerías portables y lenguajes interpretados en máquinas virtuales son algunos de ellos.

Docker es un proyecto de código abierto que permite automatizar el despliegue de aplicaciones dentro de contenedores de software proporcionando una capa de abstracción y automatización. Fue desarrollado por Docker Inc. y su primera versión fue liberada en 2013. Está escrito en el lenguaje Golang y actualmente es un estándar en la industria del Software [19]. Una estructura general del sistema docker es presentado en la figura 2.5.

FIGURA 2.5. Estructura general de Docker.¹

Un contenedor de software es un componente aislado que se ejecuta como un proceso más dentro del sistema operativo del host y posee todas las dependencias y requerimientos que la aplicación necesita para funcionar correctamente. Por cada contenedor en ejecución hay un proceso ejecutándose en el sistema [19].

Las partes que componen del ecosistema Docker pueden resumirse de la siguiente manera[19][20]:

- *Images*: son plantillas que describen el contenedor. Se componen de una base y sobre estas se modifica según necesidad.
- *Containers*: son instancias de ejecución de una imagen. Se pueden iniciar, detener, mover o eliminar mediante comandos. Se puede conectar un contenedor a una o más redes.
- *Networks*: Docker utiliza una interfaz virtual para comunicarse con la red del equipo host.
- *Volumes*: permiten generar espacios de almacenamiento dentro de cada contenedor, el funcionamiento es similar a las carpetas compartidas en las máquinas virtuales.
- *Registry*: un registro almacena imágenes de docker. Dockerhub es un registro público que cualquiera puede utilizar.

Para poder correr un contenedor docker es necesario ejecutar el comando:

CÓDIGO 2.1. Ejecución de contenedor.

```
>docker-run --[parametro1] --[parametro2] ...
nombre_contendor:version
```

2.5.1. Uso de Docker-compose

Docker Compose es una herramienta del ecosistema Docker que sirve para definir aplicaciones multicontenedor. La configuración se guarda en un archivo de texto llamado "docker-compose.yml". Con esta herramienta se facilita el paso de

¹Imagen tomada de [19]

parámetros al contenedor y posibilita generar imágenes complejas utilizando e interconectando distintos contenedores.

2.6. Introducción a las bases de datos

Se define un dato como un elemento o característica que permite hacer la descripción de un objeto [21]. En el contexto de los sistemas de Internet de las Cosas, es la mínima expresión de información que se puede gestionar.

Una base de datos es una colección de datos organizados, estructurados y almacenados electrónicamente en un sistema de computadoras. En su mayoría son controladas por un DBMS (*database management system*, sistema gestor de bases de datos). Los datos, el DBMS y la aplicación que está asociada a ellos, conforman el llamado sistema de base de datos, o base de datos en forma abreviada [22].

Durante los años 80 se popularizó el tipo de base de datos relacional, cuya información se organizaba como un conjunto de tablas con columnas y filas. Este tipo de base de datos provee la forma más eficiente y flexible de acceder a información estructurada.

En 1970, IBM junto a Oracle desarrollaron un lenguaje de programación llamado SQL (*Structured Query Language*, Lenguaje de Colas Estructurado). Su principal objetivo era manipular, encolar, definir datos y proveer control de acceso en las bases de datos relacionales. Este lenguaje es ampliamente utilizado hoy en día, aún cuando surgen nuevos.

Se podría empezar a clasificar las bases de datos en relacionales y no relacionales. Las primeras utilizan SQL como su lenguaje de programación y entre ellas podemos encontrar: PostgreSQL, MySQL y SQL Server. Las segundas no utilizan un lenguaje SQL, ya que no trabajan en base a estructuras definidas, poseen una gran escalabilidad y están diseñadas para manejar grandes volúmenes de datos [21].

2.6.1. Descripción de MySQL

En este trabajo se seleccionó el DBMS MySQL. Es un sistema de gestión de bases de datos con más de 10 millones de instalaciones. Fue desarrollado a mediados de la década de los 90, y es de uso libre. Es potente, rápido y altamente escalable [23].

Existen tres formas de interactuar con una base de datos MySQL: utilizando la consola, a través de una interfaz web como phpMyAdmin y mediante un lenguaje de programación.

En la figura 2.6 se puede observar la interfaz:

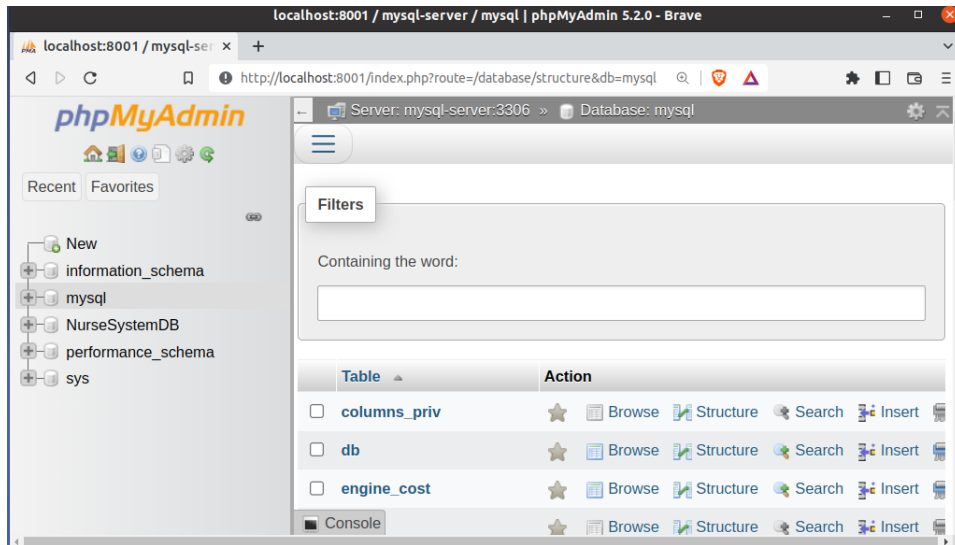


FIGURA 2.6. Página de gestión phpMyAdmin.

Los comandos más utilizados en SQL se observan en la tabla 2.1 [23]:

TABLA 2.1. Comandos básicos SQL.

Comando	Función
ALTER	Modificar una base de datos o una tabla
BACKUP	Copia de seguridad de una tabla
\c	Cancelar la entrada
CREATE	Crear la base de datos
DELETE	Borrar una fila de una tabla
DESCRIBE	Describir columnas de una tabla
DROP	Eliminar una base de datos o una tabla
EXIT	Salir.
GRANT	Cambiar permisos de un usuario
INSERT INTO..VALUES	Insertar datos
RENAME	Cambiar nombre de una tabla
USE	Usar una base de datos
UPDATE	Actualizar un registro existente
SELECT	Consultar una fila
JOIN... ON	Consultar múltiples tablas

Ejemplo de uso, suponiendo 2 tablas: .

TABLA 2.2. TABLA1 para el ejemplo 1.

PacienteId	Nombre
1	Pedro
2	Juan

TABLA 2.3. TABLA2 para el ejemplo 1.

PacienteId	Apellido
1	Perez
2	Salvador

Si se quiere obtener el nombre y apellido del paciente con pacienteId igual a 1, una forma de obtenerlo es ejecutando el código:

CÓDIGO 2.2. Ejecución de comando SQL.

```
>SELECT Nombre,Apellido FROM TABLA1 as TB1 \
JOIN TABLA2 as TB2 ON TB1.pacienteId=TB2.pacienteId \
WHERE TB1.pacienteId=1;
>Pedro Perez
```

Una de las múltiples ventajas que poseen los motores de bases de datos relacionales es la posibilidad de realizar transacciones. Las mismas consisten en secuencias de operaciones que se ejecutan en orden y se completan con éxito.

CÓDIGO 2.3. Secuencia de transacción SQL.

```
>BEGIN;
>SELECT Nombre FROM TABLA1;
>SELECT Apellido FROM TABLA2;
>COMMIT;
>Pedro
>Juan
>Perez
>Salvador
```

2.7. Frameworks/librerías para desarrollo web/móvil

En esta sección se presentan las librerías y frameworks que se utilizaron para el desarrollo.

2.7.1. Librerías Node.js, Express y JWT

El trabajo realizado utiliza como lenguaje de programación Javascript para las tareas del backend. Para generar el servidor se utiliza Node.js [24], que es un *runtime environment* (entorno de ejecución) de Javascript. Node.js se encuentra orientado a eventos asíncronos y está diseñado para crear aplicaciones de red escalables.

Además de la alta velocidad de ejecución, Node.js posee un bucle de eventos (*Event loop*), que permite gestionar múltiples clientes de forma asíncrona. La principal ventaja que posee el método de bucle de eventos de Node.js con respecto al método tradicional (que se valían de programación basada en hilos) es que no escala la cantidad de memoria a utilizar a medida que se aumentan las conexiones.

Por último, es importante mencionar que Node.js posee un gestor de paquetes/librerías llamado NPM, que permite fácilmente incorporar piezas de software probadas a los proyectos.

Para el ruteo de los endpoints con los que se comunica la página web se utiliza Express.js [25] que es una infraestructura web rápida, minimalista y flexible para Node.js.

Express.js facilita la creación de la API (*Application Program Interface*, Interfaz de Programación de Aplicaciones), así como la incorporación de software *Middlewa-re*.

Con respecto a la seguridad para el acceso de la página web, existen dos métodos: por medio de *cookies* o por medio de web *tokens*. Se seleccionó utilizar el segundo y para ello se utiliza JWT [26] que implementa el estándar RFC 7519.

2.7.2. Framework Ionic/Angular, Capacitor y Android Studio

Drifty.co introdujo Ionic en el año 2013 [27]. Nació originalmente para desarrollar aplicaciones móviles con Angular 1 pero hoy en día permite desarrollar para dispositivos celulares, páginas web progresivas y aplicaciones de escritorio junto con Electron [28]. Ionic permite trabajar con Vue, React y Angular en su última versión.

Este framework posee muchos *plugins* (o software de ayuda) que permiten acceder a los recursos de los dispositivos móviles como ser su micrófono, la cámara, la información de posicionamiento, entre otros. Junto con Ionic se utiliza Capacitor [29], que es un runtime nativo para iOS, Android y aplicaciones web progresivas.

El código que genera Ionic debe compilarse para generar el archivo ejecutable (ya sea .ipa para iOS o .apk para Android). Para ello, se utilizan los entornos de desarrollo Xcode para iOS, y Android Studio para Android.

Por ejemplo, el método de desarrollo para una aplicación que se ejecutará en un dispositivo Android es el siguiente:

1. Desarrollo: se genera el código fuente en una PC.
2. Prueba virtual en el navegador: se realiza debug localmente, utilizando los comandos `ionic serve` o `ionic-lab`
3. Prueba en el dispositivo: se genera un *build* (construcción de software) y se carga Android Studio. Luego se descarga en el dispositivo.

Capítulo 3

Diseño e implementación

En este capítulo se explica la forma en que se utilizaron las herramientas mencionadas en el capítulo 2 para resolver los objetivos del trabajo presentados en el capítulo 1.

3.1. Generación del entorno base para el desarrollo del sistema de backend

En esta sección se presentan las distintas partes que componen el sistema. La solución fue concebida como un sistema distribuido, por lo que la página web de configuración, el backend, el *broker* y las aplicaciones móviles pueden estar en distintos dispositivos físicamente. La única condición necesaria es que se encuentren en una red que permite la conexión entre ellos. La red puede ser una red local (implementada con router) o bien internet. La aplicación web utiliza HTTP para interactuar con el backend y las aplicaciones móviles utilizan MQTT.

3.2. Broker Mosquitto

El broker Mosquitto es una parte central del sistema ya que se encarga de distribuir los mensajes entre los distintos elementos. El único requisito es que se encuentre instalado en la misma red que los clientes (en un dispositivo que no requiere de muchos recursos). Su instalación, en un entorno operativo Ubuntu se realiza con los siguientes comandos:

```
1 sudo apt-get install mosquitto mosquitto-clients
2 sudo systemctl enable mosquitto.service
```

CÓDIGO 3.1. Instalación/lanzamiento del broker Mosquitto.

Para configurarlo se debe editar el archivo: `/etc/mosquitto/mosquitto.conf`", donde se especifica el puerto que se utiliza, en este caso, el puerto 9001 con el protocolo Websockets. Se permite la interconexión de cualquier cliente y el archivo donde se guarda el log de todos los eventos se encuentra en la ubicación `/var/log/mosquitto/mosquitto.log`".

```
1 log_dest file /var/log/mosquitto/mosquitto.log
2
3 include_dir /etc/mosquitto/conf.d
4 listener 9001
5 protocol websockets
6 allow_anonymous true
```

CÓDIGO 3.2. Contenido archivo mosquitto.conf.

3.3. Sistema Docker

Para iniciar el contenedor *Docker* se ejecuta el comando: "*Docker-compose up*". Este comando busca en el archivo *docker-compose.yml* la información de los distintos servicios que debe inicializar y su configuración.

Los siguientes servicios son instalados:

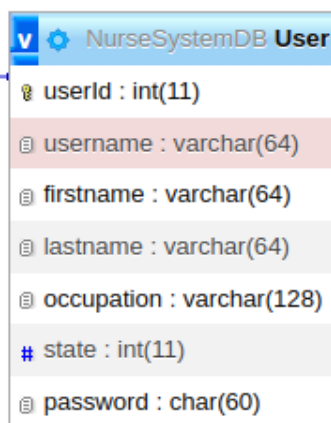
- servidor mysql: se utiliza la imagen: 5.7. Utiliza el puerto 3036 para las comunicaciones.
- servidor mysql-admin: se utiliza la imagen: phpmyadmin/phpmyadmin. Se configura el puerto 8001:80 para servir la aplicación.
- backend node: se utiliza la imagen abassi/nodejs-server:10.0-dev.

Por otra parte, se configura una red interna *NurseSystem-net* que hace de *bridge* (puente) con la red del sistema host.

3.4. Base de datos del sistema

En esta sección se describen las distintas tablas que se almacenan en la base de datos y son útiles para el sistema.

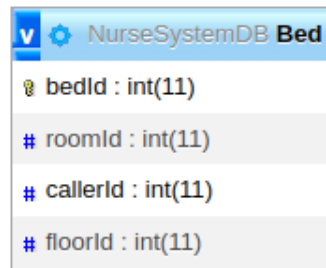
1. Tabla usuarios (*User*): tabla que contiene la información de los usuarios del sistema. El campo *id* se incrementa automáticamente, es decir, al incorporar un nuevo usuario al sistema la base de datos asigna el *id* (se utiliza este campo como llave primaria). En el campo contraseña se almacena el hash de la misma (generado con Bcrypt [30]). El campo ocupación representa la actividad a la que se dedica el usuario en el establecimiento y puede tomar uno de los siguientes valores: "Enfermero", "Médico" o "Administrador". El campo estado (*State*) se utiliza en versiones posteriores del sistema. La estructura de la tabla se presenta en la figura 3.1.



NurseSystemDB User	
🔑	userId : int(11)
📄	username : varchar(64)
📄	firstname : varchar(64)
📄	lastname : varchar(64)
📄	occupation : varchar(128)
#	state : int(11)
📄	password : char(60)

FIGURA 3.1. Tabla Usuarios.

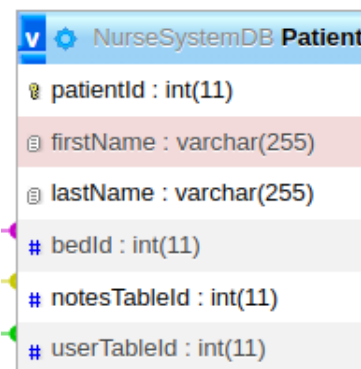
2. Tabla Camas (*Bed*): tabla que contiene la siguiente información de las camas: su identificación (como en el ítem anterior se incrementa automáticamente y es clave primaria), ubicación (piso y cuarto) y el número de dispositivo llamador. La estructura de la tabla se presenta en la figura 3.2.



NurseSystemDB Bed	
🔑	bedId : int(11)
#	roomId : int(11)
#	callerId : int(11)
#	floorId : int(11)

FIGURA 3.2. Tabla Camas.

3. Tabla Paciente (*Patient*): tabla que contiene la información de los pacientes. En esta tabla el id no se incrementa automáticamente (ya que es asignado manualmente por el administrador). Además, los campos *bedId*, *notesTableId* y *userTableId* contienen claves foráneas que identifican un elemento de la tabla *Bed* (cama), *notesTable* (tablas de notas) y *userTable* (tablas de médicos relacionados al paciente). La estructura de la tabla se presenta en la figura 3.3.



NurseSystemDB Patient	
🔑	patientId : int(11)
📄	firstName : varchar(255)
📄	lastName : varchar(255)
#	bedId : int(11)
#	notesTableId : int(11)
#	userTableId : int(11)

FIGURA 3.3. Tabla pacientes.

4. Relación médicos-pacientes: para generar la relación se utilizan un par de tablas intermedias que permiten que un mismo paciente posea varios médicos asignados. Además, un paciente puede utilizar a los mismos médicos de otro paciente. Se presenta la relación en la figura 3.4.
5. Relación tabla de notas-pacientes: como en el caso anterior, se utiliza una tabla de notas generales con una tabla intermedia que los indexa. En este caso, y como es particular de cada paciente, no se puede repetir. Se presenta la relación en la figura 3.5.

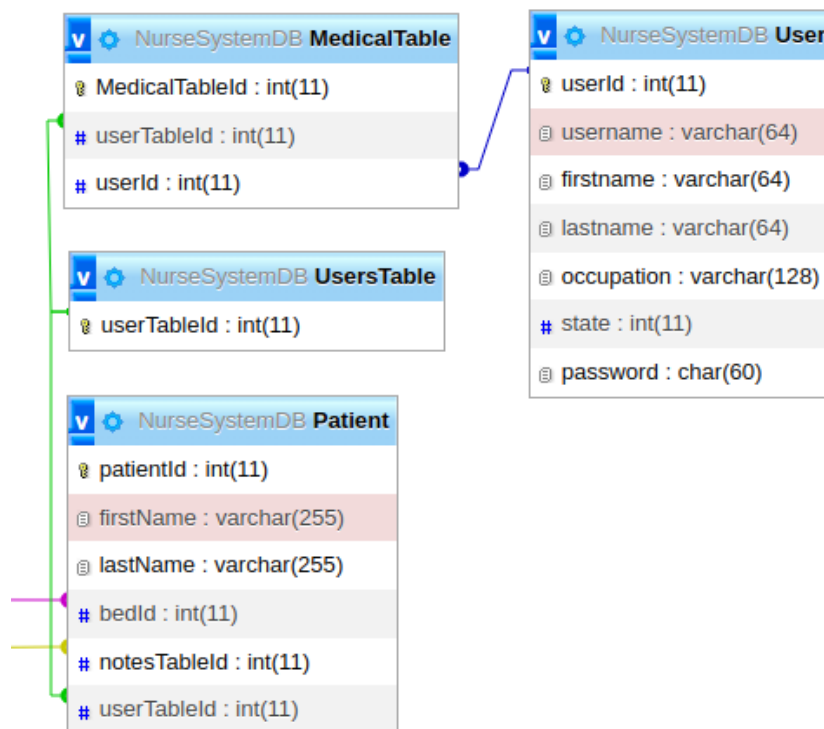


FIGURA 3.4. Relación médicos-pacientes.

6. Tabla de eventos programados: En esta tabla se cargan las tareas programadas para un paciente. Se representa en la figura 3.6 y sus campos son los siguientes:
 - eventId: identificador de evento, se auto incrementa.
 - patientId: número de paciente, es asignado por el cliente y hace referencia a la identificación del paciente
 - type: tipo de evento, puede ser *diario*, *mensual* o *anual*
 - Datetime: momento que se requiere que se realice la acción.
 - Nota: es donde se coloca la tarea a realizar (Por ejemplo, suministrar X gramos de medicamento Y).
7. Tabla de log eventos: En esta tabla se guardan los eventos relacionados con un paciente. Su estructura se muestra en 3.7 y sus componentes son:
 - logEventId: se asigna al guardarse el evento.
 - type: el tipo puede ser *tarea programada* o *llamada paciente*.
 - patientId: identificador del paciente.
 - userId: identificador del usuario que finalizó la tarea.
 - Nota: en el caso de tarea programada, se guarda la nota de la tarea.
 - Nota2: se cargan por el usuario que finalizó la acción al escribir la memoria de lo realizado.

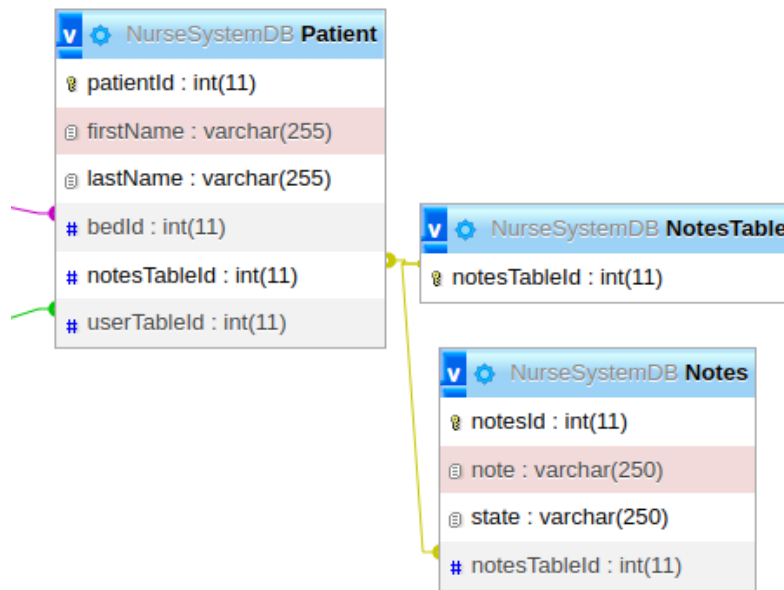


FIGURA 3.5. Relación notas-pacientes.

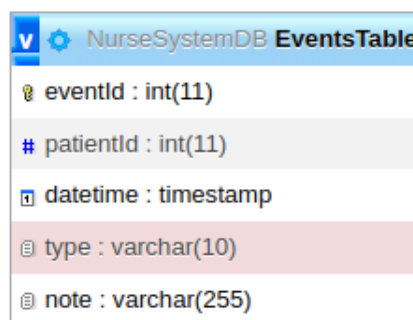
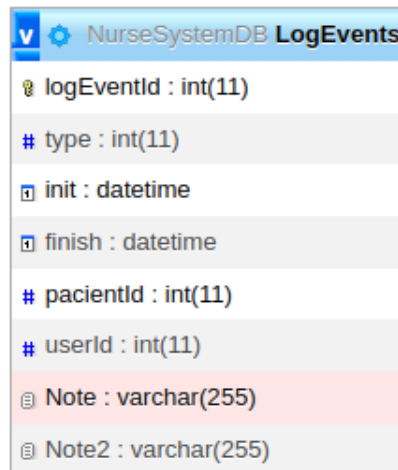


FIGURA 3.6. Tabla de eventos programados.

8. Tabla de especialidades (*SpecTable*): En esta tabla se presentan las distintas especialidades que poseen los enfermeros. Cada especialidad es un tratamiento que puede ser asignado a un paciente.
9. Tabla de especialidades de enfermeros (*NurseSpecTable*): En esta tabla se relacionan a los enfermeros con su Id y las distintas especialidades que pueden realizar. Un enfermero puede poseer más de una especialidad.
10. Tabla de tratamientos de pacientes (*PatientSpecTable*): En esta tabla se relacionan a los pacientes con su Id y un tratamiento (se obtiene de la *SpecTable*).
11. Tabla de códigos QR (*QRbed*): En esta tabla se almacenan, en formato texto, los códigos QR correspondientes a las camas para su reconocimiento.
12. Tabla de prioridades (*PriorityTable*): En esta tabla el administrador puede asignar prioridades a las distintas camas del sistema. Las prioridades son de 5 niveles, siendo 5 la más alta prioridad.



logEventId : int(11)
type : int(11)
init : datetime
finish : datetime
pacientId : int(11)
userId : int(11)
Note : varchar(255)
Note2 : varchar(255)

FIGURA 3.7. Tabla de registro de eventos.

3.5. Sistema de gestión

El backend se diseñó fragmentándolo en tres partes:

- Módulo de monitoreo: clases que ayudan a publicar a todos los clientes MQTT los estados del sistema (usuario logeados y estados de habitaciones/pacientes).
- Módulo de respuestas al cliente página web: expone una API para que los administradores del sistema puedan incorporar usuarios, pacientes, camas, observar estadísticas, etc.
- Módulo de respuestas al cliente MQTT: se suscribe a los tópicos correspondientes para responder consultas.

En la figura 3.8 se presenta la estructura de directorios del backend.

La descripción de los contenidos de cada carpeta es:

- middleware: contiene el programa que filtra el acceso a la API desde la página web.
- Monitoring: contiene clases con información del estado del sistema que se actualiza cada cierto tiempo. Cada una de estas clases contiene una lista de todos los elementos y se presenta a los tópicos correspondientes.
- mqtt: contiene clases que se encargan de procesar y responder los mensajes que se reciben por medio del protocolo MQTT.
- mysql: contiene la clase con el *pool* ("grupo") de conexiones a la base de datos. El objetivo es poder reutilizar conexiones por distintos usuarios.
- routes: contiene las rutas de express para acceder a la base de datos desde las peticiones por HTTP.

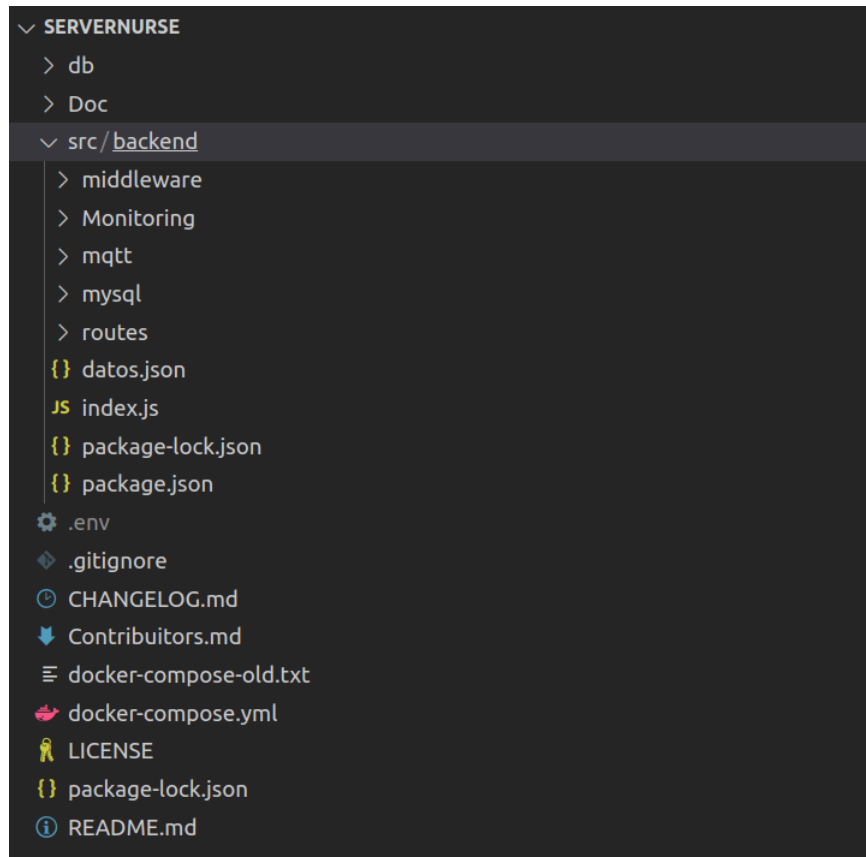


FIGURA 3.8. Estructura de carpetas (backend).

3.5.1. Descripción de las clases para monitoreo del sistema

Las siguientes son las clases que se utilizan para reportar mediante MQTT los estados del sistema. Para resumir se presentan los elementos principales, pero no las funciones que las componen.

- *Monitoreo de camas* (código 3.3):
La clase *Bedslist* contiene una lista de elementos JSON con información de cada cama (se instancia al iniciar el backend).

```

1 class BedsList {
2     constructor() {
3         this.bedlist=[{id:0,st:0,spec:0}];
4     }
5 ...}

```

CÓDIGO 3.3. Clase Bedlist.

Los componentes de cada elemento son

1. id: identificador de cama.
2. st: estado de la cama. Los valores que puede poseer este campo son:
 - 0: no ocupada.
 - 1: ocupada.
 - 2: llamando paciente.

- 3: llamada aceptada por enfermero.
- 4: enfermero atendiendo.
- 5: tarea programada.
- 6: enfermero solicitando ayuda.

3. spec: número de tratamiento del paciente en la cama (es utilizado por los clientes enfermeros para filtrar si pueden o no atenderlo).

■ *Monitoreo de usuarios* (código 3.4):

La clase *UserList* contiene una lista de elementos JSON con información de cada usuario (se instancia al iniciar el backend):

```

1 class UserList {
2   constructor() {
3     this.UserList=[{id:0,st:0}];
4
5   }
6   ...}

```

CÓDIGO 3.4. Clase Userlist.

Los componentes de cada elemento son

1. id: identificador de usuario.
2. st: estado de usuario. Puede poseer los siguientes estados:
 - 0: no logeado.
 - 1: logeado.

■ *Monitoreo de eventos programados* (código 3.5):

La clase se utiliza para presentar a los clientes las notas de las tareas programadas que se lanzan. Es una lista que se vacía cuando la tarea finaliza (sirve para mantener ordenadas las tareas programadas).

```

1 class Calendarlist {
2   constructor() {
3     this.CalendarList=[{calendarId:0,bedId:0,note:null}];
4
5   }
6   ...}

```

CÓDIGO 3.5. Clase CalendarList.

Los componentes de cada elemento son

1. calendarId: identificador de evento.
2. bedId: número de cama.
3. note: Nota de la tarea (obtenida de la base de datos).

■ *Monitoreo de paciente-usuario-tipo de evento* (código 3.6):

La clase BedUserlist se utiliza para saber quien está atendiendo, en todo momento, a un paciente.

```

1 class BedUserlist {
2   constructor() {
3     this.beduserlist=[{bedId:0 ,userId:0 ,type:0}];
4   }
5   ...}

```

CÓDIGO 3.6. Clase BedsUserList.

Los componentes de cada elemento son

1. bedId: número de cama.
2. userId: número de usuario.
3. type: tipo de evento:
 - provocado por un paciente
 - visita programada

3.5.2. Descripción de API MQTT para la mensajería de la aplicación móvil

En esta sección se describe los módulos que permiten la interacción del sistema con los clientes utilizando el protocolo MQTT.

El módulo principal es mqtt.js (dentro de la carpeta mqtt) y contiene la inicialización de la conexión inicial al broker, el lanzamiento de las tareas de monitoreo (publicación de las listas antes definidas en los tópicos específicos) y la escucha de los mensajes. Una vez recibido un mensaje, dependiendo del tipo, se deriva a la clase correspondiente que realiza el tratamiento acorde a la petición. En el fragmento de código 3.7 se presentan las tareas de inicialización del módulo.

```

1 var client = mqtt.connect(process.env.MQTT_CONNECTION)
2 //listening to messages
3 client.on('connect', function () {
4   client.subscribe('/User/#', function (err) {
5     if (err) {
6       console.log("error:"+err);
7     }
8   })
9
10  client.subscribe('/Pacient/#', function (err) {
11    if (err) {
12      console.log("error:"+err);
13    }
14  })
15  client.subscribe('/Beds/#', function (err) {
16    if (err) {
17      console.log("error:"+err);
18    }
19  })
20
21
22
23
24
25

```

```

26
27 //task that will publish beds state each second
28 setInterval(publishBedStates , 10000);
29 //task that will publish users state
30 setInterval(publishUserStates , 15000);
31 })

```

CÓDIGO 3.7. Tareas ejecutadas por mqtt.js.

El sistema recibe mensajería por medio de tres tópicos principales: *User*, *Patient* y *Beds*, según se observa en las líneas 4, 10 y 15 de 3.7. Por otra parte, se utilizan variables de entorno como ser *MQTT_CONNECTION*, que contiene información de la IP del broker y el puerto.

Por último, dentro de esta función se setean dos temporizadores para ejecutar las funciones *publishBedStates* y *publishUserStates* que publican los listados de los estados de camas y usuarios cada cierto tiempo.

El contenido útil (*payload*) de los mensajes MQTT tiene la siguiente forma:

```

1
2 payload={_username: "xxxx", _content: "xxx", _bedId: xx, _type: x}

```

CÓDIGO 3.8. Formato mensaje MQTT.

La descripción de los campos es la siguiente:

- *_username*: nombre del usuario que envió el comando.
- *_content*: contenido.
- *_bedId*: número de cama a la que se hace referencia.
- *_type*: tipo de mensaje.

El tipo de mensaje permite al sistema derivar las tareas a realizar a las clases correspondientes. En la tabla 3.1 se presentan la asignación numérica de cada tipo de mensaje:

Las distintas clases que se utilizan son:

1. *beds*: consulta a la base de datos información relacionada a las camas.
2. *patient*: consulta a la base de datos información relacionada a los pacientes.
3. *users*: consulta a la base de datos información relacionada a los usuarios (funciones de logueo y deslogueo).
4. *calendar*: consulta a la base de datos información relacionada a los eventos programados.
5. *nurse*: consulta a la base de datos información sobre las especialidades de los enfermeros.

Todas estas clases responden en un tópico correspondiente a los clientes que consultan.

Las excepciones al formato general de los mensajes vienen dadas por:

- Mensaje de último testamento: informa que un usuario se desconectó. En su *payload* contiene información del nombre de usuario y se publica en el tópico *"/User/Disconnect"*.

TABLA 3.1. Tipos de mensajes del sistema.

Tipo	Descripción
1	Login.
2	Logout.
3	Escribir nota a paciente.
4	Solicitar información de paciente de una cama.
5	Solicitar notas del paciente.
7	Mensaje de texto entre usuarios.
8	Solicitar información de cama (ubicación).
9	Solicitando información de camas de cada médico.
10	Solicitar información de un paciente de una cama.
11	Chequeo de QR.
12	Aceptación de parte de una enfermera.
13	Finalización de trabajo.
14	Solicitar ayuda.
16	Especialización de enfermera.
17	Consultar tabla de médicos asignada a paciente.
18	Eliminar nota de paciente.
19	Cancelar visita.
20	Notas de evento calendario.
22	Mensaje de audio entre usuarios.

- Mensaje de llamador: informa que un dispositivo llamador fue accionado por un paciente. En su *payload* contiene información del número de llamador y se publica en el tópico `"/Beds/Caller - events"`.

3.5.3. Descripción de API Rest para aplicación Web

La API desarrollada utiliza Express junto con Cookie-parser.

En este trabajo se utiliza un *middleware* (capa de software intermedia entre los recursos y la consulta de los usuarios), el cual consiste en una función que realiza el control de acceso a los endpoints. Para lograr la autenticación el backend utiliza las librerías *jsonwebtoken* [31] y *bcrypt* [30]. En una primera versión, solo se habilita el uso de la página de administración a usuarios administradores.

Se denomina ruteo a la forma que una aplicación express direcciona a las peticiones del cliente a los respectivos módulos que se encargan de su tratamiento. El objeto express posee métodos que realizan las operaciones sobre la base de datos o el sistema. El fragmento de código con las rutas express se observa en el código 3.9:

```

1 app.use('/api/patient', auth.isAuthenticated, routerPatient);
2 app.use('/api/user', auth.isAuthenticated, routerUser);
3 app.use('/api/messages', auth.isAuthenticated, routerMessages);
4 app.use('/api/notes', auth.isAuthenticated, routerNotes);
5 app.use('/api/beds', auth.isAuthenticated, routerBeds);
6 app.use('/api/usersTable', auth.isAuthenticated, routerUsersTable);
7 app.use('/api/medicalTable', auth.isAuthenticated, routerMedicalTable);
8 app.use('/api/QR', auth.isAuthenticated, routerQR);
9 app.use('/api/events', auth.isAuthenticated, routerEvents);
10 app.use('/api/logEvents', auth.isAuthenticated, routerLogEvents);
11 app.use('/api/Statistics', auth.isAuthenticated, routerStatistics);

```

```
12 app.use('/api/authentication',routerAuthenticate);
13 app.use('/api/specTable',auth.isAuthenticated,routerSpecTable);
14 app.use('/api/nurseSpecTable',auth.isAuthenticated,
    routerNurseSpecTable);
15 app.use('/api/treatment',auth.isAuthenticated,routerPatientSpecTable);
```

CÓDIGO 3.9. Rutas express.

Brevemente, a continuación, se presenta la funcionalidad de cada endpoint:

- /api/patient: endpoint que permite la gestión de pacientes (alta, baja, modificación)
- /api/user: endpoint que permite la gestión de usuarios (alta, baja, modificación)
- /api/messages: permite obtener un listado de los mensajes entre usuarios (en esta versión de código y debido las limitaciones de almacenamiento del servidor prototipo, se encuentra comentada toda esta funcionalidad)
- /api/notes: endpoint que permite la gestión de las notas de los pacientes (alta, baja, modificación).
- /api/beds: endpoint que permite la gestión de las camas (alta, baja y modificación).
- /api/usersTable: endpoint que permite la gestión de las tablas de usuarios. Son las agrupaciones de los médicos que pueden atender a un paciente.
- /api/medicalTable: endpoint que permite la gestión de las tablas de tablas de usuarios.
- /api/QR: endpoint que permite la gestión de los códigos QR (alta, baja y modificación).
- /api/events: endpoint que permite ingresar una tarea programada al calendario.
- /api/logEvents: endpoint que permite obtener el listado de eventos.
- /api/Statistics: endpoint que permite obtener información estadística de la base de datos. Por ejemplo:
 1. número total de visitas de cada enfermero, en formato JSON.
 2. número total de llamados de cada paciente, en formato JSON.
 3. número de pacientes con igual tratamiento, en formato JSON.
 4. número de enfermeras con una especialización, en formato JSON.

Un ejemplo de uso de esta información puede ser: al observarse un incremento de pacientes con cierto tratamiento se puede solicitar capacitar a nuevas enfermeras en esa especialización. Por el contrario, si disminuye un tratamiento se capacita al personal o bien se lo reduce.

- /api/authentication: endpoint que permite loguearse al sistema. Se explica con más detalle por su importancia.

- /api/specTable: endpoint que permite modificar la tabla de especialidades de enfermeros (tratamientos para los pacientes). Se permite agregar o quitar especialidades.
- /api/nurseSpecTable: endpoint que permite asignar o quitar especialidades a una enfermera. Por ejemplo, una enfermera recibió una capacitación y desde ese momento queda habilitada para atender pacientes con una cierta necesidad.
- /api/treatment: endpoint que permite asignar a un paciente un tratamiento o bien modificar el tratamiento.

La ruta authentication se utiliza para entregar un *token* al cliente. En dicha función se verifica que el usuario sea administrador.

```

1 routerAuthenticate.post('/', async function(req, res) {
2   if (req.body) {
3     var user = req.body;
4     await pool.query('Select username,password,occupation from User
WHERE username=?',[user.username], async function(err, result,
fields) {
5       if (err) {
6         var response = JSON.stringify(response_conform);
7         res.status(403).send({
8           errorMessage: 'Auth required!'});
9         return;
10      }
11      else{
12        try{
13          testUser.username=result[0].username;
14          testUser.password=result[0].password;
15        } catch (e){res.status(403).send({
16          errorMessage: 'Auth required!'});
17          return;
18        }
19        await bcrypt.compare(user.password, result[0].
password, (err, resultComp)=> {
20
21          if ((resultComp==true ) &&(result[0].occupation
=="Administrador") ){
22            var token = jwt.sign(user, process.env.
JWT_SECRET,{
23              expiresIn: process.env.JWT_EXP_TIM
24            });
25            res.status(200).send({
26              signed_user: result[0],
27              token: token
28            });
29          } else {res.status(403).send({
30            errorMessage: 'Auth required!'
31          }); }
32        })
33      }
34    })
35  } else {
36    res.status(403).send({
37      errorMessage: 'Please provide username and password'
38    });
39  }
40
41 })

```

CÓDIGO 3.10. Logueo web.

En el código 3.10 se presenta la función que se encarga de autenticar al usuario. Al recibir un mensaje de autenticación, el backend genera la siguiente secuencia de comprobación:

1. Verificación que el usuario existe: línea 4
2. Verificación que el password recibido corresponde con el usuario: línea 21
3. Verificación que el usuario es administrador: línea 21

Luego de eso genera un token que expira luego de un cierto tiempo dado por una variable de entorno y se responde al frontend.

Una vez logueado, la función *auth.isAuthenticated(request,response,next)*, dentro del archivo *./middleware/authentication* comprueba que el usuario que solicita el recurso posea el token correspondiente.

```
1 exports.isAuthenticated = async(req, res, next)=> {
2   let authHeader = (req.headers.authorization || '');
3   if (authHeader.startsWith("Bearer ")) {
4     token = authHeader.substring(7, authHeader.length);
5   } else {
6     return res.send({ message: 'No Auth' });
7   }
8   if (token) {
9     jwt.verify(token, process.env.JWT_SECRET, function(err) {
10      if (err) {
11        console.log("Alguien cambio el token, no es valido");
12        return res.json({ message: 'Invalid Token' });
13      } else {
14        console.log("Validado el token y todo ok");
15        return next();
16      }
17    });
18  } else {
19    return res.send({ message: 'No token' });
20  }
21 }
```

CÓDIGO 3.11. Control de token.

Cada recurso posee su ruta correspondiente. Esta forma de organizar el código permite que se incorporen funcionalidades al backend de forma sencilla.

3.6. Página Web

La página web de administración consiste en un *dashboard* (tablero de control) que permite al usuario administrador gestionar el sistema. Fue implementada utilizando el framework Ionic/Angular, con el lenguaje de programación Typescript y las principales secciones de la visualización se observan en la figura 3.9.

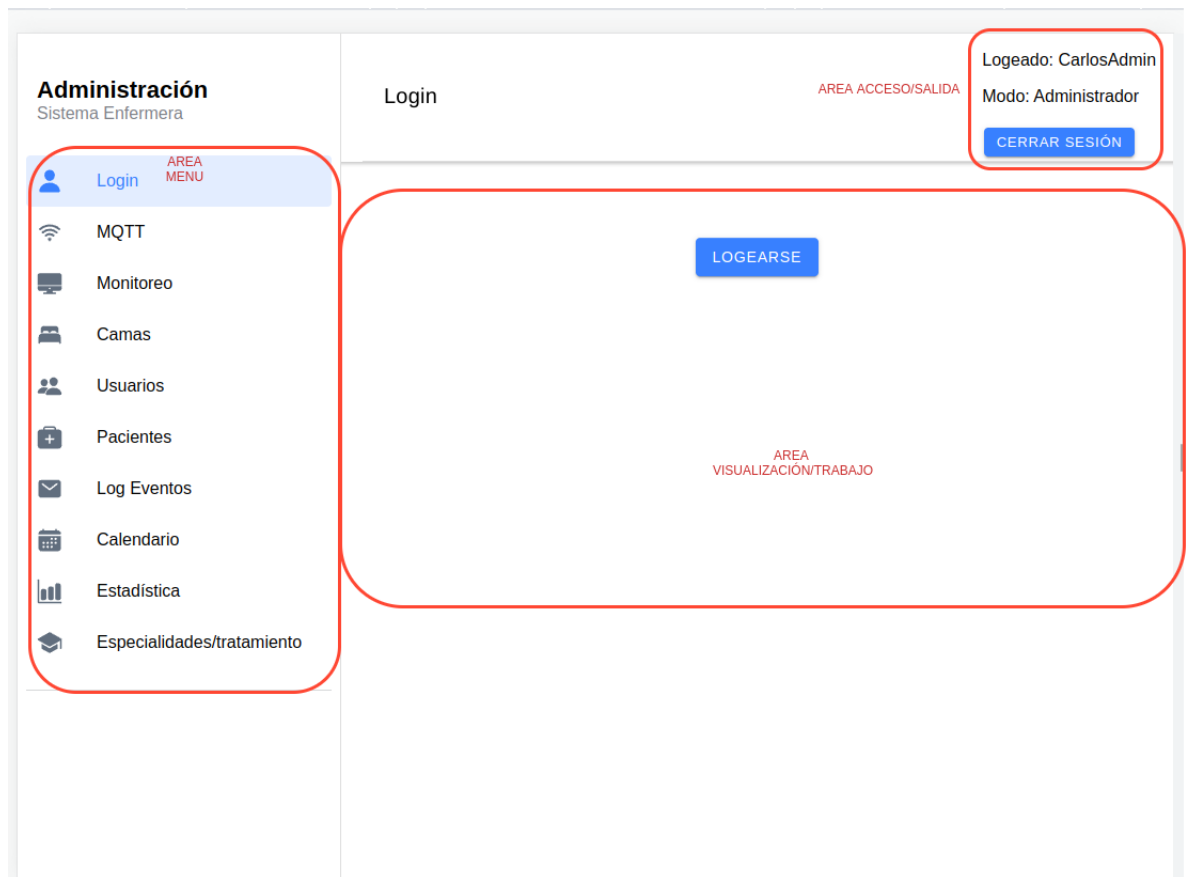


FIGURA 3.9. Página web.

En el área menú se presentan las distintas utilidades de la página:

- Login: acceso al sistema. El visitante ingresa su nombre de usuario y contraseña y se le otorga los permisos correspondientes (la conexión recibe el *token* del backend).
- MQTT: permite editar o probar la conexión al broker MQTT del sistema. Es necesario si se desea monitorear el estado de los usuarios o camas en tiempo real.
- Monitoreo: se visualiza el estado de las camas o de los usuarios en tiempo real.
- Camas: permite editar información de las camas como ser código QR, número de llamador, cuarto y piso.
- Usuarios: permite agregar, editar o dar de baja usuarios.
- Pacientes: permite agregar, editar o dar de baja pacientes.

- Log Eventos: permite observar el listado de eventos que se hayan guardado en la base de datos.
- Calendario: permite observar o agregar tareas rutinarias asignadas a un paciente.
- Estadísticas: permite observar el número de intervenciones de un enfermero, la distribución de especialidades dentro del grupo de enfermeros y la distribución de tratamientos requeridos por los pacientes. Con esta información el administrador puede notificar sobre necesidades de mayor capacitación en un tema para los enfermeros.

3.6.1. Estructura y organización del software

El software web generado contiene dos versiones: una para un entorno de desarrollo y otra para un entorno productivo. En este desarrollo, las características del entorno se encuentran en una carpeta `"/src/environments"`.

Para ejecutar la aplicación en la máquina local se debe ejecutar el comando: `"ionic lab"`, mientras que para compilar la página productiva se debe ejecutar `"ionic build prod"`. El resultado de la construcción se encuentra dentro de la carpeta `"/www"`.

Todo el código de la aplicación se encuentra en la carpeta `"/src/app"` y se presenta una captura en la figura 3.10.

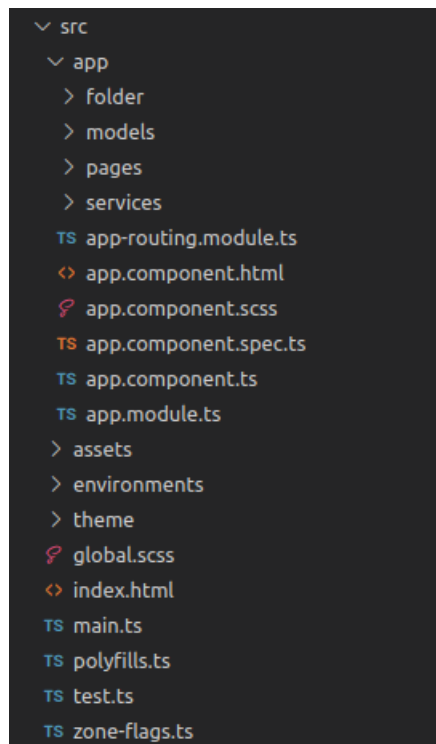


FIGURA 3.10. Estructura de carpetas de la aplicación web.

Dentro de la carpeta `models` se encuentran las distintas clases que se utilizan para la comunicación con el backend. El listado de clases, con una pequeña mención de su utilidad, se presenta a continuación:

- `bed-status`: gestiona información del estado de las camas ("ocupada", "llamando",...) y del tratamiento que utiliza el paciente.
- `bed`: gestiona información de las camas.
- `calendarEvent`: se utiliza para gestionar información de tareas programadas.
- `logEvent`: se utiliza para gestionar información de eventos realizados.
- `medicalTable`: relaciona un usuario con una tabla de médicos.
- `message-model`: modelo de mensaje que se transmite a través de MQTT.
- `note`: modelo de nota para un paciente (no utilizado en la aplicación web).
- `nurseSpec`: clase que se utiliza para observar las especialidades de los distintos enfermeros (número, número de especialidad y nombre de especialidad).
- `patient`: clase que contiene información del paciente (id, nombre, apellido, cama, índice en la tabla de notas, índice en la tabla de médicos)
- `patientTreat`: clase que relaciona un paciente con un tratamiento (índice, número de paciente, número de tratamiento y nombre de tratamiento).
- `qr`: relaciona un índice con un texto (se utiliza para guardar o recuperar índices de qr)
- `spec`: clase que contiene una especialidad o tratamiento. Contiene un índice y un texto con el nombre.
- `user-status`: clase que permite obtener desde el sistema el estado de los usuarios (identificación de usuario y estado "logueado" o "no logeado").
- `user`: contiene información del usuario (nombre, apellido, ocupación, contraseña, nombre de usuario).

Dentro de la carpeta *pages* se encuentran las distintas secciones de acción de la página (se presentan en el área de visualización).

Dentro de la carpeta *services* se organizan los distintos servicios.

Dentro de la carpeta *folder* se encuentra el layout principal de la página.

La aplicación posee un módulo principal llamado "app.module". Utilizando el angular-router, se redirecciona desde cualquier página a una deseada y se presenta dentro del campo de visualización de la pantalla principal.

3.6.2. Acceso de usuario

Un usuario no puede acceder a las distintas utilidades de la página si no se encuentra logeado o si no es administrador. Todas las consultas al backend necesitan poseer un *token* de autenticación en su encabezado. Esto se logra mediante el servicio interceptor:

```

1 export class AuthInterceptorService implements HttpInterceptor {
2
3   constructor(private _router: Router) { }
4
5   intercept(req: HttpRequest<any>, next: HttpHandler): Observable<
      HttpEvent<any>> {

```

```

6     if (req.url.includes("/authenticate")){
7         return next.handle(req);
8     }
9     const token: string = localStorage.getItem('token');
10    let request = req;
11    if (token) {
12        request = this.addToken(request, token);
13        return next.handle(request)
14        .pipe(catchError((error: HttpResponse) => {
15            let errorMsg = '';
16            if (error.error instanceof ErrorEvent) {
17                console.log('Client Error');
18                errorMsg = `Error: ${error.error.message}`;
19            }
20            else {
21                console.log('Server Error');
22                errorMsg = `Error Code: ${error.status}, Message: ${error.
message}`;
23            }
24            console.log(errorMsg);
25            return throwError(errorMsg);
26        })
27    );
28    } else {
29        return next.handle(request)
30        //if i have no token navigate to login
31        this._router.navigate(['/login']);
32    }
33 }
34
35 private addToken(request: HttpRequest<any>, token: string) {
36     return request.clone({
37         setHeaders: {
38             'Authorization': `Bearer ${token}`
39         }
40     });
41 }
42 }

```

CÓDIGO 3.12. Inserción de token.

3.6.3. Monitoreo del sistema

Si se quiere monitorear al sistema, es necesario configurar la ubicación del *broker* dentro de la solapa MQTT. El monitoreo de camas se observa en la figura 3.11. El sistema reporta el estado de las camas cada un segundo (o cuando hay un cambio abrupto) y el estado de los usuarios cada un segundo y quinientos milisegundos.



FIGURA 3.11. Monitoreo de camas.

3.6.4. Gestión de tareas programadas

Si se desea generar un nuevo evento programado, se selecciona del menú calendario, se selecciona el número de paciente y luego se presiona agregar. En ese momento se presenta el formulario de la figura 3.12, se lo completa y se presiona enviar. Los datos a completar son:

- Nota: campo donde se presenta la tarea propiamente dicha.
- tipo: se selecciona si es diario, semanal o mensual.
- en el calendario se selecciona fecha y hora de inicio, desde ese momento se repite según el tipo.

El backend guarda en la base de datos la nueva tarea y genera la acción correspondiente.

← VOLVER

Nota

colocar vacuna brazo derecho

Tipo

Mensual ▾

ENVIAR

November 2022 ▾

< >

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

Time

3:21 PM

FIGURA 3.12. Tareas programadas.

3.6.5. Estadísticas del sistema

En esta sección se puede observar la cantidad de atenciones por enfermero guardadas en la base de datos, la relación entre las distintas especialidades y la relación entre tratamientos y pacientes. En la figura 3.13 se muestra una de las subsecciones que se pueden seleccionar. Para generar las gráficas se utiliza HighCharts [32] (las especialidades son de fantasía, creadas solo para el ejemplo).

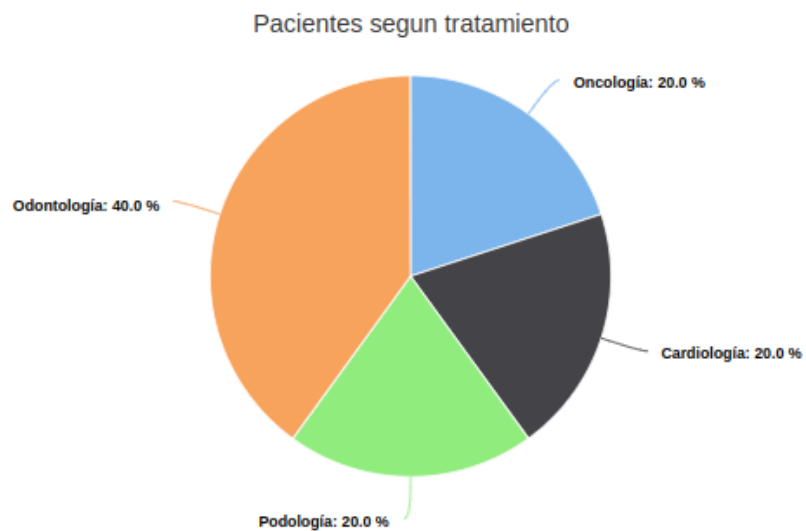


FIGURA 3.13. Relación paciente tratamiento.

3.7. Aplicación Móvil

En esta sección se presenta la funcionalidad de la aplicación móvil y algunos detalles de la implementación.

3.7.1. Estructura y organización del software

La estructura de directorios del proyecto es la presentada en la figura 3.14.

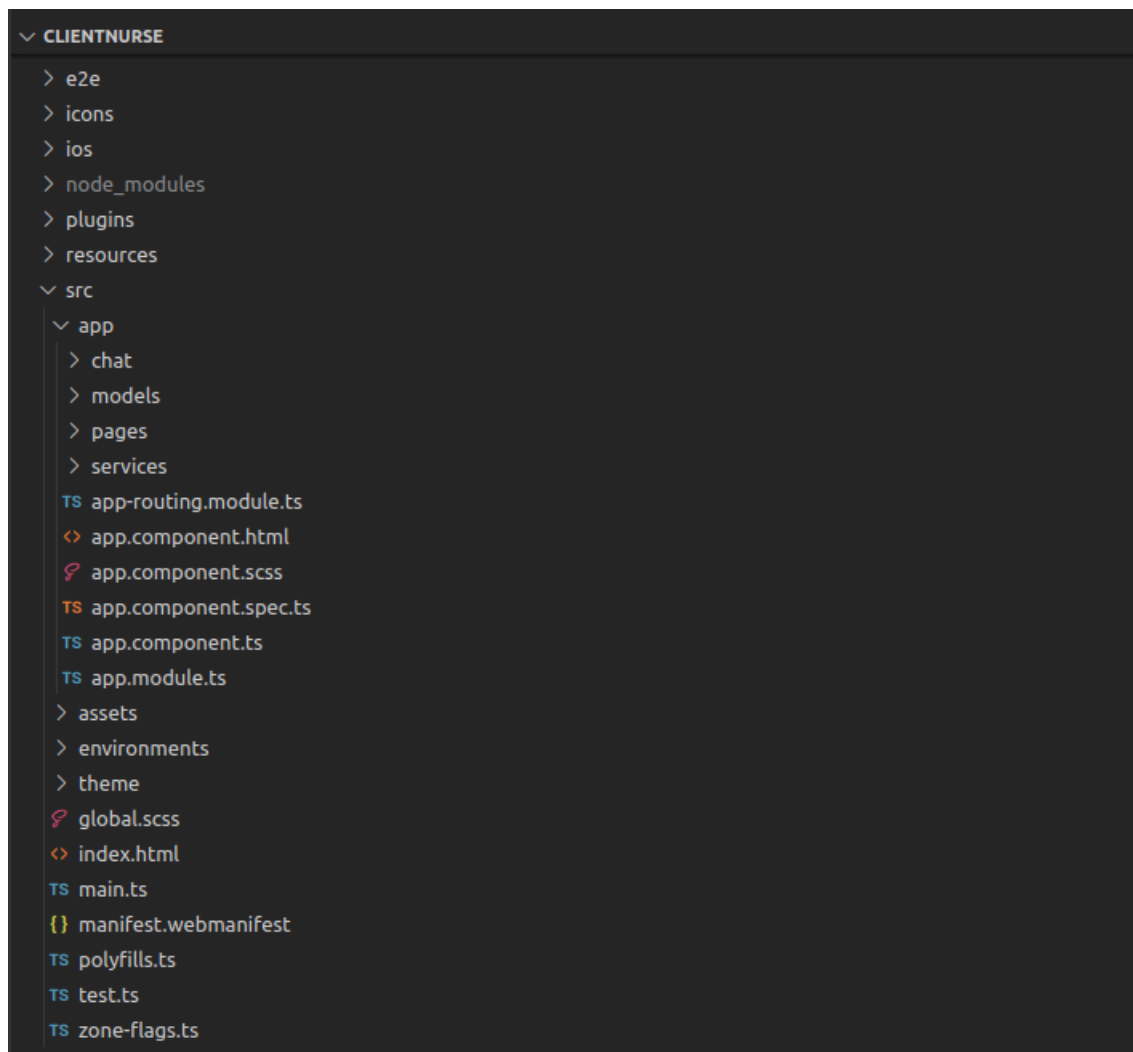


FIGURA 3.14. Estructura de carpetas de la aplicación móvil.

El código de las distintas páginas por las que navega el usuario se encuentra dentro de `./src/app/pages`. Los modelos ("clases") utilizados se encuentran en la carpeta `./src/app/models` y los servicios en la carpeta `./src/app/services`. La carpeta `./src/app/chat` contiene la primera versión de la intercomunicación entre enfermeros-médicos que simulaba una sala de reunión (similar a la ofrecida por el producto en el mercado). Luego se migró a la definitiva, que cumple estrictamente con lo solicitado en un principio, pero por solicitud del cliente, no se eliminó la carpeta.

Como se mencionó en la sección 3.1, la aplicación móvil interactúa con el backend solo utilizando websockets MQTT.

La aplicación posee un módulo principal llamado "app.module". Utilizando el angular-router, se redirecciona desde cualquier página a una deseada. En el código 3.13 se presenta como se redireccionan las páginas en Ionic:

```
1 const routes: Routes = [  
2   {  
3     path: 'home',  
4     loadChildren: () => import('./pages/home/home.module').then( m => m.  
      HomePageModule)  
5   },  
6   {  
7     path: '',  
8     redirectTo: 'home',  
9     pathMatch: 'full'  
10  },  
11  {  
12    path: 'mqtt-config',  
13    loadChildren: () => import('./pages/mqtt-config/mqtt-config.module')  
      .then( m => m.MqttConfigPageModule)  
14  },  
15  {  
16    path: 'login',  
17    loadChildren: () => import('./pages/login/login.module').then( m =>  
      m.LoginPageModule)  
18  },  
19  ...
```

CÓDIGO 3.13. Fragmento de las rutas de la aplicación móvil.

3.7.2. Configuración del broker y acceso de usuario

Cuando se inicia la aplicación se presenta una pantalla con dos pulsadores, uno redirige a la página de configuración del broker MQTT y otro a la página de ingreso (ver figura 3.15a).

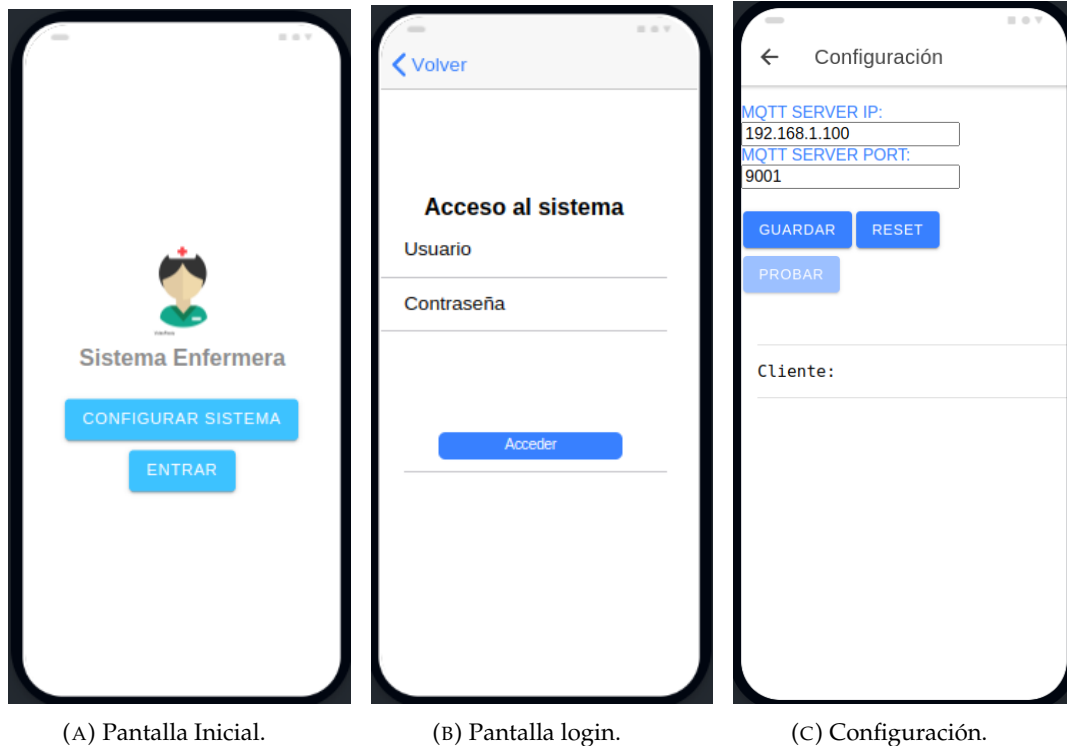


FIGURA 3.15. Inicio de sistema.

En la página de configuración (figura 3.15c) se ingresa la IP del broker del sistema y el puerto. Luego se puede probar la comunicación y guardar los parámetros en el localStorage del dispositivo, como se presenta en el código 3.14.

```

1  /**
2   * Saving port values to localStorage
3   */
4  public saveValues = async () => {
5      this.localSto.saveValuesString('MQTTSERVER', this.MQTTSERVER);
6      this.localSto.saveValuesNumber('MQTTPORT', this.MQTTPORT);
7  };

```

CÓDIGO 3.14. Funciones del servicio que guardan en el localStorage.

En la página de logueo (figura 3.15b) se ingresan el nombre de usuario, y su contraseña.

La aplicación publica la información en el tópico "/User/username" con el formato de mensaje correspondiente y recibe en "/Session/nrodesesión" el número de usuario y el modo de uso. De esta manera la aplicación se setea en el modo correspondiente.

3.7.3. Modo administrador

En el modo administrador (figura 3.16) se puede monitorear el estado de los usuarios y las camas. Para seleccionar lo que se desea monitorear, se presiona el botón correspondiente.

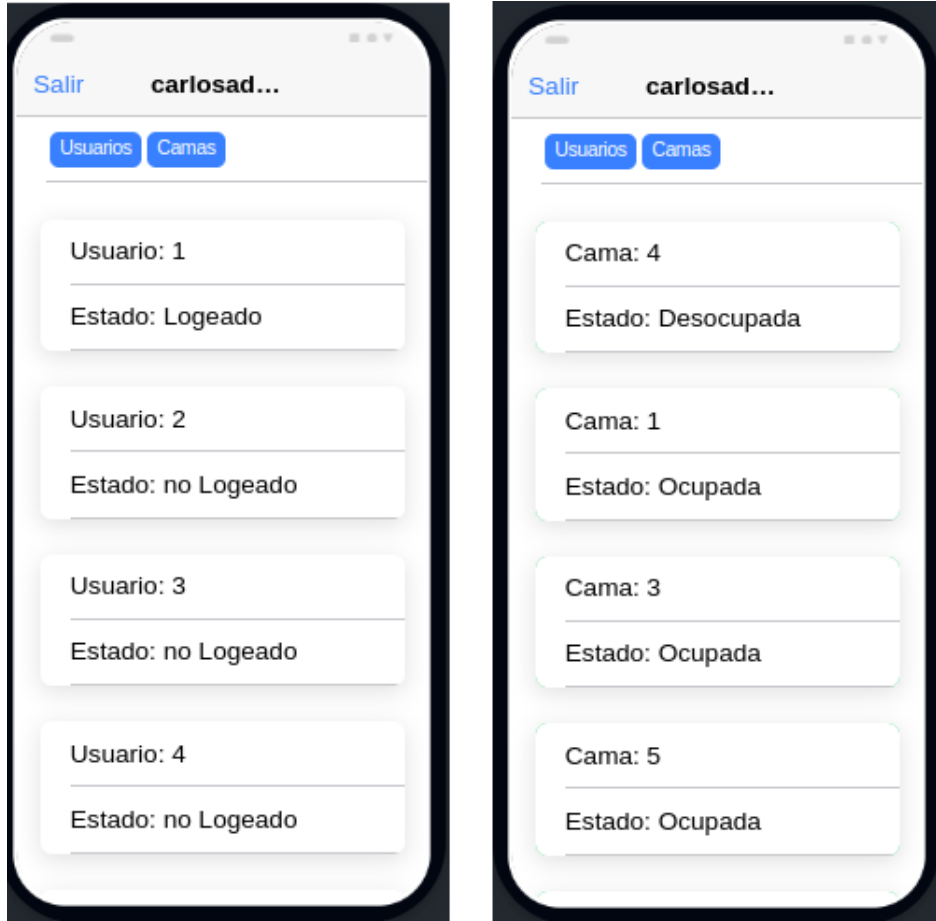


FIGURA 3.16. Pantallas de administración.

El orden en que se presentan las camas tienen que ver con la prioridad que se les asignó a cada una.

3.7.4. Modo médico

Al ingresar en modo médico, la aplicación queda en modo de espera hasta que el usuario decida que hacer. El diagrama de estados se presenta en la figura 3.17 y las distintas pantallas a las que se puede acceder se presentan en la figura 3.18.

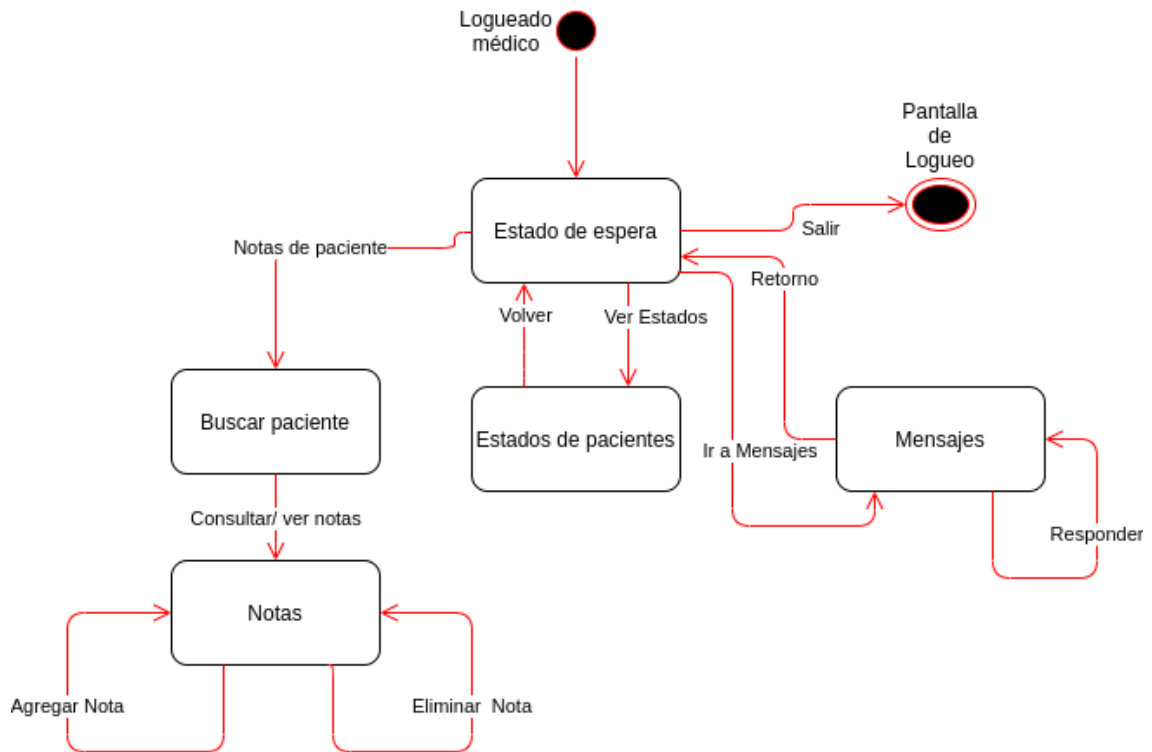


FIGURA 3.17. Diagrama de estados en modo médico.

Las acciones que puede realizar un médico son:

- Actualizar una nota de un paciente: para la gestión de estos mensajes se utiliza el tópico `/Pacient/id` donde `id` es el número de paciente. Cuando el médico solicita información del paciente (nombre, apellido y número) se responde en `/Pacient/id/info`. Cuando se consulta las notas se responde en `/Pacient/id/notes` y cuando se quiere ingresar una nueva nota se publica en `/Pacient/id/newNote`. La pantalla correspondiente se muestra en la figura 3.18b.
- Responder a una consulta de una enfermera. La pantalla correspondiente se muestra en la figura 3.18c
- Monitorear las camas con los pacientes que le fueron encargados: simplemente filtra por sus pacientes el listado publicado en `/Beds/status`. La pantalla correspondiente se muestra en la figura 3.18a

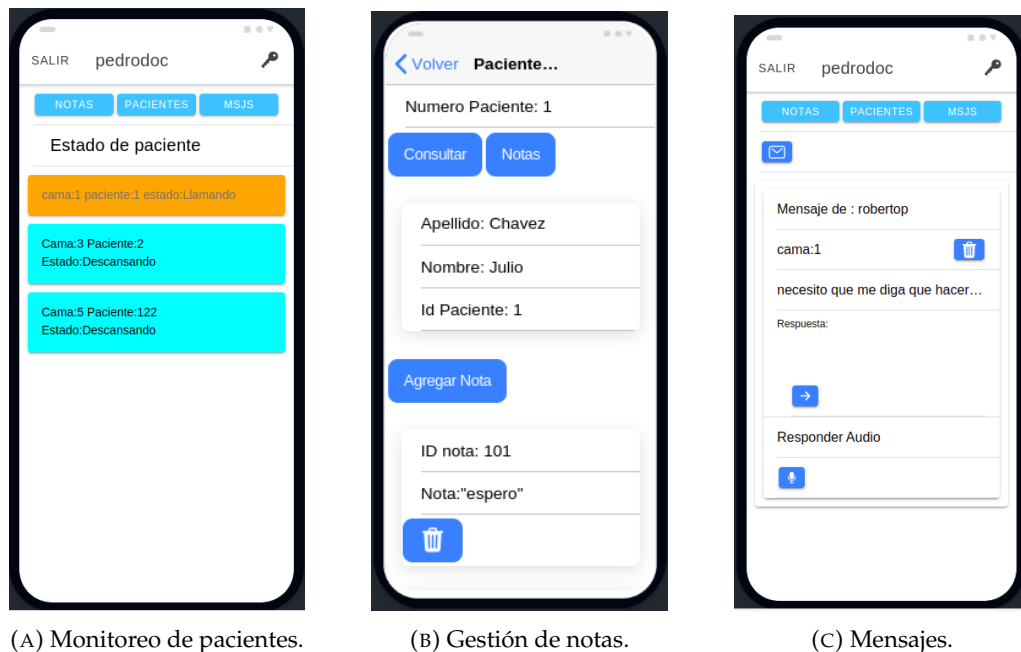


FIGURA 3.18. Modo médico.

3.7.5. Modo enfermera

Es el modo más complejo por la cantidad de opciones que se pueden presentar.

Al ingresar en modo enfermera, la aplicación consulta al backend por la especialización del enfermero correspondiente y la respuesta se obtiene de escuchar en el tópico `"/User/userId/Specs"` donde `userId` se recibió al loguearse.

Para el usuario, la aplicación queda en modo espera hasta que se reciba una notificación de necesidad de atención. El diagrama de estados se presenta en la figura 3.19.

La aplicación se encuentra escuchando al tópico MQTT `"/Beds/status"`, y filtra los estados (solo acepta estados con llamadas, llamadas programadas o ayuda) y por especialidad del enfermero. De esta manera, cuando un enfermero acepta una tarea, automáticamente se actualiza el backend y ningún otro enfermero puede aceptarla.

Cuando arriba la notificación, se presenta una tarjeta con dos botones: información de la cama y aceptación. En caso que la enfermera desee consultar donde se encuentra (piso y habitación) debe presionar el botón información. En caso que decida ir a la habitación, debe presionar aceptar. Al presionar aceptar automáticamente el estado de la cama cambia a desplazándose a la habitación (la información proviene del sistema que recibió la aceptación y cambió el estado de la habitación). Se presentan estas capturas en la figura 3.20.

Cuando el usuario se encuentra frente al paciente, puede ingresar el código correspondiente a la cama manualmente o bien leer el código QR con el celular, como se observa en la figura 3.21.

Una vez que se aceptó desde el sistema el código de la cama, en la aplicación se presenta una pantalla con distintos botones (ver figuras 3.22). Las acciones que se permiten realizar son:

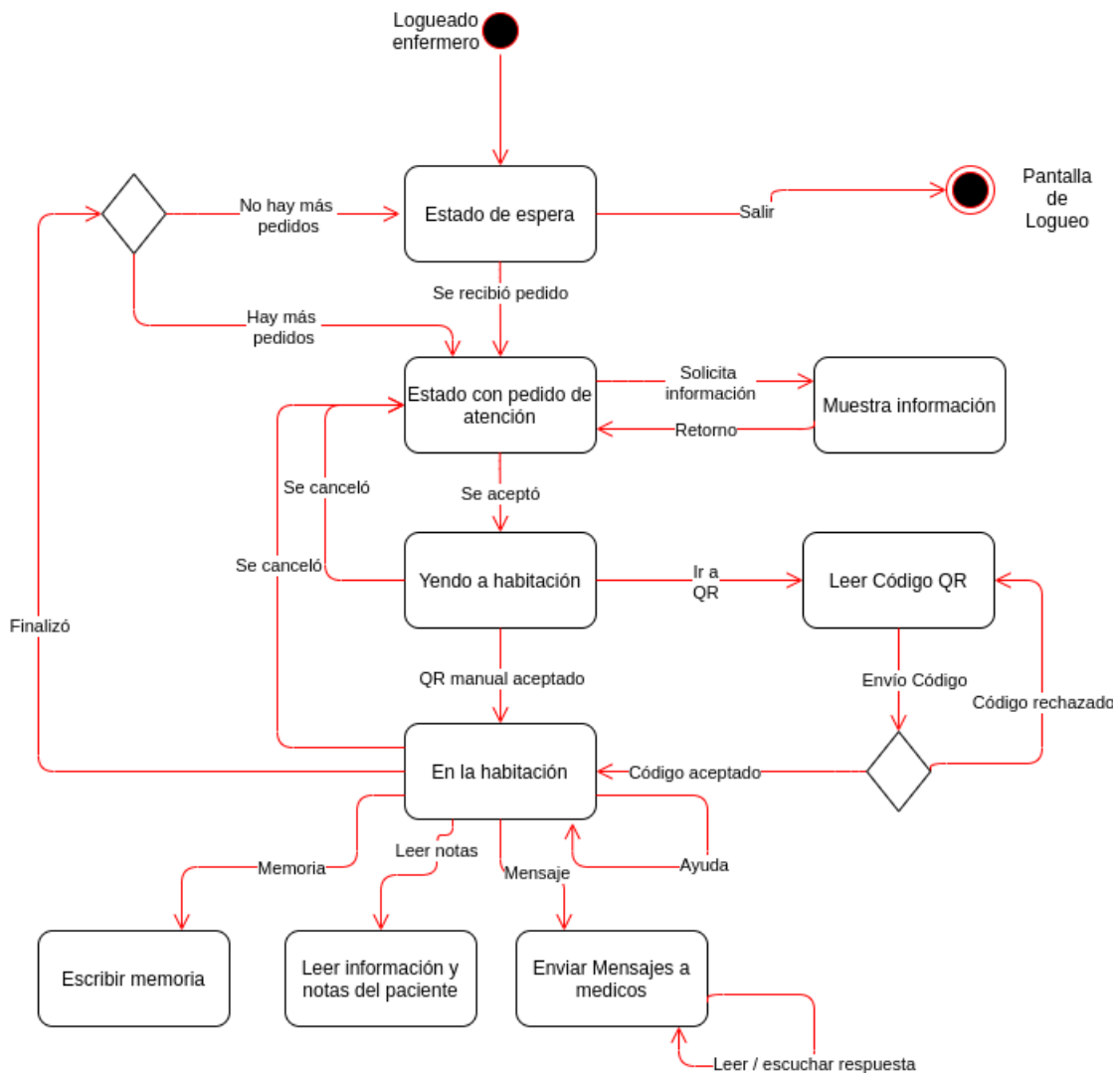


FIGURA 3.19. Diagrama de estados en modo enfermera.

1. Cancelar: envía al sistema automáticamente la solicitud de cancelar la tarea. El sistema vuelve a colocar a la cama en situación de llamada.
2. Listo: envía al sistema automáticamente la solicitud de finalizar la tarea. El sistema coloca a la cama en situación de ocupada.
3. Notas: consulta al sistema las notas referidas al paciente.
4. Mensajes: permite enviar audio o texto a un médico.
5. Ayuda: solicita al sistema que marque a la cama como con solicitud de ayuda para que otros enfermeros puedan socorrer al usuario.
6. Memoria: Permite incorporar un texto que se almacena junto con la tarea al presionar Listo.

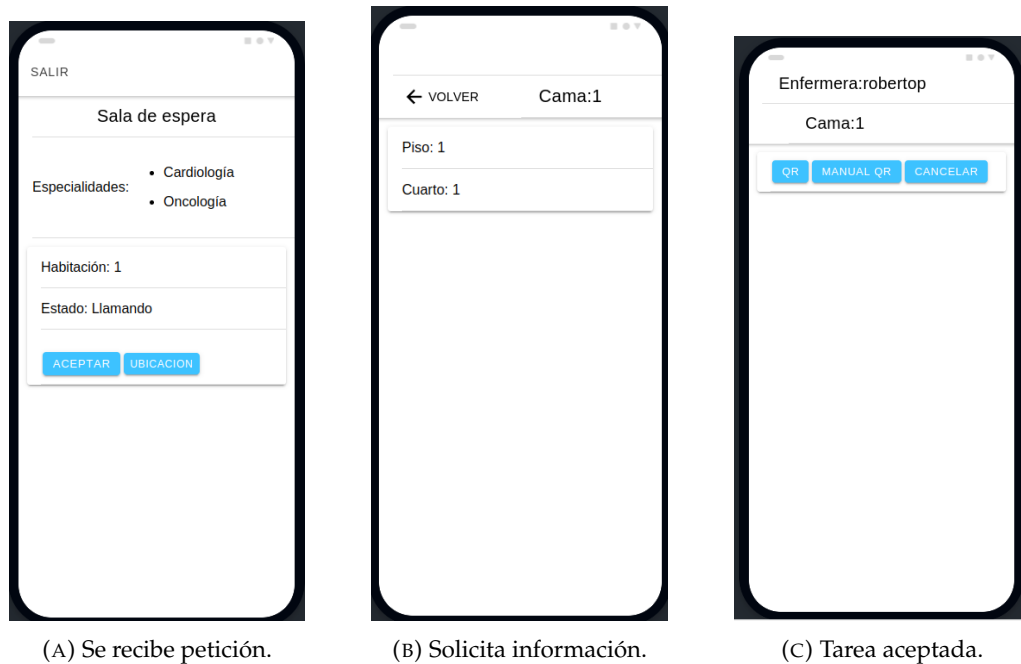


FIGURA 3.20. Recepción de tarea

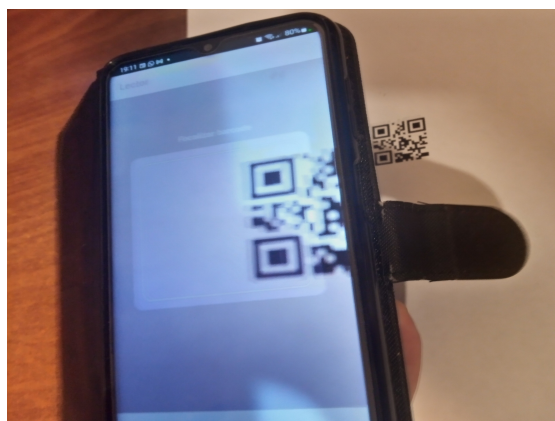


FIGURA 3.21. Captura de QR en la aplicación.



FIGURA 3.22. Ejecución de tarea.

Capítulo 4

Ensayos y resultados

En este capítulo se describen los ensayos realizados, se comentan las herramientas que se utilizaron y se presentan los resultados. Se fragmenta el capítulo en tres partes: pruebas unitarias, integración del sistema y contraste contra requerimientos.

4.1. Pruebas unitarias

En esta sección se presentan las herramientas que se utilizaron para ensayar las distintas partes del sistema en forma separada.

4.1.1. Pruebas del servidor Mosquitto

Para simular mensajes y observar el comportamiento de cada una de las partes del sistema se utilizó la herramienta MQTT explorer [33] que se presenta en la figura 4.1.

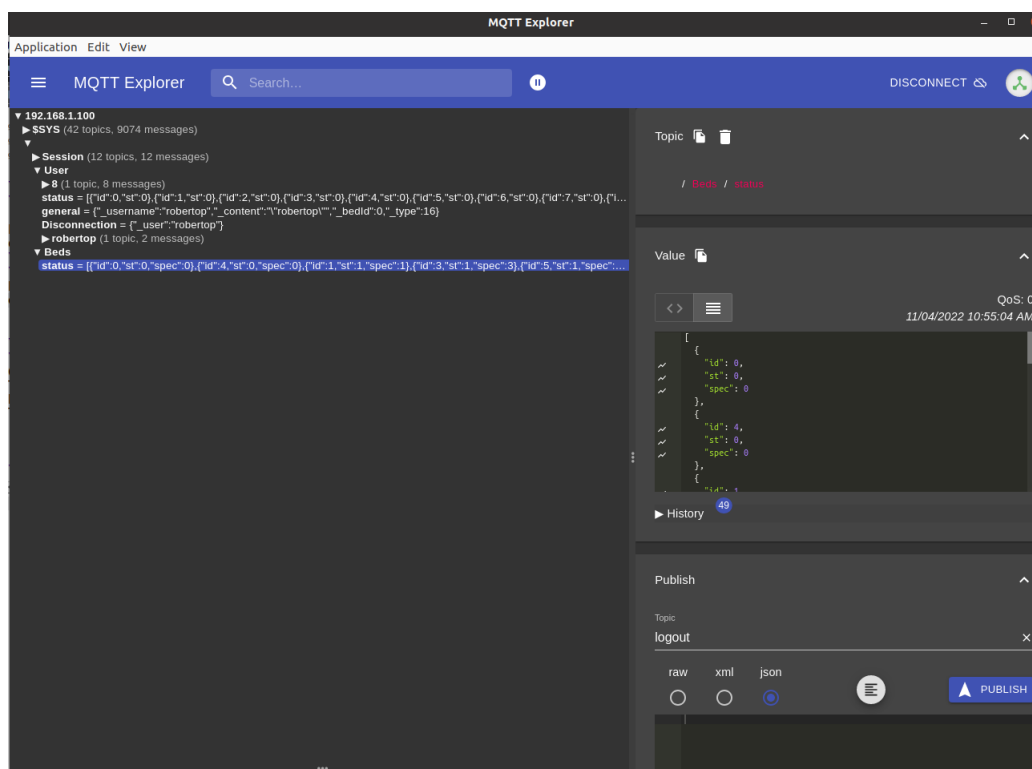


FIGURA 4.1. Imagen de MQTT Explorer.

Esta herramienta fue importante no solo para realizar ensayos sino también para el desarrollo de las funcionalidades propiamente dichas.

4.1.2. Pruebas unitarias de la API Rest

Para simular el logeo y la consulta a la base de datos se utilizó la herramienta PostMan [34], Newman [35] y la consola de administrador de phpMyAdmin mencionada en 2.6.1. En la figura 4.2 se observa el *token* devuelto por el backend al loguearse con las credenciales correspondientes y en 4.3 se observa la respuesta al ingresar una contraseña inadecuada.

En esta secuencia, el cliente (en este caso Postman), envía en el body el nombre de usuario ("username") y la contraseña ("password") en formato JSON. Según lo explicado en la sección 3.5.3, el backend realiza ciertas verificaciones y devuelve un token de sesión en el body.

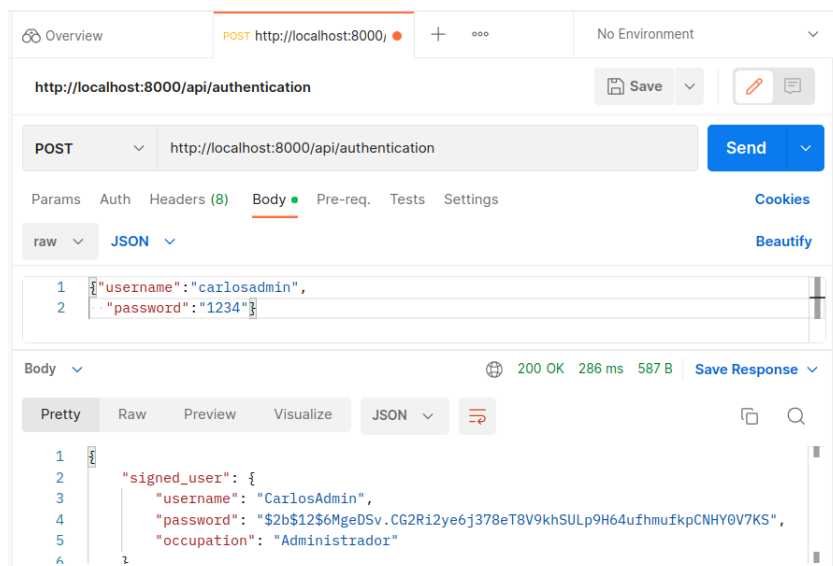


FIGURA 4.2. Logeo en el sistema con Postman.

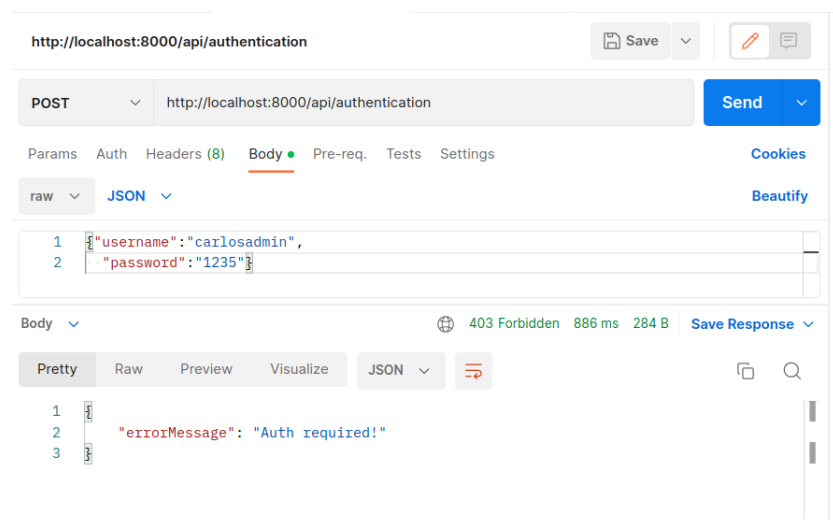


FIGURA 4.3. Rechazo de logeo.

Postman permite automatizar los test utilizando colecciones (que son secuencias de consultas a la API). Las colecciones se pueden exportar a un formato JSON de modo de poder ser utilizadas por otras herramientas.

En un entorno de integración continua y entrega continua, una herramienta de testeo automático permite, una vez definidas las colecciones de test, volver a ejecutarlas luego de haber realizado una modificación, asegurar que no se modificó lo que ya funcionaba y que las mejoras pasan los nuevos test. La ventaja de utilizar newman radica en que al tener un cliente de consola se puede generar un script automático.

Para utilizar Newman, que es el cliente de consola, se deben exportar las colecciones y luego utilizar la línea de comandos con dicha colección (ver figura 4.4). Existen varias opciones de reportes de resultados, en este trabajo se utiliza cli y htmlextra, ejecutando en la consola el código 4.1.

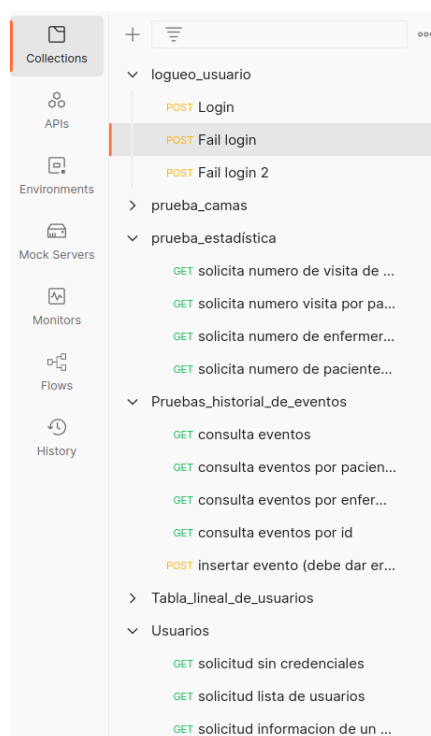


FIGURA 4.4. Colección test de logueo.

```
1 >>newman run ./logueo_usuario.postman_collection.json -r cli,htmlextra
```

CÓDIGO 4.1. Ejecución de Newman en consola.

En la figura 4.5 se observa el reporte por consola.

```

newman

logueo_usuario

→ Login
  POST http://localhost:8000/api/authentication [200 OK, 587B, 393ms]
  ✓ test de logueo correcto, retorna token

→ Fail login
  POST http://localhost:8000/api/authentication [403 Forbidden, 284B, 261ms]
  ✓ test de falla logueo, contraseña incorrecta

→ Fail login 2
  POST http://localhost:8000/api/authentication [403 Forbidden, 284B, 5ms]
  ✓ test de falla logueo, usuario incorrecto

```

	executed	failed
iterations	1	0
requests	3	0
test-scripts	3	0
prerequest-scripts	1	0
assertions	3	0
total run duration: 735ms		
total data received: 407B (approx)		
average response time: 219ms [min: 5ms, max: 393ms, s.d.: 161ms]		

FIGURA 4.5. Reporte en consola de Newman.

4.2. Integración del sistema

En esta sección se explica como se instaló el backend y el servidor web en una máquina remota, se instaló la aplicación en múltiples dispositivos, se desarrolló un dispositivo que simula los eventos de los llamadores de los hospitales, se ensayó el sistema durante una semana. Finalmente se analizan los resultados de la simulación de uso.

Se utilizó como máquina remota una instancia en Amazon Web Services. Los dispositivos móviles en los cuales se instaló la aplicación poseían el sistema Android como sistema operativo.

4.2.1. Instalación del sistema en una instancia de AWS

Con el objetivo de no incorporar costos en las pruebas, se generó una instancia Free Tier en Amazon Web Services, en la cual se instaló Ubuntu, el broker Mosquitto, el backend, la página Web y un servidor NGINX para que funcione como proxy reverso. También se contrató el servicio route 53 que permite personalizar las políticas de ruteo. De esta manera, se puede acceder al sistema desde cualquier dispositivo móvil conectado a una red. Los pasos para configurar el sistema backend en la instancia EC2 con ubuntu son:

1. Loguearse en la instancia con ssh.
2. Setear la base temporal en Buenos Aires Argentina (esto es importante ya que hay eventos de calendario que se deben realizar en el tiempo correspondiente a este huso horario). Esto se presenta en el código 4.2.

```
1  sudo echo "America/Argentina/Buenos_Aires" | sudo tee /etc/  
    timezone  
2  sudo dpkg-reconfigure --frontend noninteractive tzdata  
3  sudo reboot
```

CÓDIGO 4.2. Configuración de zona horaria.

3. Instalar Git:
`sudo apt-get install git`
4. Instalar el broker Mosquitto:
`sudo apt-get install mosquitto`
5. Configurar el broker Mosquitto según la sección 3.2.
6. Instalar Docker y Docker-compose según el sitio [19]
7. Descargar el backend desde GitHub:
`git clone https://github.com/gustavobastian/ServerNurse`
8. Descargar desde [36] el archivo con la base de datos de ejemplo. Luego transferir a la instancia el archivo (utilizar el protocolo SCP) y descomprimir dentro de la carpeta /db (ver [37]). Para descomprimir, dentro de la carpeta /db ejecutar:
`sudo tar xvjf <dirección relativa del archivo comprimido>`

9. Crear el archivo de entorno (.env) según el código 4.3:

```

1 TAG="v1.0.1 "
2
3 ##SECURITY
4 #secret pass
5 JWT_SECRET = <palabra secreta para las contraseñas>
6 #token expiration time
7 JWT_EXP_TIM = 120s
8
9
10 ##MQTT CONFIGURATION
11 ##setting por for using websockets
12 MQTT_CONNECTION = 'ws://<IP de broker>:<puerto de broker>'
13
14 ##server
15 PORT_LOCAL      = 3000
16
17 ##timezone
18
19 TZ = America/Argentina/Buenos_Aires

```

CÓDIGO 4.3. Entorno del backend.

10. Ejecutar Docker-compose up

Los pasos para configurar la página web son:

- Compilar la página web: para ello, dentro del proyecto ejecutar:
"ionic build -prod"
- Comprimir la carpeta www.
- Loguearse con ssh en la instancia EC2.
- Copiar el archivo comprimido en la instancia y descomprimirlo en "/var/www/html".
- Configurar nginx para redirigir a la página (ver apéndice B).

4.2.2. Generación/instalación de la aplicación móvil en Android

Para que la aplicación pueda utilizarse en un dispositivo móvil, la misma debe ser compilada para ejecutarse en código nativo. Con esto se genera el código instalable .apk para Android y .ipa para IOS.

Para poder descargar el archivo ejecutable a un dispositivo Android se debe generar el archivo .apk correspondiente. Para ello se deben ejecutar los siguientes pasos:

1. Generar el código con ionic capacitor: "ionic cap build android"
2. Abrir Android Studio (ver apéndice C).
3. Generar el bundle o en su defecto, conectar el celular en modo debug y descargar la aplicación.

4.2.3. Equipo simulador de llamadores

Se desarrolló un dispositivo que simula los llamadores con un microcontrolador ESP32, una fuente y un teclado matricial. Presionando una tecla, se genera un evento en MQTT. El código y el esquemático del mismo se encuentra en [32] y se presentan algunos detalles del código en el anexo A. Es importante mencionar que el dispositivo desarrollado posee la funcionalidad mínima, pero permite simular múltiples habitaciones (no fue pensado para un uso real).

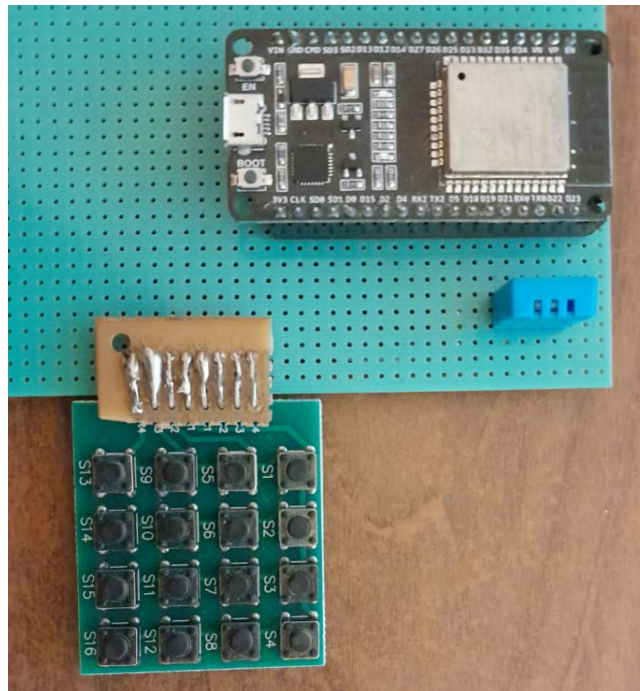


FIGURA 4.6. Simulador de llamadores.

4.2.4. Resultados de las pruebas de integración

El sistema permaneció funcionando de manera adecuada durante una semana completa, generando los eventos programados y permitiendo la interacción de dispositivos.

4.3. Contraste con los requerimientos

En esta subsección se detalla el grado de cumplimiento de los requerimientos relevados en el plan de proyecto.

1. Requerimientos del servidor:

1.1 debe tener instalado el broker mosquitto.

Verificación: Se muestra el funcionamiento del servidor utilizando la aplicación MQTT explorer.

2. Requerimientos de la base de datos:

2.1 Debe poseer una base de datos relacional.

2.2 Debe poseer las siguientes tablas: eventos, pacientes, médicos, enfermeras.

2.2 La base de datos debe poseer datos cargados por default.

Verificación: Se muestra el contenido de la base de datos con la aplicación phpMyAdmin.

3. Requerimientos de la página web

3.1 Debe ser cliente del broker MQTT.

Verificación: Se presenta la configuración del broker. Se presenta el monitoreo de las habitaciones y los usuarios conectándose al broker.

3.2 La página debe poseer funciones de consulta o modificación de la base de datos.

Verificación: Se muestra como la página web permite cambiar contraseñas de usuarios y otros parámetros.

3.3 La página debe permitir observar la estadística de pacientes y enfermeras.

Verificación: Se muestra como la página web permite observar distintas gráficas.

3.4 La página debe contener acceso con usuario y contraseña para cada persona.

Verificación: Se muestra como la página web permite loguear los distintos usuarios.

(*) Cumplimiento parcial: solo permite loguear al usuario administrador.

4. Requerimientos de la aplicación móvil

4.1 y 4.2 La aplicación debe poseer tres modos de uso: médico, enfermera y sistema.

Verificación: Se ingresa a la aplicación en los distintos modos.

4.3 La aplicación en modo enfermera debe permitir leer código QR.

Verificación: Se ingresa a la aplicación en modo enfermera y siguiendo los pasos se obtiene el código QR.

4.4 La aplicación en modo enfermera debe descargar información relevante del paciente.

Verificación: Se ingresa a la aplicación en modo enfermera y siguiendo los pasos se descarga las notas del paciente.

4.5 La aplicación en modo sistema debe mostrar las habitaciones sin atención, según una tabla de prioridades y en caso de igualdad de prioridades mostrar según un orden de llamada.

(*) Cumplimiento parcial: se muestra las habitaciones con una tabla de prioridades pero no según el orden de llamadas.

Verificación: Se ingresa a la aplicación en modo enfermera y siguiendo los pasos se descarga las notas del paciente.

4.6 El modo de usuario médico y el modo usuario enfermera deben poder enviar mensajes de texto o sonido.

Verificación: Se transmiten distintos audios entre participantes.

5. Requerimientos de la documentación

5.1 Documento con información relativa a la base de datos: detalles de la misma y API para acceder.

Verificación: Se observa la presencia de la información en el repositorio de GitHub.

5.2 Memoria del proyecto con diagramas de aplicación móvil y página web.

Verificación: se verifica con este documento.

6. Requerimientos de la integración del sistema

6.1 El sistema debe integrar el funcionamiento del servidor con la base de datos, aplicación web y aplicaciones móviles.

Verificación: se verificó el funcionamiento en un laboratorio con 4 usuarios y un simulador de 8 llamadores durante una semana.

(*) Cumplimiento parcial: Validación: No se pudo validar el sistema en un nosocomio.

7. Requerimientos de la entrega del producto:

7.1 El Código fuente del servidor debe ser subido a dockerhub y compartido con la comunidad.

(*) Cumplimiento parcial: Validación: Se subió el código fuente del servidor a GitHub.

7.2 El Código fuente de la aplicación debe ser subido a GitHub y compartido con la comunidad.

Validación: Se subió el código fuente del servidor a Github.

Capítulo 5

Conclusiones

En este capítulo se presentan los resultados del desarrollo del trabajo, las herramientas aprendidas durante el transcurso de la carrera que fueron fundamentales para la ejecución y las mejoras que se pueden realizar al sistema realizado.

5.1. Resultados obtenidos

El proyecto se pudo finalizar cumpliendo con los requerimientos en un porcentaje muy alto, según se analizó en la subsección 4.3. Los principales logros se resumen en:

- Se desarrolló una base de datos relacional.
- Se instaló y configuró un broker MQTT Mosquitto.
- Se desarrolló un backend con dos API, una utilizando HTTP y otra MQTT.
- Se desarrolló una página web de administración para cargar datos, monitorear el estado de las camas/pacientes y obtener estadísticas del uso del sistema.
- Se desarrolló una aplicación móvil que interactúa con el backend utilizando MQTT y permite recibir notificaciones desde las habitaciones, intercomunicar médicos y enfermeras (ya sea mediante texto o audios), consultar a la base de datos, monitorear el estado del sistema y leer códigos QR. La aplicación fue probada en dispositivos con sistema operativo Android.

La validación del sistema no se pudo realizar en las condiciones deseadas. El principal obstáculo para realizar una validación en un establecimiento radica en que legalmente no se puede utilizar en una situación real, ya que por la normativa nacional el software debe estar homologado en el país.

El principal aporte que presenta este trabajo es el modelo del sistema de backend, con dos APIs totalmente separadas para interactuar con la base de datos. De esta manera, si se quiere escalar el proyecto, se pueden agregar funcionalidades de manera independiente a la página de administración y a la aplicación móvil. Otro aporte importante fue el uso de MQTT para una aplicación de estas características con el modelo de suscripción a tópicos.

5.1.1. Cumplimiento de planificación original

No se pudo seguir con la planificación original ya que se presentaron situaciones no contempladas, que se enumeran a continuación:

1. Agregado de funcionalidades necesarias pero no contempladas en un inicio.
2. Configuración de los dispositivos para la aplicación (fue hablado con el cliente y se llegó a consenso).

5.1.2. Gestión de riesgos

El principal riesgo que se previó era la imposibilidad de desarrollar la aplicación para celulares con sistema IOS debido a la ausencia de un dispositivo físico donde realizar el desarrollo y testeo. Efectivamente se cumplió y no se pudo llegar a satisfacer ese requerimiento.

Los demás riesgos no se dieron por lo que no influyeron en el resultado.

5.1.3. Recursos aprendidos durante la carrera:

En esta sección se presentan las materias de la carrera que más influyeron en el desarrollo de este trabajo. Son enumeradas con su aporte al trabajo:

- Gestión de proyectos: se elaboró un plan de trabajo y el relevamiento de requerimientos.
- Protocolos de Internet, Protocolos de IoT y Diseño de aplicaciones para IOT: se utilizaron los conceptos aprendidos sobre MQTT.
- Arquitectura de datos: se utilizó los conceptos sobre bases de datos.
- Ciberseguridad en Internet de las Cosas: se utilizaron los conceptos sobre seguridad, principalmente en la página web de configuración.
- Testing de sistemas de IoT: se utilizó la técnica de testing del backend utilizando Postman.
- Diseño de páginas Web, Diseño de aplicaciones multiplataformas y Diseño de aplicaciones para Iot: se utilizaron los frameworks, las técnicas y los lenguajes de programación.

5.2. Próximos pasos

Este trabajo sirve de base para la implementación de un sistema de gestión hospitalaria integral. Se plantea un conjunto de mejoras a realizar:

- Generar el código para que la aplicación se ejecute en IOS.
- Utilizar una base de dato clave-valor para reemplazar la lista de usuarios en el monitoreo del backend.
- Mejorar la seguridad utilizando TLS en el broker Mosquitto (por una razón de tiempo no se pudo desarrollar).
- Utilizar otra identificación suplementaria como ser nombre y apellido en el logueo de eventos (se guardan números de referencia de usuario que realizó la acción y número de paciente).
- La estadística que se consulta a la base de datos es mínima. Se recomienda agregar otras consultas.

- Desarrollar un dispositivo llamador que se pueda utilizar en ámbitos hospitalarios.
- Certificar los paquetes de software (aplicación móvil, backend y página web) de modo que se pueda comercializar.
- Mejorar el diseño siguiendo las buenas prácticas presentadas en el documento [38].

Apéndice A

Descripción Simulador de llamador

En este anexo se menciona el dispositivo utilizado en las pruebas de integración del sistema. Solo fue diseñado con el objetivo de facilitar las pruebas de integración. El funcionamiento es el siguiente: una tarea energiza un grupo de pines, que hacen de columnas en el teclado matricial y observa el estado de otros (las filas). Con los valores de columna y fila publica el número de llamador correspondiente.

Para la simulación solo se utilizaron 8 pulsadores de la placa keyboard [39] y la placa procesadora utilizada es la ESP32-WROOM-32 [40].

La dirección IP del broker y el puerto se encuentran codificados en el firmware.

Se utilizó FreeRtos [41] como sistema operativo en tiempo real.

El fragmento de código que publica en el broker se presenta en el código A.1:

```

1
2  /**
3   * @brief simple task for keyboard polling
4   *
5   * @param arg
6   */
7
8  static void keyb_task(void* arg)
9  {   while(true) {
10      int d = keyboardRefresh();
11      if(d!=0){
12          printf("sending event\n");
13          generateJsonKey(buff,d);
14          printf("%s\n",topicCallerEvent);
15          esp_mqtt_client_publish(client , topicCallerEvent , buff , 0, 0, 0)
16      ;
17      }
18      vTaskDelay(200 / portTICK_RATE_MS);
19  }
20 }
```

CÓDIGO A.1. Tarea que publica en el broker la simulación de la llamada.

La función que genera el mensaje a publicar se presenta en el código A.2:

```

1
2 void generateJsonKey(char *buffer , int key)
3 {
4     cJSON *my_json;
```

```
5  cJSON *keyId = NULL;
6  char *string = NULL;
7  my_json = cJSON_CreateObject();
8  if (my_json == NULL)
9  {
10     return;
11 }
12 keyId = cJSON_CreateNumber(key);
13 if (keyId == NULL)
14 {
15     return;
16 }
17 //Populate my_json
18 //my_json.
19 cJSON_AddItemToObject(my_json, "callerId",keyId);
20 //Convert my_json to char array, for sending in an API perhaps
21 string = cJSON_Print(my_json);
22 //printf(string);
23 sprintf(buffer, string);
24 //Free the memory
25 cJSON_Delete(my_json);
26 }
```

CÓDIGO A.2. Función que genera el payload.

Para generar los mensajes se utiliza la librería cJSON desarrollada por Dave Gamble.

Apéndice B

Configuración de servidor Nginx

Nginx [42] es un servidor web liviano y de alta performance diseñado para soportar alto tráfico. Uno de sus casos de uso es ser un servidor proxy inverso. Además es soportado por Linux.

Un proxy inverso es un servidor que acepta todo el tráfico y lo reenvía hacia un recurso específico, como ser el frontend.

Nginx trabaja enfocado a eventos, lo que permite procesar solicitudes de forma asíncrona, ahorrando memoria y espacio.

Para instalar Nginx en una instancia AWS ejecutando Ubuntu (o en cualquier sistema linux), se utiliza el comando "apt install nginx" (como usuario root en la consola).

Para la configuración de Nginx en el servidor se utilizan los siguientes archivos y directorios:

- /etc/nginx : directorio de configuración.
- /etc/nginx/nginx.conf : archivo de configuración principal.
- /etc/nginx/sites-available : directorio donde se pueden guardar bloques de servidor por sitio. Nginx no utiliza los archivos de configuración de este directorios a menos que estén vinculados a sites-enabled. La configuración se realiza en este directorio y luego se habilita estableciendo un vinculo con el otro directorio.
- /etc/nginx/sites-enabled : directorio donde se guardan los bloques de servidor habilitados por sitios.
- /etc/nginx/snippets : directorio donde se pueden guardar fragmentos de configuración.

Los dominios utilizados en las pruebas de este trabajo son:

- Frontend: `http://www.gabiot.com.ar:8024`
- Backend (generado por docker): `http://www.gabiot.com.ar:8000`
- phpMyAdmin (generado por docker): `http://www.gabiot.com.ar:8001`
- Broker MQTT (utilizando websockets): `http://www.gabiot.com.ar:9001`

Dentro de la carpeta sites-available se guarda un archivo de nombre "Nurse.com.conf" con el contenido presentado en el código B.1:

```
1 server {  
2     listen 8024 default_server;  
3     root /var/www/html;  
4     index index.html;  
5 }
```

CÓDIGO B.1. Archivo
/etc/nginx/sites-available/Nurse.com.conf.

En la carpeta sites-enabled se genera el enlace dinámico al archivo anterior mediante la instrucción:

- "sudo ln -s /etc/nginx/sites-available/Nurse.com.conf /etc/nginx/sites-enabled/Nurse.com.conf "

El archivo de la página se debe descomprimir dentro de "/var/www/html" y luego de ello reiniciar el servidor Nginx con los comandos:

- "sudo systemctl stop nginx"
- "sudo systemctl start nginx"

Apéndice C

Compilación para aplicaciones Android

En esta sección se presenta, brevemente, los pasos para regenerar el código utilizado en un dispositivo con sistema operativo Android (archivo .apk). Para el desarrollo de aplicaciones se consultó el documento [38] donde se menciona la importancia de la seguridad en las aplicaciones y en el resguardo de los datos.

Para poder compilar nativamente la aplicación es necesario poseer instalado Android Studio [43] y asignarle los permisos al software. Eso se logra incorporando componentes al android manifest, como se observa en el código C.1. Además es necesario incorporar algunas librerías al archivo variables.gradle y setear parámetros en build.gradle. Para una descripción detallada de todo el proceso de generación de código ejecutable se puede recurrir a la documentación oficial [44].

El archivo Android Manifest describe la información esencial de la aplicación para que pueda ser interpretadas por las herramientas de generación de código ejecutable, el sistema operativo Android y Google Play.

Entre muchas otras cosas, el archivo de manifiesto (ver código C.1) debe declarar lo siguiente:

- El nombre del paquete de la aplicación, que normalmente coincide con el espacio de nombres del código. Las herramientas de compilación de Android usan esto para determinar la ubicación de las entidades de código cuando se compila el proyecto. Al empaquetar la aplicación, las herramientas de compilación sustituyen este valor por el ID de aplicación de los archivos de compilación de Gradle, que se utiliza como identificador único de la aplicación en el sistema y en Google Play.
- Los componentes de la aplicación, que incluyen todas las actividades, servicios, receptores de emisiones y proveedores de contenido. Cada componente debe definir propiedades básicas, como el nombre de su clase Kotlin o Java. También puede declarar capacidades, como las configuraciones de dispositivos que puede manejar, además de filtros de intents que describen cómo se puede iniciar el componente.
- Los permisos que necesita la aplicación para acceder a las partes protegidas del sistema o a otras aplicaciones. También declara cualquier permiso que otras aplicaciones deben tener si necesitan acceder al contenido de esta aplicación.

El archivo build.gradle de nivel superior (ver código C.3), ubicado en el directorio raíz del proyecto, define dependencias que se aplican a todos los módulos de tu proyecto y en el archivo variables.gradle (ver código C.2) se indica la versión de APK que se utiliza. La API utilizada es Android API:33.

```

1 package="com.gabiot.Enfermera">
2     <!-- Permissions -->
3
4     <uses-permission android:name="android.permission.INTERNET" />
5     <uses-permission android:name="android.permission.
6     READ_EXTERNAL_STORAGE" />
7     <uses-permission android:name="android.permission.
8     WRITE_EXTERNAL_STORAGE" />
9     <uses-permission android:name="android.permission.
10    ACCESS_NETWORK_STATE" />
11    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"
12    />
13
14    <uses-permission android:name="android.permission.WAKE_LOCK" />
15    <uses-permission android:name="android.permission.READ_PHONE_STATE"
16    />
17    <uses-permission android:name="android.permission.CAMERA" />
18    <uses-sdk tools:overrideLibrary="com.google.zxing.client.android" />
19    <uses-permission android:name="android.permission.RECORD_AUDIO" />
20
21    <application
22        android:allowBackup="true"
23        android:icon="@mipmap/ic_launcher"
24        android:label="Enfermera"
25        android:roundIcon="@mipmap/ic_launcher_round"
26        android:supportRtl="true"
27        android:theme="@style/AppTheme"
28        android:usesCleartextTraffic="true"
29        android:hardwareAccelerated="true">
30        <!-- Mqtt Service -->
31        <service
32            android:enabled="true"
33            android:name="org.eclipse.paho.android.service.MqttService"
34            android:exported="false"
35        />
36    </application>
37 ...

```

CÓDIGO C.1. Modificaciones al Android Manifest.

```

1
2 ext {
3     minSdkVersion = 21
4     compileSdkVersion = 33
5     targetSdkVersion = 33
6     androidxActivityVersion = '1.2.0'
7     androidxAppCompatVersion = '1.2.0'
8     androidxCoordinatorLayoutVersion = '1.1.0'
9     androidxCoreVersion = '1.3.2'
10    androidxFragmentVersion = '1.3.0'
11    junitVersion = '4.13.1'
12    androidxJUnitVersion = '1.1.2'
13    androidxEspressoCoreVersion = '3.3.0'
14    cordovaAndroidVersion = '7.0.0'
15 }

```

CÓDIGO C.2. Archivo variables.gradle.

```

1 apply plugin: 'com.android.application'

```



```

2
3 android {
4     //compileSdkVersion rootProject.ext.compileSdkVersion
5     compileSdk 31
6
7     defaultConfig {
8         applicationId "com.gabiot.Enfermera"
9         minSdkVersion rootProject.ext.minSdkVersion
10        targetSdkVersion rootProject.ext.targetSdkVersion
11        versionCode 1
12        versionName "1.0"
13        testInstrumentationRunner "androidx.test.runner.
AndroidJUnitRunner"
14        aaptOptions {
15            // Files and dirs to omit from the packaged assets dir,
modified to accommodate modern web apps.
16            // Default: https://android.googlesource.com/platform/
frameworks/base/+ /282e181b58cf72b6ca770dc7ca5f91f135444502/tools/
aapt/AaptAssets.cpp#61
17            ignoreAssetsPattern '!svn:!.git:!.ds_store:!.scc:!.*:!CVS:!
thumbs.db:!picasa.ini:!*~'
18        }
19    }
20    buildTypes {
21        release {
22            minifyEnabled false
23            proguardFiles getDefaultProguardFile('proguard-android.txt')
, 'proguard-rules.pro'
24        }
25    }
26 }
27
28 repositories {
29     flatDir {
30         dirs '../capacitor-cordova-android-plugins/src/main/libs', 'libs
31     }
32
33     /*maven {
34         url "https://repo.eclipse.org/content/repositories/paho-releases
/"
35     }*/
36
37     maven {
38         url "https://repo.eclipse.org/content/repositories/paho-
snapshots/"
39     }
40
41 }
42
43 dependencies {
44     implementation fileTree(include: ['*.jar'], dir: 'libs')
45     //noinspection GradleDependency
46     implementation "androidx.appcompat:appcompat:
$androidxAppCompatVersion"
47
48     implementation project(':capacitor-android')
49     testImplementation "junit:junit:$junitVersion"
50     //noinspection GradleDependency
51     androidTestImplementation "androidx.test.ext:junit:
$androidxJUnitVersion"
52     androidTestImplementation "androidx.test.espresso:espresso-core:
$androidxEspressoCoreVersion"

```

```
53     implementation project(':capacitor-cordova-android-plugins')
54     api ('org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.2.5')
55     implementation 'com.google.android.material:material:1.5.0'
56
57     implementation ('org.eclipse.paho:org.eclipse.paho.android.service
58         :1.1.1')
59 }
60 apply from: 'capacitor.build.gradle'
61
62 try {
63     def servicesJSON = file('google-services.json')
64     if (servicesJSON.text) {
65         apply plugin: 'com.google.gms.google-services'
66     }
67 } catch (Exception e) {
68     logger.info("google-services.json not found, google-services plugin
69         not applied. Push Notifications won't work")
70 }
```

CÓDIGO C.3. Archivo build.gradle(Module:android.app).

En Android Studio se puede realizar un debug de la aplicación. Cuando se ejecuta la aplicación consulta por los permisos correspondientes.

Bibliografía

- [1] Sankeerthana Neelam. «Internet of Things in Healthcare». En: (2017).
- [2] Lyman Chapin Karen Rose Scott Eldridge. «The Internet of Things: an Overview». En: (2015).
- [3] Carlos Ruben Domenje. «Sistema de gestión remota de dispositivo conversor Modbus a MQTT». En: (2021).
- [4] TigerConnect. *Secure Healthcare Communication and Collaboration*. <https://tigerconnect.com/products/clinical-collaboration/>. Dic. de 2021. (Visitado 25-10-2022).
- [5] Juliana De Groot. *What is HIPAA Compliance?* <https://digitalguardian.com/blog/what-hipaa-compliance>. Abr. de 2022. (Visitado 25-10-2022).
- [6] MATT HALBLEIB. *What is HITRUST Compliance?* <https://www.securitymetrics.com/blog/what-hitrust-compliance>. Dic. de 2021. (Visitado 25-10-2022).
- [7] OASIS. *MQTT Version 5.0. Edited by Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. 07 March 2019. OASIS Standard.* <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>. <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>. Mar. de 2019. (Visitado 02-07-2022).
- [8] Vasil Sarafov. «Comparison of IoT Data Protocol Overhead». En: (2018).
- [9] W3schools. *MySQL Tutorial*. <https://www.w3schools.com/mysql/default.asp/>. Ene. de 2021. (Visitado 20-01-2022).
- [10] Meta Platforms. *React*. <https://es.reactjs.org/>. Ene. de 2022. (Visitado 30-10-2022).
- [11] OpenJS Foundation. *jQuery, write less, do more*. <https://jquery.com/>. Ene. de 2022. (Visitado 30-10-2022).
- [12] Evan You. *The Progressive JavaScript Framework*. <https://vuejs.org/>. Ene. de 2022. (Visitado 30-10-2022).
- [13] Google. *The modern web developer's platform*. <https://angular.io/>. Ene. de 2022. (Visitado 30-10-2022).
- [14] <https://ionicframework.com/>. *The mobile SDK for the Web*. <https://ionicframework.com/>. Ene. de 2022. (Visitado 30-10-2022).
- [15] IBM. *Calidades de servicio proporcionadas por un cliente MQTT*. <https://www.ibm.com/docs/es/ibm-mq/9.1?topic=concepts-qualities-service-provided-by-mqtt-client>. Ago. de 2022. (Visitado 30-09-2022).
- [16] HiveMQ. *MQTT over WebSockets - MQTT Essentials Special*. <https://www.hivemq.com/blog/mqtt-essentials-special-mqtt-over-websockets/>. Abr. de 2015. (Visitado 02-04-2022).
- [17] Eclipse Foundation. *Eclipse Mosquitto An open source MQTT broker*. <https://mosquitto.org/>. Ene. de 2022. (Visitado 15-03-2022).

- [18] Mozilla.org. *Generalidades del protocolo HTTP*. <https://developer.mozilla.org/es/docs/Web/HTTP/Overview>. Ene. de 2022. (Visitado 30-10-2022).
- [19] Agustin Bassi. *Introducción al ecosistema Docker*. https://www.gotoiot.com/pages/articles/docker_intro/index.html. Feb. de 2021. (Visitado 15-01-2022).
- [20] Mauricio Collazos. *Una guía no tan rápida de Docker y Kubernetes*. <https://medium.com/ingeniería-en-tranqui-finanzas/una-guía-no-tan-rápida-de-docker-y-kubernetes-933f5b6709df>. Jun. de 2018. (Visitado 15-02-2022).
- [21] Atlantic Technologies. *¿Como se dividen las bases de datos y qué función tiene cada una?* <https://en.atlantictech.io/blog/base-de-datos-como-se-dividen>. Ene. de 2021. (Visitado 20-10-2022).
- [22] Oracle. *What is a Database*. <https://www.oracle.com/database/what-is-database/>. Ene. de 2022. (Visitado 15-10-2022).
- [23] Robin Nixon. *Aprender PHP, MySQL y JavaScript, 5ta edición*. Marcombo, 2019.
- [24] Fundación OpenJS. *Node.js*. <https://nodejs.org/en/>. Ene. de 2022. (Visitado 30-10-2022).
- [25] Fundación OpenJS. *Express Infraestructura web rápida, minimalista y flexible para Node.js*. <https://expressjs.com/es/>. Ene. de 2022. (Visitado 30-10-2022).
- [26] Auth0. *JWT*. <https://jwt.io/>. Ene. de 2022. (Visitado 30-10-2022).
- [27] JavaTpoint. *Ionic History*. <https://www.javatpoint.com/ionic-history>. Ene. de 2019. (Visitado 30-10-2022).
- [28] OpenJS Foundation. *Electron-Build cross-platform desktop apps with JavaScript, HTML, and CSS*. <https://www.electronjs.org/>. Ene. de 2022. (Visitado 30-10-2022).
- [29] Ionic Open Source. *A cross-platform native runtime for web app*. <https://capacitorjs.com/>. Ene. de 2022. (Visitado 30-10-2022).
- [30] Cardozo et al. *node.bcrypt.js*. <https://www.npmjs.com/package/bcrypt>. Ene. de 2022. (Visitado 30-09-2022).
- [31] Okta Inc. *jsonwebtoken*. <https://www.npmjs.com/package/jsonwebtoken>. Ene. de 2018. (Visitado 30-09-2022).
- [32] Highcharts. *Highcharts*. <https://www.highcharts.com/>. Ene. de 2022. (Visitado 30-09-2022).
- [33] Thomas Nordquist. *MQTT-Explorer - An all-round MQTT client that provides a structured topic overview*. <http://mqtt-explorer.com/>. Ene. de 2022. (Visitado 30-09-2022).
- [34] PostMan. *PostMan : Build APIs together*. <https://www.postman.com/>. Ene. de 2022. (Visitado 30-09-2022).
- [35] PostmanLabs. *Newman, the cli companion for postman*. <https://www.npmjs.com/package/newman>. Mar. de 2022. (Visitado 09-11-2022).
- [36] Gustavo Bastian. *Base de datos ejemplo*. <https://drive.google.com/file/d/1eWSW7uG1hFr87aKrVnCOLjj4MOsV8Xzu/view?usp=sharing>. Ene. de 2022. (Visitado 23-11-2022).
- [37] Inc. Amazon Web Services. *File transfer: Local Windows or MAC PC to Linux AWS EC2*.

- <https://docs.aws.amazon.com/managedservices/latest/appguide/qs-file-transfer.html>. Ene. de 2022. (Visitado 23-11-2022).
- [38] Dirección Nacional de Protección de Datos Personales y Fundación Sadosky. *Guía de Buenas Prácticas para el Desarrollo de Apps*. Fundación Sadosky, 2015.
- [39] Proto Supplies. *Tactile Keypad 4X4 Matrix*. <https://protosupplies.com/product/tactile-keypad-4x4-matrix/>. Sep. de 2022. (Visitado 16-11-2022).
- [40] Descubre arduino. *ESP32, módulo ESP32-WROOM GPIO Pinout*. <https://descubrearduino.com/esp32-modulo-esp32-wroom-gpio-pinout/>. Oct. de 2022. (Visitado 17-11-2022).
- [41] Amazon Web Services. *FreeRTOS™ Real-time operating system for microcontrollers*. <https://www.freertos.org/>. Ene. de 2022. (Visitado 17-11-2022).
- [42] Linode. *How to Configure NGINX*. <https://www.linode.com/docs/guides/how-to-configure-nginx/>. Sep. de 2021. (Visitado 16-11-2022).
- [43] Google. *Android Studio developers*. <https://developer.android.com/studio>. Ene. de 2022. (Visitado 17-11-2022).
- [44] Gradle Inc. *Gradle build language*. https://docs.gradle.org/current/userguide/writing_build_scripts.html. Ene. de 2021. (Visitado 17-11-2022).