



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:

Sergio Burgos (UTN, Facultad Regional Paraná)

Daniel Iván Cruz Flores (UBA, Facultad de Ingeniería)

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.

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
2.2. Descripción del protocolo MQTT	6
2.2.1. Uso de WebSockets en MQTT	8
2.2.2. Broker MOSQUITTO	9
2.3. Descripción del protocolo HTTP	9
2.4. Componentes del sistema	10
2.5. Sistemas de contenedores Docker	10
2.5.1. Uso de Docker-compose	11
2.6. Introducción a las bases de datos	12
2.6.1. Descripción de MySQL	12
2.7. Frameworks/librerías para desarrollo web/móvil	15
2.7.1. Librerías Node.js, Express y JWT	15
2.7.2. Librería Eclipse Paho	15
2.7.3. Framework Ionic/Angular, Capacitor y Android Studio	15
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	20
3.5.1. Descripción de las clases para monitoreo del sistema	22
3.5.2. Descripción de API MQTT para la mensajería de la aplicación móvil	23
3.5.3. Descripción de API REST para aplicación Web	26
3.6. Página Web	28
3.6.1. Estructura y organización del software	29
3.6.2. Acceso de usuario	30
3.6.3. Monitoreo del sistema	31
3.6.4. Gestión de tareas programadas	31
3.6.5. Estadísticas del sistema	32

3.7. Aplicación Móvil	32
3.7.1. Estructura y organización del software	32
3.7.2. Configuración del broker y acceso de usuario	33
3.7.3. Modo administrador	33
3.7.4. Modo médico	33
3.7.5. Modo enfermera	33
3.7.6. Interacción médico-enfermera	33
3.8. Contraste con los requerimientos	34
4. Ensayos y resultados	35
4.1. Generación del sistema de pruebas	35
4.2. Pruebas unitarias	35
4.2.1. Pruebas del servidor Mosquitto	35
4.2.2. Pruebas unitarias de la API Rest	36
4.3. Integración del sistema	37
4.3.1. Instalación del sistema en una instancia de AWS	37
4.3.2. Generación/instalación de la aplicación móvil	37
4.3.3. Equipo simulador de llamadores	37
4.3.4. Resultados de utilizar el sistema	37
5. Conclusiones	39
5.1. Notas sobre el sistema desarrollado	39
5.1.1. Cumplimiento de requerimientos	39
5.1.2. Cumplimiento de planificación original	41
5.1.3. Gestión de riesgos	41
5.1.4. Evaluación de las técnicas presentadas durante la carrera . .	42
5.2. Trabajo futuro	42
Bibliografía	43

Índice de figuras

1.1. Infraestructura Hospitalaria.	1
1.2. Componentes del sistema.	4
2.1. Esquema de un sistema MQTT.	7
2.2. Uso de WebSockets con MQTT.	8
2.3. Pila de comunicaciones en un navegador.	9
2.4. División del sistema.	11
2.5. Página de gestión phpMyAdmin.	13
3.1. Tabla Usuarios.	18
3.2. Tabla Camas.	18
3.3. Tabla pacientes.	19
3.4. Relación médicos-pacientes.	19
3.5. Relación notas-pacientes.	19
3.6. Tabla de eventos programados.	20
3.7. Tabla de registro de eventos.	20
3.8. Estructura de directorio (backend).	21
3.9. Página web.	28
3.10. Estructura de carpetas página web.	29
3.11. Monitoreo de camas.	31
3.12. Tareas programadas.	32
3.13. Pantalla inicial de la aplicación.	33
4.1. Imagen de MQTT Explorer.	35
4.2. Logueo en el sistema con Postman.	36
4.3. Rechazo de logueo.	36

Índice de tablas

1.1. Modelo de capas IoT	2
2.1. Comandos SQL	13
2.2. Tabla 1 Ejemplo SQL	14
2.3. Tabla 2 Ejemplo SQL	14
3.1. Tipos de mensajes MQTT	25

***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].

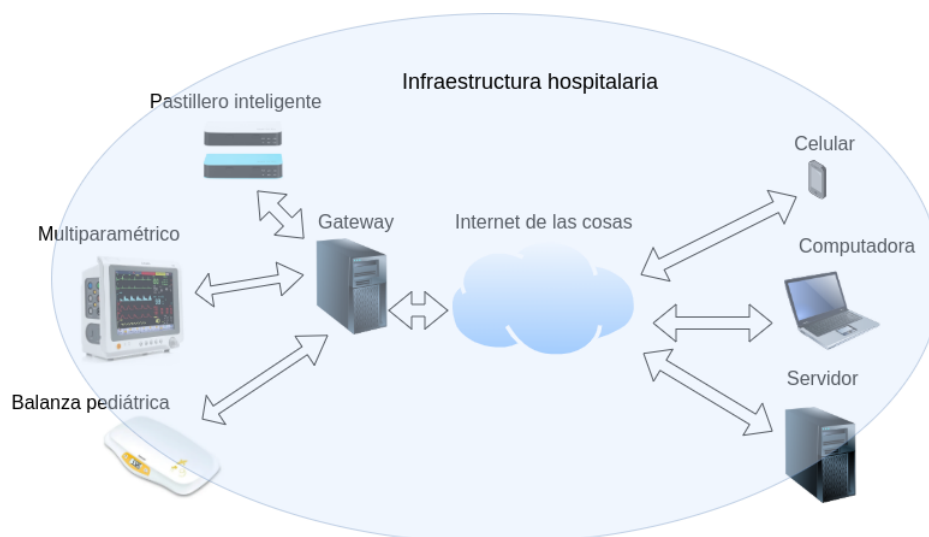


FIGURA 1.1. Infraestructura Hospitalaria.

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 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 que 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 3.1 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 anterior, 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 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.

El alcance del trabajo consiste en:

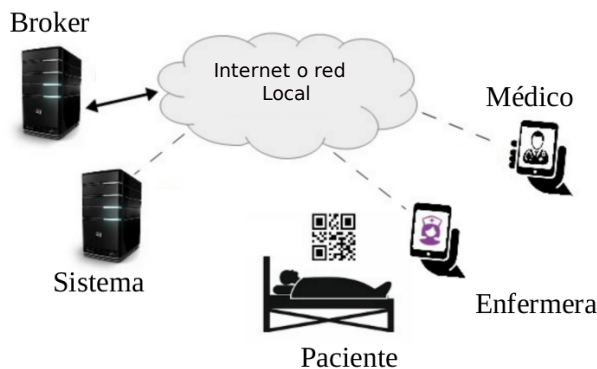


FIGURA 1.2. Componentes del sistema.

- 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 paciente, 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

En la ingeniería de software existe una arquitectura llamada cliente/servidor, en el cual la tarea a desarrollar se distribuye entre los proveedores de servicios o recursos y quienes los consultan llamados clientes o demandantes. En la programación web, se suele dividir el modelo en 2 partes: *backend* y *frontend*.

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 2 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 accesorios 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, 2.7.2 se explicará el funcionamiento del *backend*.

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. El equipo que desarrolla el *frontend* consiste en programadores diseñadores UX/UI (del

ingles *User Experience / User Interface*, Experiencia de Usuario/Interfaz de Usuario) y diseñadores gráficos. Las más conocidas tecnologías 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 de *frontend* mediante librerías como React [10], JQuery [11] o Frameworks como ser Vue [12], Angular [13], 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 2.7.3, se presenta una breve introducción.

2.2. 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 CONNACK 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 2.1 se presentan todos los componentes de una red MQTT:

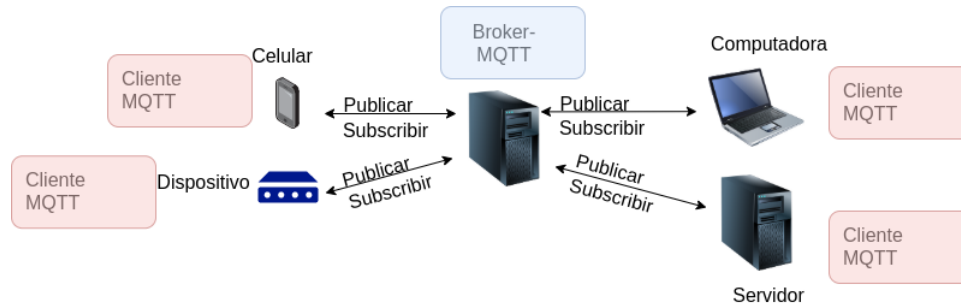


FIGURA 2.1. Esquema de un sistema MQTT.

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].

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/SLL 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].

La carga útil del mensaje debe ser codificada en UTF-8 (*8-bit Unicode Transformation Format*, Formato Unicode de Codificación de 8-bits) [7].

MQTT 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 mas 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. En el segundo par de transmisiones, el remitente indica al destinatario que puede completar el proceso del mensaje, "PUBREL". Si el remitente no recibe un acuse de recibo del "PUBREL" mensaje, el "PUBREL" se envía de nuevo hasta que se recibe un acuse de recibo. El remitente suprime el mensaje que ha guardado cuando recibe el acuse de recibo al "PUBREL" mensaje. 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.

2.2.1. 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 2.2 se muestra conceptualmente como se encapsula la información del protocolo MQTT dentro de una trama WebSocket:

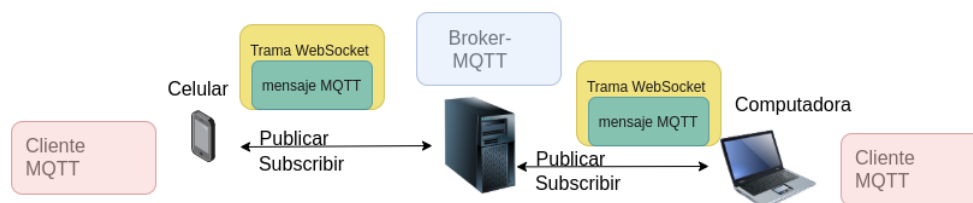


FIGURA 2.2. Uso de WebSockets con MQTT.

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.

2.2.2. 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.

2.3. 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 2.3 se observa donde se ubica el protocolo dentro del modelo de comunicación TCP/IP.

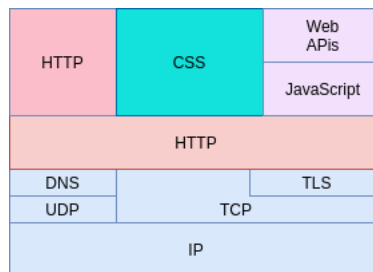


FIGURA 2.3. Pila de comunicaciones en un navegador.

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]:

- 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 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 reusa la conexión.

En el encabezado del mensaje http se puede introducir información de la sesión. Lo cual es muy importante en referencia a la seguridad.

2.4. Componentes del sistema

El sistema desarrollado se lo puede descomponer en seis partes definidas:

- 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 operaciones.
- Servidor web/ página 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.
- Sistema: 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.

Como se puede observar, se trata de un sistema distribuido y sus partes pueden estar ubicadas en distintas localidades.

Con el objetivo de facilitar el desarrollo se utilizó un sistema de contenedores que permite correr tres de las 6 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 una sola aplicación que provee ambos servicios: el sistema y el servidor REST. En la figura 2.4 se presenta un esquema general según esta división.

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 estandar, 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 estandar en la industria del Software [19].

Un contenedor de software es un componente aislado que se ejecuta como un proceso mas 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 ejecutandose en el sistema [19].

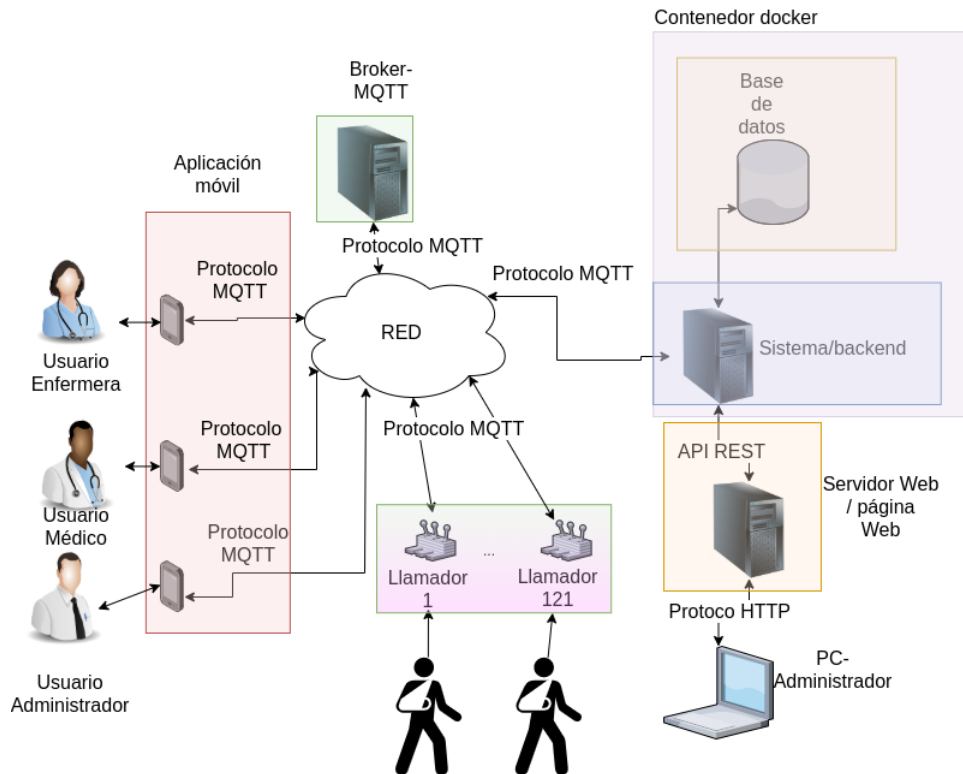


FIGURA 2.4. División del sistema.

Resumo brevemente las partes del ecosistema Docker, para información más detallada consultar[19] o bien [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 mas 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 en un archivo de texto llamado "docker-compose.yml".

Con esta herramienta se facilita el paso de parámetros al contenedor y posibilita generar imágenes complejas utilizando e interconectando distintos contenedores.

Como ejemplo, en [21] se presenta un archivo para ejecutar un contenedor *Wordpress* junto con una base de datos MySQL.

2.6. Introducción a las bases de datos

Que es un dato? Es un elemento o característica que permite hacer la descripción de un objeto [22].

Una base de datos es una colección de datos (o información) 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 [21].

Durante los años 80 se popularizó el tipo de base de dato 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 con contribuciones de 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 estan diseñadas para manejar grandes volumentes de datos [22].

2.6.1. Descripción de MySQL

En este trabajo se seleccionó el DBMS MySQL. Es un sistema de gestión de bases de datos con mas de 10 millones de instalaciones. Fue desarrollado a mediados de la decada 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 2.5 se puede observar la interfaz:

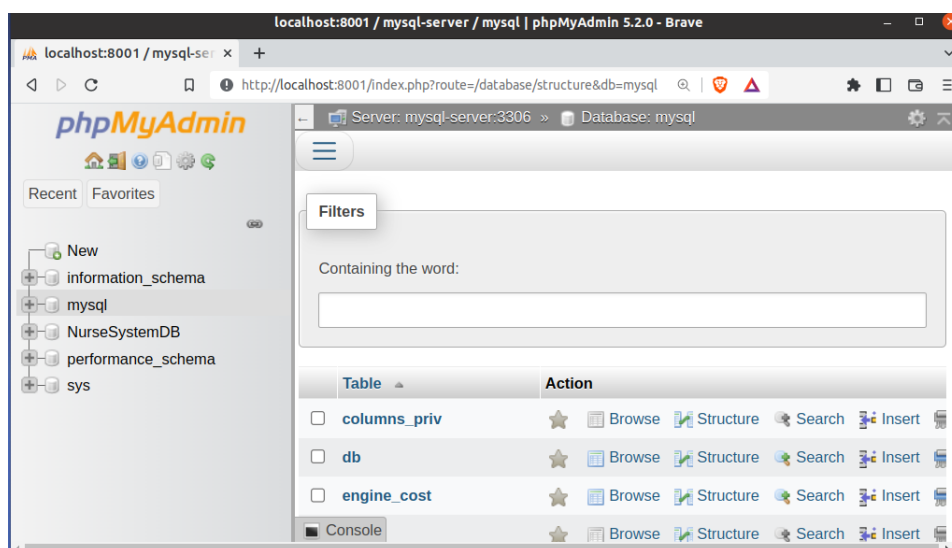


FIGURA 2.5. Página de gestión phpMyAdmin.

Las sentencias SQL finalizan siempre con el símbolo ‘;’.

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 multiples 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
```

Uno de las multiples 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
```

Existe muchas referencias sobre programación con MySQL, para un estudio mas profundo se puede utilizar tutoriales como [9].

2.7. Frameworks/librerías para desarrollo web/móvil

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], el cual es un *runtime environment* (entorno de ejecución) de javascript. Node.js se encuentra orientada a eventos asíncronos y está diseñada 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 de que se aumentan las conexiones.

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

Para más información sobre Node.js consultar [24].

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 incorporar software *Middleware*.

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. Librería Eclipse Paho

Para interconectar el programa servidor escrito en Node.js con el broker MQTT se utiliza la librería Paho MQTT [27].

En las aplicaciones móviles se utilizó un wrapper (rsup - MQTT) que utiliza la misma librería [28].

2.7.3. Framework Ionic/Angular, Capacitor y Android Studio

Drifty.co introdujo Ionic en el año 2013 [29]. 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 [30]. 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 microfono, la cámara, la información de posicionamiento. Junto con Ionic se utiliza Capacitor [31], 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 como 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. Es necesario aclarar que es 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.

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`:

```
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

En este archivo, se especifica los puertos que se utilizará, en este caso, el puerto 9001 con el protocolo Websockets. Se permite la interconexión de cualquier cliente y el archivo donde se guardará un log de todos los eventos se encuentra en la ubicación `/var/log/mosquitto/mosquitto.log`.

3.3. Sistema Docker


El archivo docker-compose.yml contiene los siguientes servicios instalados:

- servidor mysql:

3.4. Base de datos del sistema

En esta sección se describirán las distintas tablas que se almacenan en la base de datos y son útiles para el sistema. Estas tablas pueden ser observadas y editadas con el frontend phpMyAdmin (pero no es recomendable).

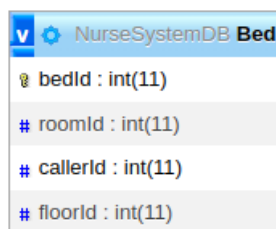
1. Tabla usuarios (*User*): tabla que contiene la información de los usuarios del sistema. El campo id se incrementa automáticamente, es decir, al incorporar un nuevo usuario al sistema la base de datos asigna el id (se utiliza este campo como llave primaria). En el campo contraseña se almacena el hash de la misma (generado con Bcrypt [32]). En el campo ocupación se almacena uno de valores: "Enfermero", "Médico", "Administrador". El campo estado (*State*) se utiliza en versiones posteriores del sistema.



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 información de las camas como ser su identificación (como en el ítem anterior se incrementa automáticamente y es clave primaria), ubicación (piso y cuarto) y el número de dispositivo llamador.



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 puede ser 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).

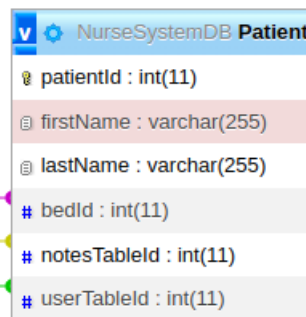


FIGURA 3.3. Tabla pacientes.

4. Relación médicos-pacientes : para generar la relación se utilizan un par de tablas intermedias que permiten que un mismo paciente posea varios médicos tratándolo. Además, un paciente puede utilizar a los mismos médicos de otro paciente... es una de las ventajas de las bases de datos relacionales.

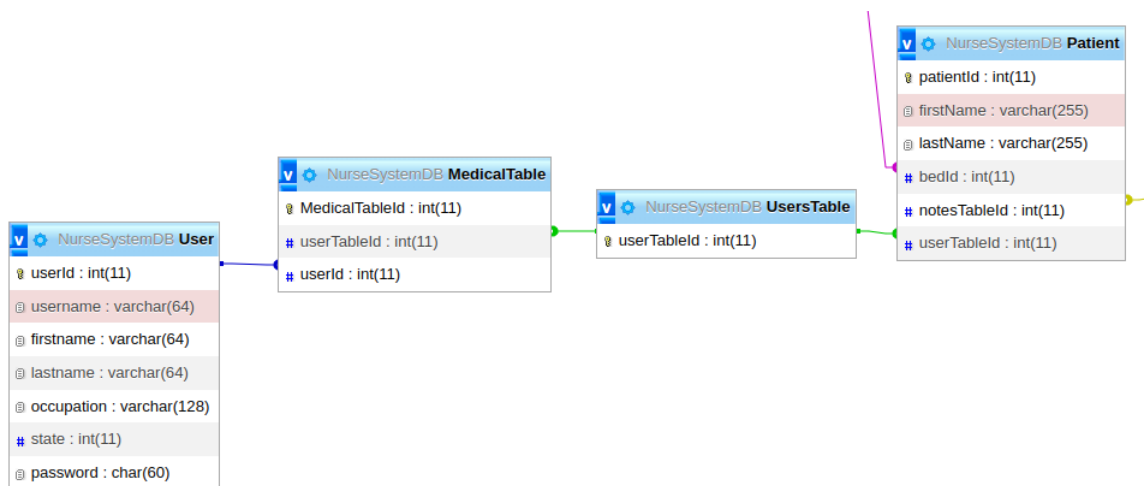


FIGURA 3.4. Relación médicos-pacientes.

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.

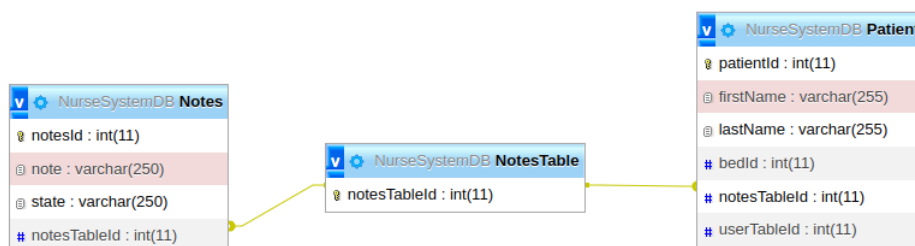
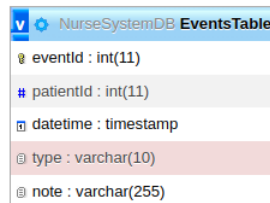


FIGURA 3.5. Relación notas-pacientes.

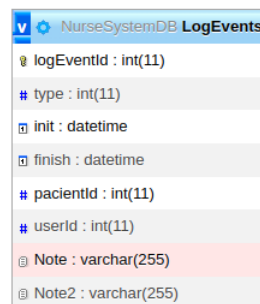
6. Tabla de eventos programados: En esta tabla se cargan las tareas programadas para un paciente. El identificador se incrementa al generarse, el nro de paciente es enviado por el cliente, el campo tipo de evento puede ser **diario**, **mensual** o **diario** y el campo datetime a la hora que se requiere que se realice. El campo nota es donde se coloca la tarea a realizar (Por ejemplo, suministrar X gramos de medicamento Y).



NurseSystemDB EventsTable	
🔑	eventId : int(11)
#	patientId : int(11)
🕒	datetime : timestamp
📄	type : varchar(10)
📄	note : varchar(255)

FIGURA 3.6. Tabla de eventos programados.

7. Tabla de log eventos: En esta tabla se guardan los eventos relacionados con un paciente. El id se asigna al guardarse el evento. El tipo puede ser **tarea programada** o **llamada paciente**. El id del paciente, el id de usuario y la nota se cargan por el usuario que finalizó la acción.



NurseSystemDB LogEvents	
🔑	logEventId : int(11)
#	type : int(11)
🕒	init : datetime
🕒	finish : datetime
#	patientId : int(11)
#	userId : int(11)
📄	Note : varchar(255)
📄	Note2 : varchar(255)

FIGURA 3.7. Tabla de registro de eventos.

8. Tabla de especialidades (*SpecTable*): En esta tabla se las distintas especialidades que poseen todos los enfermeros.
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 una tratamiento(se obtiene de la SpecTable).
11. Tabla de códigos QR (*QRbed*): En esta tabla se almacenan en 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.

3.5. Sistema de gestión

Se diseñó el sistema backend fragmentándolo en tres partes:

- Monitoreo: clases que ayudan a publicar a todos los clientes MQTT los estados del sistema (usuario logeados y estados de habitaciones/pacientes)
- Respuestas al cliente página web: expone una API para que los administradores del sistema puedan incorporar usuarios, pacientes, camas ... observar estadísticas, etc.
- 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.

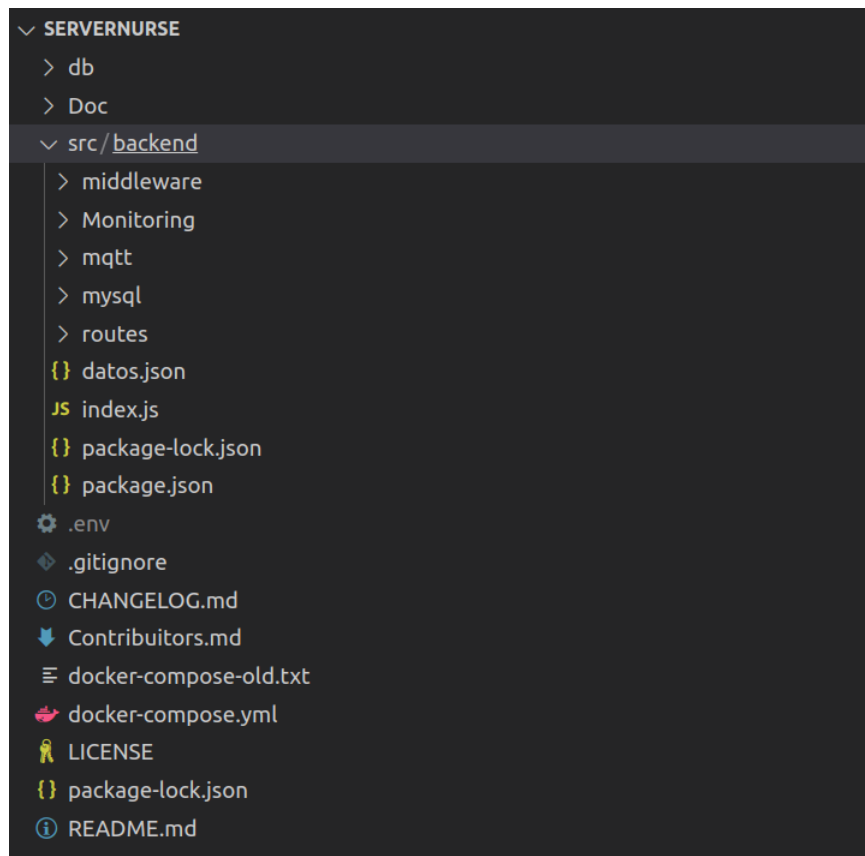


FIGURA 3.8. Estructura de directorio (backend).

La descripción de los contenidos de cada carpeta es:

- middleware: contiene el programa que filtra el acceso a la api desde el la página web.
- Monitoring: contiene clases con información del estado del sistema que se actualiza en tiempo real. 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.
- routes: contiene las rutas de express para acceder a la base de datos desde http.

3.5.1. Descripción de las clases para monitoreo del sistema

Las siguientes son 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:**

Una clase llamada *Bedslist* la cual 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. Puede poseer los siguientes estados:
 - 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:**

Una clase *UserList* la cual contiene una lista de elementos json con información de cada usuario (se instancia al iniciar el backend):

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

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:

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

```

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

```

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:

Esta clase se utiliza para en todo momento saber quien está atendiendo a un paciente.

```

1 constructor () {
2     this.beduserlist=[{ bedId:0 ,userId:0 ,type:0 }];
3
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: 1 llamador, 2 calendario.

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

El archivo 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 del mismo se deriva los mismo a la clase correspondiente que realiza el tratamiento.

```

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 })

```

```

14  })
15  client.subscribe('/Beds/#', function (err) {
16    if (err) {
17      console.log("error:" + err);
18    }
19  })
20
21  //task that will publish beds state each second
22  setInterval(publishBedStates, 10000);
23  //task that will publish users state each second
24  setInterval(publishUserStates, 15000);
25  })

```

CÓDIGO 3.7. Tareas ejecutadas por mqtt.js

Como se puede observar en el código anterior, el sistema recibe mensajería por medio de tres tópicos principales User, Patient y Beds. Por otra parte, se utilizan variables de entorno como ser **MQTT_CONNECTION**, que contiene información de la IP del broker y el puerto del mismo.

Por último, se llama a 2 funciones `publishBedStates` y `publishUserStates` que publican los listados de los estados de camas y usuarios cada cierto tiempo respectivamente.

En general, contenido útil 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 a las clases correspondientes las tareas a realizar.

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.

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.

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"*.
- 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 del trabajo 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. Por definición la página web de configuración solo puede ser accedida por los usuarios administradores, para lograr autenticación el *backend* utiliza las librerías *jsonwebtoken* [33] y *bcrypt* [32].

Se denomina ruteo a la forma que un *endpoint* (punto de acceso, en español) de una aplicación responde a las peticiones de los clientes. En las aplicaciones express, el objeto express posee métodos que realizan las operaciones sobre la base de datos o el sistema.

```

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 ;
16 app.use('/api/treatment',auth.isAuthenticated ,routerPatientSpecTable);

```

CÓDIGO 3.9. Rutas express

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

Luego, la función *auth.isAuthenticated(request,response,next)*, dentro del archivo *./middleware/authentication* filtra el acceso solo a la página web con un usuario administrador logueado.

```

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. Es implementada utilizando el framework Ionic/Angular, con el lenguaje de programación Typescript y las principales áreas de la visualización se observa 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.

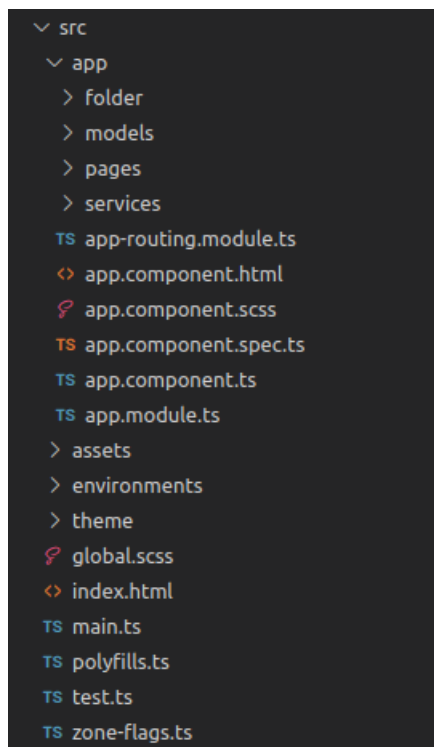


FIGURA 3.10. Estructura de carpetas página web.

Dentro de la carpeta `models` se encuentran las distintas clases que se utilizan para la comunicación con el backend. Las características se presentan 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.
- message: modelo de mensaje almacenado en base de datos(no utilizado).
- 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. Propiedades: numero, numero de especialidad y nombre de especialidad.
- patient: clase que contiene información del paciente (id, nombre, apellido, cama, indice en la tabla de notas, indice en la tabla de médicos)
- patientTreat: clase que relaciona un paciente con un tratamiento. Propiedades: indice, número de paciente, número de tratamiento y nombre de tratamiento.
- qr: relaciona un indice con un texto (se utiliza para guardar / recuperar indices de qr)
- spec: clase que contiene una especialidad/tratamiento. Contiene un indice 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 "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 presentación/trabajo).

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

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

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 el (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 en tiempo real, es necesario configurar la ubicación del *broker* dentro de la solapa MQTT. El monitoreo de camas se observa en la figura 3.11.

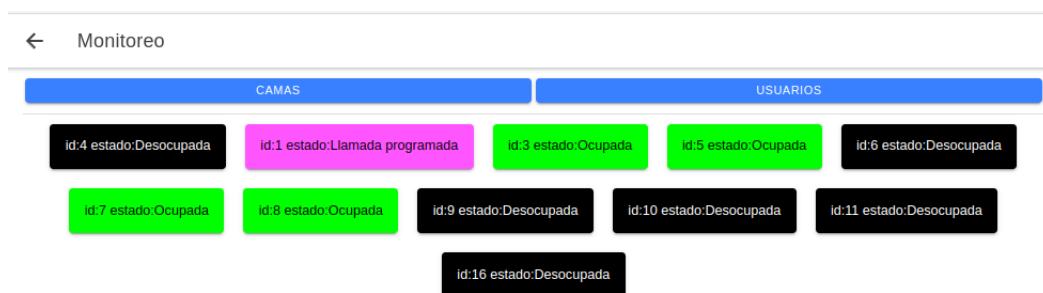


FIGURA 3.11. Monitoreo de camas.

3.6.4. Gestión de tareas programadas

Si se desea generar un nuevo evento programado, se completa el formulario y se presiona enviar.

The screenshot shows a mobile application interface for scheduling tasks. At the top, there is a back arrow and the text "VOLVER". Below this is a form with two fields: "Nota" with the value "colocar vacuna brazo derecho" and "Tipo" with the value "Mensual" and a dropdown arrow. A blue button labeled "ENVIAR" is positioned below the form. Underneath the button is a calendar for "November 2022". The calendar has a header with days of the week (S, M, T, W, T, F, S) and a grid of dates. The date "7" is circled in blue, and the date "9" is highlighted with a solid blue circle. Below the calendar is a "Time" field with a button showing "3:21 PM".

FIGURA 3.12. Tareas programadas.

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

3.6.5. Estadísticas del sistema

En esta función 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.

3.7. Aplicación Móvil

3.7.1. Estructura y organización del software

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.

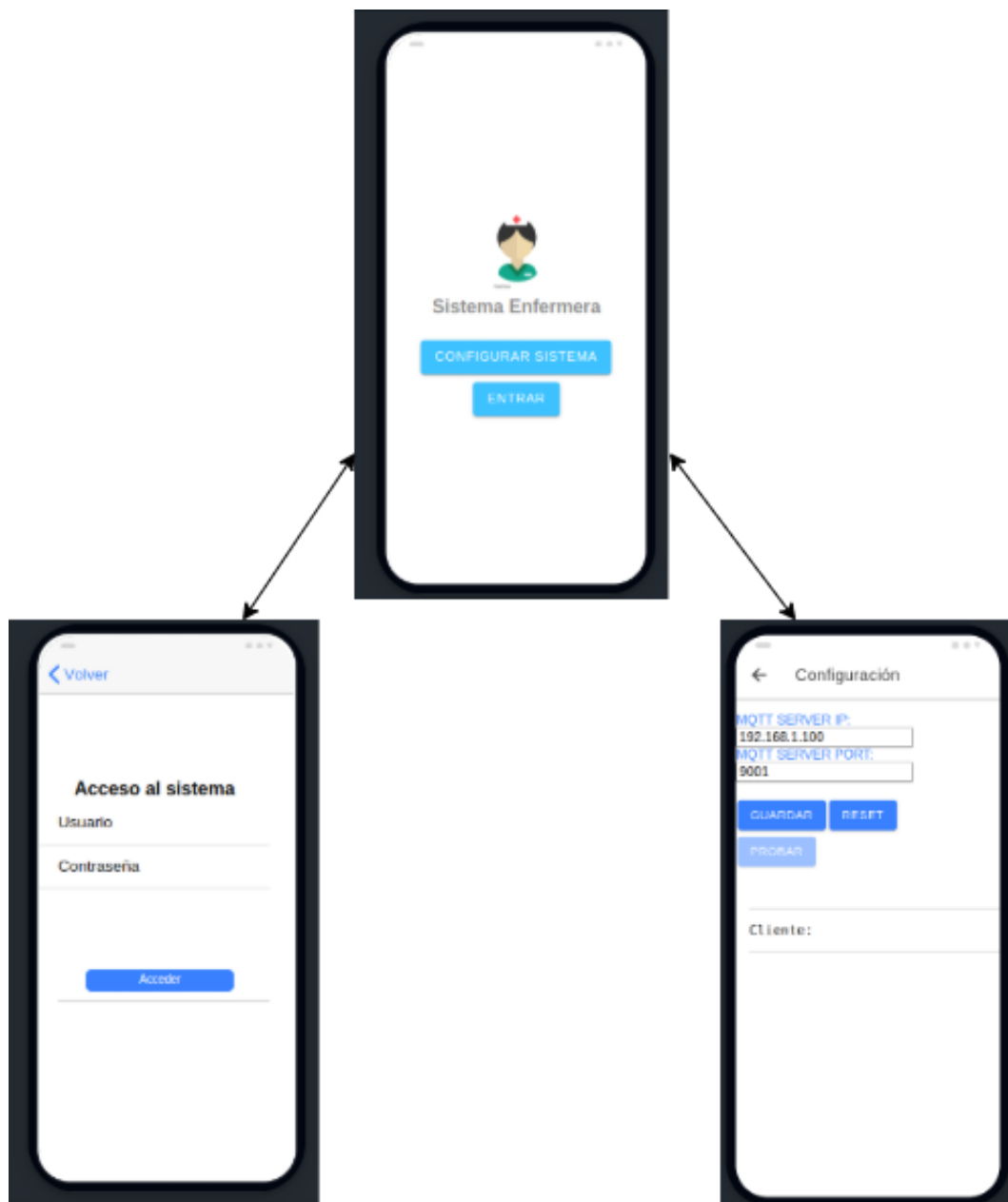


FIGURA 3.13. Pantalla inicial de la aplicación.

3.7.3. Modo administrador

3.7.4. Modo médico

3.7.5. Modo enfermera

3.7.6. Interacción médico-enfermera

3.8. Contraste con los requerimientos

Capítulo 4

Ensayos y resultados

En este capítulo se menciona los ensayos realizados así como

4.1. Generación del sistema de pruebas

La idea de esta sección es explicar cómo se hicieron los ensayos, qué resultados se obtuvieron y analizarlos.

4.2. Pruebas unitarias

4.2.1. 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 [34].

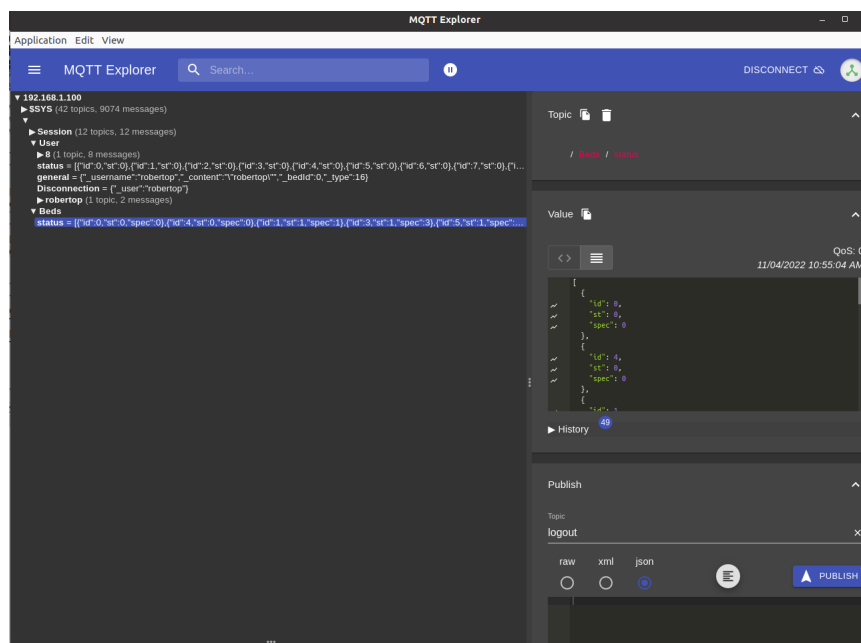


FIGURA 4.1. Imagen de MQTT Explorer.

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

4.2.2. Pruebas unitarias de la API Rest

Para simular el logeo y la consulta a la base de datos se utilizó la herramienta PostMan [35] y la consola de administrador de phpMyAdmin mencionada en 2. En la 4.2 se observa el *token* devuelto por el backend al logearse con las credenciales correspondientes y en 4.3 se observa la respuesta al ingresar una contraseña inadecuada.

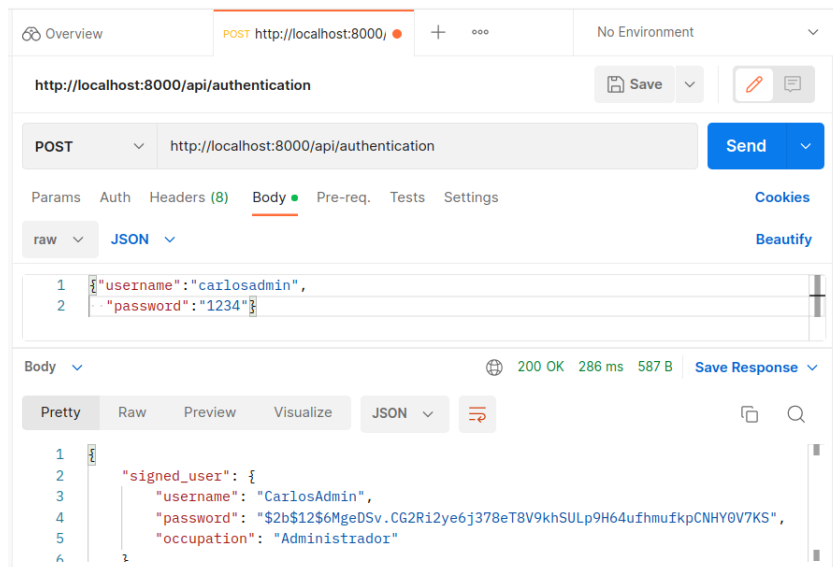


FIGURA 4.2. Logeo en el sistema con Postman.

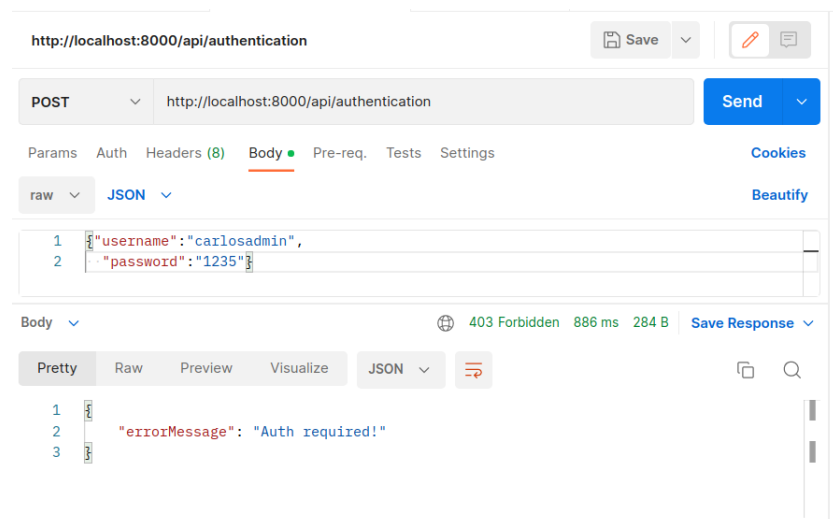


FIGURA 4.3. Rechazo de logeo.

Durante el desarrollo, se deshabilitó la opción de logeo para facilitar las pruebas.

4.3. Integración del sistema

4.3.1. 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 inverso. También se contrató el servicio route 53 que permite customizar las políticas de ruteo. De esta manera, se puede acceder al sistema desde cualquier dispositivo móvil conectado a una red.

4.3.2. Generación/instalación de la aplicación móvil

4.3.3. Equipo simulador de llamadores

4.3.4. Resultados de utilizar el sistema

Capítulo 5

Conclusiones

5.1. Notas sobre el sistema desarrollado

5.1.1. Cumplimiento de requerimientos

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 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 verifico 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.

5.1.2. Cumplimiento de planificación original

No se pudo seguir con la planificación original ya que se presentaron situaciones no contempladas en la misma. Se enumeran a continuación:

- Configuración de la aplicación para ejecutar en un dispositivos móviles: no se contempló el estudio de los permisos y los distintos pasos para poder descargar la aplicación en el dispositivo. Por motivos económicos no se pudo obtener equipos con IOS por lo que se realizaron los ensayos y pruebas con dispositivos Android. Todo ese tiempo no fue contemplado en la planificación.
- Configuración del broker Mosquitto en aplicación móvil: se debió generar una página extra de configuración que permita setear la dirección ip y el puerto del broker Mosquitto de manera de poder utilizar cualquier broker. Además, la información cargada debía almacenarse de alguna manera en el dispositivo por lo que investigó como realizarlo. Todo este tiempo no fue contemplado en la planificación.

5.1.3. Gestión de riesgos

- Riesgo 1: No contar con dispositivo necesario para el testeo de la aplicación.
Resultado: efectivamente, ocurrió este problema y no se pudo mitigar ya que la situación económica/política del país no permitió la compra de los dispositivos necesarios. Queda como trabajo a futuro la validación de la aplicación en una de las plataformas.
- Riesgo 2: Disminución del tiempo disponible para realizar el plan trazado. Efectivamente se presentó el problema (surgieron contratiempos laborales a mitad del proyecto). De todos modos, hubo una influencia mayor de otros problemas como los presentados en la sección inmediatamente anterior que afectaron de mayor manera.
- Riesgo 3: Daño de equipamiento. No se presento este problema.
- Riesgo 4: No contar con colaboradores para la realización de la validación. Se presentó el problema. Se plantea como trabajo a futuro la validación del sistema en un entorno real.
- Riesgo 5: Surgimiento de alternativas de diseño de mejores prestaciones a las propuestas en el trabajo. Efectivamente se presentó el riesgo. Casos de ejemplo:
 1. Se utilizó arreglos para mantener información del estado de los usuarios cuando se recomienda utilizar una base de datos REDIS

2. La base de datos utilizada fue MySQL (debido a ser la única conocida al comienzo del proyecto), pero hubiese sido conveniente utilizar MongoDB, que permite gestionar grandes volúmenes de datos de manera más simple.

5.1.4. Evaluación de las técnicas presentadas durante la carrera

Las herramientas presentadas durante el trayecto de la carrera más utilizadas en este proyecto fueron :

- : Contenedores: muy importante el concepto. Utilizado desde el comienzo del desarrollo.
- : Ionic/Angular: framework muy fácil de aprender. Mucha información y ejemplos para practicar.
- : Express/Node.js: utilizado en todo el backend.
- : Protocolo MQTT: base de este proyecto. Herramienta muy práctica para implementar Internet de las Cosas.
- : *Tokens* JWT : forma de implementar seguridad en el acceso a APIs.
- : Base de datos relacionales: aunque no las utilizaría en otro proyecto similar (según lo expresado en el ítem anterior), resultaron una herramienta útil para gestionar los datos.
- : PostMan: herramienta fundamental para testeo de la API REST del backend.
- : MQTT Explorer: herramienta fundamental para testeo de la API MQTT del backend y para la simulación de los llamadores.
- : ESP32: plataforma versátil que permitió desarrollar un simulador de llamadores básico.
- : Latex: perfecto para generar documentación.
- : AWS: Se utilizó al final del proyecto para poder simular el sistema con mayor cantidad de usuario.
- : Github: todo el desarrollo se guardó versionado en GitHub.

Las herramientas no utilizadas (que hubiesen facilitado el desarrollo de haberlas conocido con anterioridad) :

- Base de datos documentales.
- Base de datos clave-valor.
- Desarrollo TDD o BDD.
- Python(con Django o Flask).
- Modelos de aplicaciones seguras.

5.2. Trabajo futuro

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] Oracle. *What is a Database*. <https://www.oracle.com/database/what-is-database/>. Ene. de 2022. (Visitado 15-10-2022).
- [22] 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).
- [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] Eclipse Foundation. *Eclipse Foundation*. <https://www.eclipse.org/paho/>. Ene. de 2022. (Visitado 30-10-2022).
- [28] BYUNGI(skt t1 byungi). *Rsup MQTT Elegant wrapper for the paho mqtt client*. <https://www.npmjs.com/package/rsup-mqtt/>. Ene. de 2019. (Visitado 30-02-2022).
- [29] JavaTpoint. *Ionic History*. <https://www.javatpoint.com/ionic-history>. Ene. de 2019. (Visitado 30-10-2022).
- [30] OpenJS Foundation. *Electron-Build cross-platform desktop apps with JavaScript, HTML, and CSS*. <https://www.electronjs.org/>. Ene. de 2022. (Visitado 30-10-2022).
- [31] Ionic Open Source. *A cross-platform native runtime for web app*. <https://capacitorjs.com/>. Ene. de 2022. (Visitado 30-10-2022).
- [32] Cardozo et al. *node.bcrypt.js*. <https://www.npmjs.com/package/bcrypt>. Ene. de 2022. (Visitado 30-09-2022).
- [33] Okta Inc. *jsonwebtoken*. <https://www.npmjs.com/package/jsonwebtoken>. Ene. de 2018. (Visitado 30-09-2022).
- [34] 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).
- [35] PostMan. *PostMan : Build APIs together*. <https://www.postman.com/>. Ene. de 2022. (Visitado 30-09-2022).