



CARRERA DE ESPECIALIZACIÓN EN INTERNET DE LAS COSAS

MEMORIA DEL TRABAJO FINAL

Sistema de seguimiento de procesos en rectificadora de motopartes

Autor:

Lic. Pablo Arancibia

Director:

Esp. Ing. Diego Fernandez (FIUBA)

Codirector:

Esp. Ing. Miguel del Valle Camino (FIUBA)

Jurados:

Mg. Ing. Gustavo Zocco (FIUBA)

Esp. Ing. Pedro Rosito (FIUBA)

Esp. Ing. Lionel Gutiérrez (FIUBA)

*Este trabajo fue realizado en la ciudad de Resistencia,
entre enero de 2022 y junio de 2023.*

Resumen

En esta memoria se describe el diseño e implementación de un sistema que permitirá monitorear los estados de trabajos de tornería que se realizan en la empresa Arancibia Rectificaciones.

Para llevar a cabo este trabajo se aplicaron los conocimientos adquiridos en la especialización, especialmente los referidos a desarrollo de aplicaciones, arquitectura de protocolos y sistemas embebidos.

Agradecimientos

A todo el personal docente y administrativo de la Facultad de Ingeniería de la UBA. A mi director, codirector y a los compañeros de la Especialización en Internet de las Cosas que me acompañaron en esta formación profesional.

Índice general

Resumen	I
1. Introducción general	1
1.1. Descripción del sistema	1
1.2. Motivación	2
1.3. Estado del arte	2
1.4. Objetivos y alcance	3
2. Introducción específica	5
2.1. Internet de las cosas	5
2.2. Protocolos de comunicación	5
2.2.1. Protocolo HTTP	5
2.2.2. Protocolo MQTT	6
2.2.3. Broker Eclipse Mosquitto	7
2.3. Bases de datos	7
2.3.1. Bases de datos relacionales	7
2.3.2. Sistema de gestión de base de datos MySQL	7
2.4. Tecnologías backend	8
2.4.1. API RESTFul	8
2.4.2. Node.js	9
2.4.3. Express	9
2.4.4. Sequelize	9
2.4.5. API messenger	10
2.4.6. Portnainer	11
2.5. Tecnologías frontend	12
2.5.1. Ionic framework	12
2.6. Tecnologías de servidor	13
2.6.1. Docker y Docker Compose	13
2.6.2. Servidor web Nginx	13
2.7. Hardware utilizado	14
2.7.1. Raspberry Pi	14
2.7.2. NodeMCU Esp32	14
2.7.3. Identificación por radiofrecuencia	15
2.7.4. Módulo RC522	15
2.7.5. Buzzer sonoro	16
3. Diseño e implementación	17
3.1. Arquitectura general del sistema	17
3.1.1. Funcionamiento	17
3.1.2. Diagrama de bloques	17
3.2. Flujo general del sistema	18
3.2.1. Ingreso de repuesto	18
3.2.2. Cambio de estado	19

3.2.3.	Retiro de repuesto	20
3.3.	Arquitectura de datos	20
3.3.1.	Diagrama de base de datos	20
3.3.2.	Estructura de archivos para ORM	21
3.3.3.	Desarrollo de modelos y tablas	22
3.3.4.	Desarrollo de migraciones	24
3.3.5.	Interacción con la base de datos	25
3.4.	Desarrollo API REST	27
3.4.1.	Patrón de desarrollo	27
3.4.2.	Ruteo de la API	28
3.4.3.	Controladores de la API	29
3.4.4.	Endpoints HTTP	30
3.5.	Comunicación MQTT	31
3.5.1.	Diagrama MQTT	31
3.5.2.	Topics MQTT	31
3.5.3.	Broker MQTT	32
3.5.4.	MQTT en nodos	33
3.5.5.	MQTT en API REST	34
3.6.	API para mensajería	36
3.6.1.	Implementación	36
3.7.	Desarrollo frontend	37
3.7.1.	Patrón de desarrollo	37
3.7.2.	Interfaces de usuario	38
	Nueva orden de trabajo	39
	Listar órdenes de trabajo	43
	Retirar orden de trabajo	45
	Autenticación a API de mensajería	46
3.8.	Arquitectura de servidor	47
3.8.1.	Implementación de contenedores	47
3.9.	Implementación de Hardware	49
4.	Ensayos y resultados	53
4.1.	Pruebas funcionales del hardware	53
5.	Conclusiones	55
5.1.	Conclusiones generales	55
5.2.	Próximos pasos	55
	Bibliografía	57

Índice de figuras

2.1. Arquitectura MQTT publish/subscribe.	6
2.2. Diagrama base de datos relacional.	8
2.3. Arquitectura API Rest.	9
2.4. Arquitectura API Rest.	11
2.5. Arquitectura API Rest.	12
3.1. Diagrama de funciones generales.	17
3.2. Diagrama de bloques y tecnologías del sistema.	18
3.3. Flujo en el ingreso de una nueva orden de trabajo.	19
3.4. Flujo en el cambio de estado de una orden de trabajo.	19
3.5. Flujo en el retiro de una orden de trabajo.	20
3.6. Diagrama de base de datos.	21
3.7. Estructura de carpetas y archivos para ORM Sequelize.	21
3.8. Estructura de carpetas en patrón de desarrollo modular.	27
3.9. Estructura de archivos en carpeta de rutas.	28
3.10. Estructura de archivos en la carpeta <i>controllers</i>	29
3.11. Tabla descriptiva de <i>endpoints</i> o rutas de la API.	30
3.12. Diagrama de funciones MQTT.	31
3.13. Tabla de <i>topics</i> MQTT.	32
3.14. Estructura de carpetas del servicio Eclipse Mosquitto en Docker.	33
3.15. Tabla de detalles de sonidos de nodos.	34
3.17. Flujo en MVC.	38
3.18. Interfaz de usuario para nueva orden de trabajo.	39
3.19. Interfaz de usuario de tipo modal para seleccionar tipo de trabajo.	40
3.20. Interfaz de usuario de tipo modal para seleccionar o buscar cliente.	40
3.21. Interfaz de usuario de tipo modal con el resultado de búsqueda de clientes.	41
3.22. Interfaz de usuario de tipo modal para seleccionar o buscar motocicleta.	41
3.23. Interfaz de usuario de tipo modal con el resultado de búsqueda de motocicletas.	42
3.24. Interfaz de usuario para nueva orden de trabajo.	42
3.25. Interfaz de usuario para listar las órdenes de trabajo.	43
3.26. Confirmar envío de mensaje de texto al cliente.	44
3.27. Confirmar reenvío de mensaje de texto al cliente.	45
3.28. Formulario para retirar orden de trabajo.	46
3.29. Confirmar retiro de orden de trabajo.	46
3.30. Interfaz para autenticación en API de mensajería.	47
3.31. Contenedores de Docker.	49
3.32. Diagrama de red del sistema.	50
3.16. Flujo de código fuente para MQTT en API REST.	51

Índice de tablas

1.1. caption corto	3
------------------------------	---

Dedicado a mis padres.

Capítulo 1

Introducción general

El presente capítulo aborda cuestiones relativas a las etapas en los procesos de rectificación de motopartes en la empresa Arancibia Rectificaciones y las problemáticas de administración que motivaron la implementación del sistema.

1.1. Descripción del sistema

En el taller de rectificaciones de motopartes se realizan diferentes tipos de trabajos relacionados a la tornería [1] de piezas pertenecientes a los motores de motocicletas. Estos trabajos pasan por distintas etapas o estados en los cuales se realizan procesos específicos como encamisado de cilindro [2], cambio de biela [3], balanceo de cigüeñal [4], rectificación de cilindro [5], rectificación de tapa de cilindro [6], entre otros.

Las etapas en general que atraviesa un repuesto desde que ingresa hasta que es retirado de la empresa son:

1. Ingreso de la pieza o repuesto a la empresa:

Un cliente de la empresa se presenta con una pieza para ser reparada, el personal de atención le informa el precio del servicio, fecha de entrega, entre otros datos.

2. Registro de datos del cliente y generación de orden de trabajo:

Luego de aceptadas las condiciones por el cliente, se registran sus datos y se genera una orden de trabajo.

3. Puesta en espera del repuesto:

Se ubica la pieza en el sector de trabajos en espera.

4. Trabajo de mano de obra correspondiente:

Una vez que un empleado de taller de la empresa está libre toma el repuesto para efectuar la mano de obra necesaria.

5. Finalización del trabajo de mano de obra:

Se coloca el repuesto en el sector de finalizados a la espera de ser retirado por el cliente.

6. Entrega del repuesto al cliente:

Cuando el cliente pasa a retirar su pieza, se registran los datos correspondientes y se hace la entrega finalizando así todas las etapas del servicio.

1.2. Motivación

Este trabajo surgió de la necesidad de desarrollar un sistema que permita visualizar en qué etapa se encuentra un repuesto en particular en la empresa, esto permite conocer el estado general de los trabajos, informar a los clientes, tomar decisiones administrativas o técnicas, realizar reportes, etc.

Cuando un cliente se comunica con la empresa para saber si puede pasar a retirar la pieza, el personal de atención tiene que consultar a los empleados de taller el estado en el que se encuentra el trabajo, estos deben dejar de hacer sus tareas por un momento para buscar y responder la consulta, lo cual interrumpe el proceso, genera demoras y consume tiempo. Además, mientras esto sucede, el cliente debe esperar varios minutos.

Por otro lado, resulta complicado cuando el personal de la empresa desea obtener información como: cantidad de trabajos en cada sector, tiempos promedio de proceso, trabajos para ser retirados, cantidad de servicios efectuados en un lapso de tiempo determinado, etc., ya que la manera de obtener estos datos es realizando conteos manuales lo cual resulta improductivo y demanda demasiado tiempo por lo que nunca se realizan estos informes.

Ante este escenario es evidente la necesidad de contar con un sistema informático que posibilite registrar las etapas del proceso y generar la información necesaria para cuando esta sea requerida.

1.3. Estado del arte

En el mercado argentino actualmente se ofrecen diferentes soluciones para resolver problemas relacionados al control de productos ya sea de stock, logística, trazabilidad, transporte, distribución, entre otros. Estas soluciones están basadas en su mayoría en tecnología de lectura de código de barras o de ingresos manuales de datos mediante teclado. Además, existen algunas soluciones de empresas extranjeras, más orientadas al sector industrial, basadas en tecnología RFID [7], en su mayoría por banda UHF [8].

No se encontraron soluciones en el mercado para las necesidades específicas que se plantean en este trabajo. Una de las problemáticas que plantea el escenario para el cual se desarrolla este sistema, es el contexto en el que se realizan los servicios. Las piezas que se reparan están sometidas constantemente a aceites, residuos grasos, polvo, etc. Este escenario hace que se descarte el uso de la tecnología de lectura de código de barras, ya que cualquier lectura a un código sería dificultada por lo mencionado, quedando como mejor opción la utilización de tecnología RFID.

Las soluciones RFID encontradas están planteadas para otro tipo de rubros o industrias, usan generalmente banda UHF y son demasiado costosas para una empresa chica o mediana.

Únicamente se encontró una empresa en Argentina que ofrece servicios algo similares a los que se plantean en este trabajo, Teletrónica S.A. [9]. A continuación se detallan algunas características.

TABLA 1.1. servicios ofrecidos por Telectrónica.

Característica	Telectrónica
Tecnología RFID	Sí
Rubro motores	No
Costo accesible a empresa pequeña	No
Hardware económico	No

La principal característica que imposibilita acceder a este tipo de servicios con empresas argentinas o extranjeras es el alto costo de desarrollo e implementación, debido a que están enfocadas en industrias o empresas grandes que pueden afrontar inversiones de gran escala.

1.4. Objetivos y alcance

El objetivo de este trabajo fue desarrollar un sistema que permita registrar los estados por los que va pasando un repuesto en el taller de tornería de la empresa y poder visualizar esos estados en una plataforma web o móvil.

En primer lugar, se realizó el abordaje de requerimientos de la empresa y se comenzó con la planificación del proyecto. Se continuó con el diseño de la arquitectura tecnológica que se emplearía para el sistema, tanto a nivel de herramientas de desarrollo de software como el hardware a utilizar.

Además, se tuvo en cuenta que los trabajadores de la empresa no debían detener sus tareas para realizar ingresos en teclados ya que esto generaría una interrupción en el flujo de trabajo y el registro de datos en el sistema sería incomodo. Fue por esta razón, principalmente, que se pensó en una tecnología que permita enviar datos a un servidor sin necesidad de manipulación de teclados o dispositivos similares. La tecnología que cumple con este requerimiento es la RFID, la que abordaremos en el siguiente capítulo.

Una vez determinado el diseño y la planificación se comenzaron las investigaciones necesarias, las cuales requirieron una parte importante del tiempo total del trabajo.

El alcance del trabajo se acotó a lo siguiente:

- Desarrollo frontend: aplicación web compatible con móvil.
- Desarrollo backend: API Rest.
- Desarrollo de base de datos.
- Desarrollo e implementación en dispositivos de hardware IoT.
- Desarrollo e implementación de la infraestructura total del sistema, servidor basado en contenedores para servicio web, API Rest, bróker MQTT y base de datos.
- Implementaciones particulares como gabinetes, soportes para tags RFID, entre otros.

Capítulo 2

Introducción específica

En este capítulo se describen las herramientas, tecnologías y hardware que se utilizó para el desarrollo del sistema.

2.1. Internet de las cosas

IoT [10] (*Internet of Things*) o Internet de las cosas en español, es un concepto tecnológico que hace referencia a la interconexión digital de objetos cotidianos a través de internet. Se trata de una red de dispositivos, sensores y otros elementos físicos que se comunican entre sí y con sistemas de información, lo que permite recopilar y procesar datos de manera remota.

El IoT abarca desde dispositivos simples como sensores de temperatura hasta dispositivos más complejos como automóviles autónomos, todos conectados a internet y capaces de intercambiar información. Esta tecnología tiene el potencial de cambiar la forma en la interacción con el mundo físico, mejorando la eficiencia, seguridad y calidad de vida en muchos ámbitos, como la industria, el hogar, la salud, la agricultura, el transporte y más.

La conexión en red de los objetos permite controlarlos de manera remota, obtener información en tiempo real y tomar decisiones basadas en los datos recolectados. Esto permite la creación de sistemas inteligentes que pueden automatizar tareas, reducir costos y mejorar la eficiencia. Además, la interconexión de dispositivos puede generar nuevos modelos de negocio y oportunidades de innovación en diferentes sectores.

2.2. Protocolos de comunicación

2.2.1. Protocolo HTTP

HTTP [11], por sus siglas en inglés: *Hypertext Transfer Protocol*, es un protocolo de tipo cliente-servidor [12], mediante el cual se establece una comunicación enviando peticiones y obteniendo respuestas.

Las características principales de este protocolo son:

- Basado en arquitectura cliente-servidor.
- Además de hipertexto (HTML [13]) se puede utilizar para transmitir otro tipo de documentos como imágenes o vídeos.
- Es un protocolo de capa de aplicación del modelo OSI [14].

- Se transmite principalmente sobre el protocolo TCP [15].

HTTP define un conjunto de métodos de petición, cada uno indica una acción a ejecutar en el servidor. Los más utilizados son:

- GET: se utiliza para recuperar datos.
- POST: sirve principalmente para cargar nuevos datos.
- PATCH: este método aplica modificaciones parciales a los datos existentes.
- PUT: permite reemplazar completamente un registro.
- DELETE: elimina datos específicos.

2.2.2. Protocolo MQTT

MQTT [16] son las siglas de *Message Queuing Telemetry Transport*. Se trata de un protocolo de mensajería liviano para usar en casos donde existen recursos limitados de ancho de banda.

Se transmite sobre protocolo TCP en la arquitectura *publish/subscribe* [17].

Los roles que intervienen en un protocolo MQTT son los siguientes:

- Publicadores: son los que envían los datos.
- Suscriptores: son los que consumen los datos.
- Broker: transmite los mensajes publicados a los suscriptores.

Un cliente puede ser publicador, suscriptor o ambos. El broker es el punto central de la comunicación ya que sin este los mensajes nunca llegarían a destino.

En la figura 2.1 se puede apreciar un ejemplo de comunicación en la arquitectura MQTT.

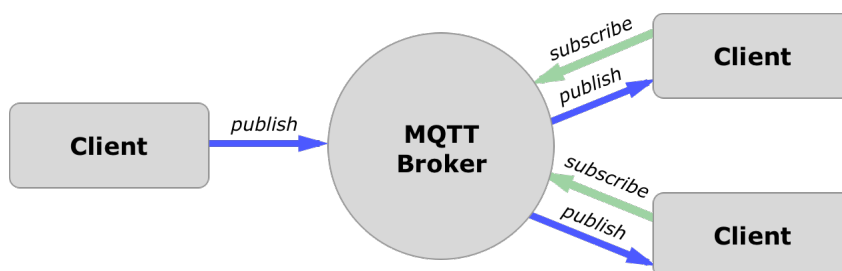


FIGURA 2.1. Arquitectura MQTT publish/subscribe.

La estructura de mensajes en este protocolo se divide en dos: *topics* los cuales son de tipo jerárquicos, utilizando la barra (/) como separador, y *payload* en dónde se incluye el mensaje que se quiere transmitir. Por ejemplo: topic: "nodos/procesos/guardar", payload: "mensaje de ejemplo". Siguiendo este ejemplo un cliente podría suscribirse a ese topic o a una jerarquía más alta y recibir todos los mensajes de los topics que comiencen con nodo/procesos.

2.2.3. Broker Eclipse Mosquitto

Eclipse Mosquitto [18] es un broker MQTT *Open Source* liviano y adecuado para utilizar en todo tipo de dispositivos sobre todo aquellos que cuenten con baja potencia como microcontroladores.

El objetivo de Mosquitto es proporcionar una implementación ligera y de bajo consumo de recursos para permitir la comunicación entre dispositivos IoT en redes con ancho de banda limitado y recursos de memoria.

Mosquitto es compatible con una amplia gama de lenguajes de programación, lo que lo hace fácilmente integrable con diferentes aplicaciones. Además, cuenta con una arquitectura flexible y escalable que permite su implementación en dispositivos con diferentes capacidades de procesamiento y memoria.

Otra característica importante de este broker es su capacidad para manejar conexiones seguras a través del uso de protocolos de seguridad como SSL/TLS y SASL. Esto permite la comunicación segura y cifrada entre dispositivos de IoT en diferentes entornos de red.

2.3. Bases de datos

Las bases de datos son una parte esencial de cualquier aplicación IoT, ya que se utilizan para almacenar y gestionar los datos recopilados por los dispositivos IoT. En esta sección, se describen las tecnologías utilizadas en bases de datos.

2.3.1. Bases de datos relacionales

Las bases de datos relacionales [19] son un tipo de sistema de gestión de bases de datos (SGBD) que se basa en el modelo de datos relacional. Este modelo se utiliza para organizar y almacenar datos en tablas, donde cada tabla representa una entidad o concepto del mundo real y cada fila representa una instancia de esa entidad.

Las tablas se relacionan entre sí mediante claves primarias y claves externas, lo que permite establecer relaciones entre las entidades y realizar consultas complejas que combinan datos de varias tablas. Además, las bases de datos relacionales utilizan el lenguaje de consulta estructurado (SQL o *Structured Query Language*) para interactuar con los datos almacenados en la base de datos.

En la figura 2.2 se puede apreciar un ejemplo de un diagrama de base de datos relacional con sus tablas, filas y relaciones.

2.3.2. Sistema de gestión de base de datos MySQL

MySQL [20] es un sistema de gestión de bases de datos relacional y de código abierto, que utiliza el lenguaje SQL para interactuar con los datos almacenados en la base de datos.

Ofrece una amplia gama de características avanzadas, como soporte para transacciones ACID, índices avanzados, clústeres de alta disponibilidad y replicación. Además, admite múltiples lenguajes de programación, incluyendo C, C++, Python, Java y Ruby, lo que lo hace extremadamente flexible y escalable.

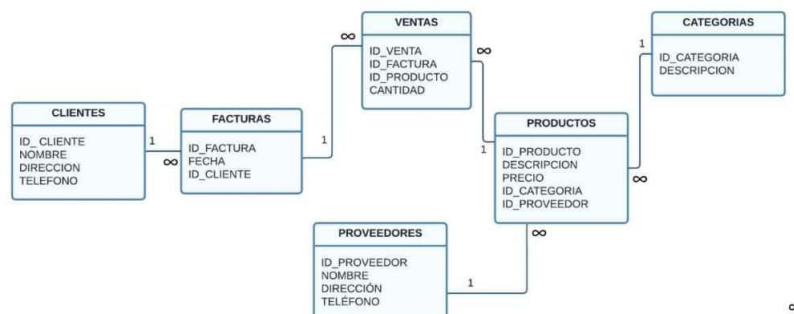


FIGURA 2.2. Diagrama base de datos relacional.

Tiene múltiples capas de seguridad integradas en el sistema, incluyendo autenticación y control de acceso basado en roles.

Debido a su combinación de características avanzadas, flexibilidad y seguridad, MySQL se utiliza ampliamente en aplicaciones de misión crítica y de alta disponibilidad, incluyendo aplicaciones IoT.

2.4. Tecnologías backend

Este tipo de tecnologías se utilizan para desarrollar la lógica de la aplicación y gestionar la comunicación entre el servidor y los dispositivos IoT. En esta sección, se describen las tecnologías backend utilizadas.

2.4.1. API RESTful

Las APIs RESTful [21] (*Representational State Transfer*) son una arquitectura de diseño de aplicaciones web que utiliza el protocolo HTTP para transferir datos. Las API RESTful están diseñadas para ser escalables, flexibles y fáciles de entender para los desarrolladores y los clientes que las consumen.

Están basadas en el concepto de recursos, que son objetos o conjuntos de datos que se pueden acceder a través de una URI (Identificador de recurso uniforme, por sus siglas en inglés). Cada recurso tiene un conjunto de operaciones que se pueden realizar sobre él, como GET (para obtener los datos del recurso), POST (para crear un nuevo recurso), PUT (para actualizar un recurso existente) y DELETE (para eliminar un recurso).

En la figura 2.3 se puede apreciar un ejemplo de una arquitectura API rest, incluyendo la petición o *request* del usuario en formato JSON, el método HTTP y la respuesta o *response* del servidor.

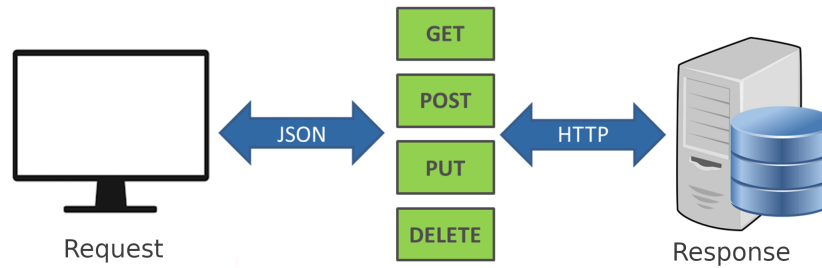


FIGURA 2.3. Arquitectura API Rest.

2.4.2. Node.js

Node.js [22] es un entorno de tiempo de ejecución de JavaScript de código abierto que se ejecuta en el servidor. Fue creado en 2009 con el objetivo de poder desarrollar aplicaciones escalables y de alto rendimiento utilizando el mismo lenguaje de programación para el lado del servidor y del cliente.

Se basa en el motor de JavaScript V8 de Google, lo que lo hace muy rápido y eficiente. Utiliza un modelo de E/S sin bloqueo y orientado a eventos, lo que significa que es capaz de manejar un gran número de solicitudes simultáneas sin bloquear el proceso. Esto lo hace especialmente adecuado para aplicaciones web en tiempo real y aplicaciones de transmisión de medios.

También cuenta con una amplia variedad de módulos y bibliotecas disponibles a través de su gestor de paquetes NPM (*Node Package Manager*). Esto permite aprovechar funcionalidades existentes, desde la creación de APIs RESTful hasta la manipulación de archivos, la comunicación con dispositivos y bases de datos.

2.4.3. Express

Express [23] es un popular framework de Node.js utilizado para la creación de aplicaciones web y APIs RESTful. Fue creado en 2010 y es mantenido por la comunidad de desarrolladores de Node.js.

Proporciona una serie de funcionalidades para simplificar el proceso de creación de aplicaciones web. Entre ellas se incluyen el manejo de rutas, la gestión de middleware, la manipulación de sesiones y cookies, la autenticación de usuarios y la integración con bases de datos.

En Express, el manejo de rutas se realiza mediante la definición estas y los controladores correspondientes. Las rutas son los patrones utilizados para identificar las solicitudes HTTP entrantes que serán atendidas por la aplicación web. Los controladores son las funciones que se encargan de procesar esas solicitudes y generar la respuesta adecuada.

Para definir una ruta, se utiliza el método correspondiente al verbo HTTP que se desea manejar (GET, POST, PUT, DELETE, etc.), seguido de la ruta o patrón que se desea asociar a esa solicitud.

2.4.4. Sequelize

Sequelize [24] es una biblioteca de ORM (*Object-Relational Mapping*) para Node.js que permite trabajar con bases de datos relacionales como MySQL, PostgreSQL,

SQLite, y MSSQL. Facilita el acceso a la base de datos y permite la creación de consultas a través de una interfaz de programación de aplicaciones (API) de alto nivel basada en objetos.

Se utiliza junto a Express y Node.js para simplificar el proceso de acceso y manipulación de datos en bases de datos relacionales. Al utilizar Sequelize, se pueden crear modelos de datos que representen las tablas de la base de datos, lo que permite interactuar con la base de datos utilizando objetos en lugar de consultas SQL.

Alguna de las principales ventajas de utilizar Sequelize son:

- **Abstracción de la base de datos:** proporciona una abstracción de la base de datos que permite a los desarrolladores trabajar con objetos y métodos en lugar de escribir sentencias SQL. Esto facilita el trabajo con la base de datos y reduce la cantidad de código necesario para realizar operaciones CRUD.
- **Seguridad:** proporciona funciones de seguridad incorporadas, como la prevención de inyecciones SQL y la validación de datos de entrada. Esto ayuda a reducir los riesgos de seguridad y garantiza que los datos almacenados en la base de datos sean confiables.
- **Migraciones de base de datos:** proporciona una forma fácil de administrar las migraciones de base de datos. Se pueden definir cambios en la estructura de la base de datos utilizando migraciones y aplicarlas en el orden correcto, lo que ayuda a garantizar que la base de datos esté actualizada y que los cambios se realicen de manera controlada.
- **Soporte para múltiples bases de datos:** admite múltiples bases de datos, lo que significa que se puede trabajar con diferentes bases de datos sin tener que aprender una nueva sintaxis para cada una. Esto hace que sea más fácil trabajar con diferentes bases de datos y reducir el tiempo de aprendizaje.
- **Integración con Express:** se integra bien con Express, lo que permite trabajar con ambos de manera conjunta. Esto hace que sea más fácil construir aplicaciones web y manejar las operaciones de la base de datos al mismo tiempo.

2.4.5. API messenger

Para el envío de mensajes por la aplicación *WhatsApp* [25] se utilizó una API adicional [26], denominada en este trabajo *API Messenger* a fines de distinguirla de la API de *backend* desarrollada.

Esta API presenta un desarrollo en un patrón modular de tipo DDD (*Domain-Driven Design*). La principal característica en este caso es la creación de capas de infraestructura, servicios y aplicaciones. Esto servirá en caso de cambiar el proveedor de mensajería *WhatsApp* o agregar otro proveedor en paralelo, lo cual hace el sistema escalable y adaptable a diferentes necesidades y escenarios.

Para los requerimientos de este trabajo se personalizó el código fuente de la *API Messenger* a fines de agregar características adicionales, se detalla esto en el capítulo 3.6.

2.4.6. Portainer

Portainer [27] es una herramienta de administración de contenedores Docker [28] que proporciona una interfaz web para administrar los contenedores y clústeres Docker.

Proporciona una variedad de características útiles, incluyendo:

- Una interfaz web intuitiva y fácil de usar para administrar contenedores Docker, imágenes, volúmenes y redes.
- La posibilidad de crear y configurar contenedores a través de una interfaz gráfica de usuario (GUI).
- La gestión de múltiples *hosts* desde una única interfaz de usuario, lo que permite administrar varios clústeres o instancias de Docker en diferentes servidores.
- La capacidad de ver y administrar fácilmente el estado de los contenedores, así como el uso de recursos y las estadísticas de rendimiento.
- La gestión de usuarios y roles de acceso, lo que permite controlar el acceso y la autorización de los usuarios.
- La posibilidad de crear y gestionar *stacks* y servicios de Docker, lo que permite crear aplicaciones complejas y multi-contenedor de manera fácil y rápida.

Para este trabajo se utilizó la herramienta solamente para el monitoreo de los contenedores y el control del uso de recursos de sistema de los mismos.

En las figuras 2.4 y 2.5 podemos observar la interfaz gráfica de la herramienta.

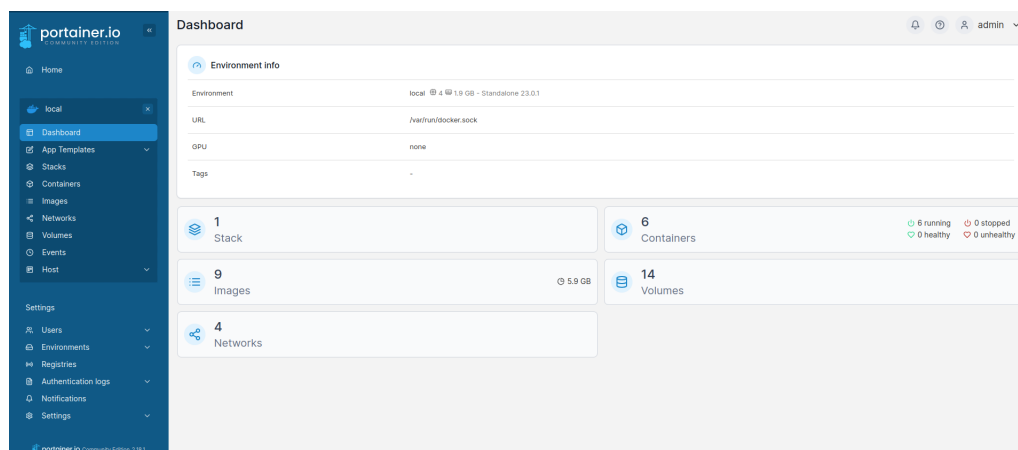


FIGURA 2.4. Arquitectura API Rest.

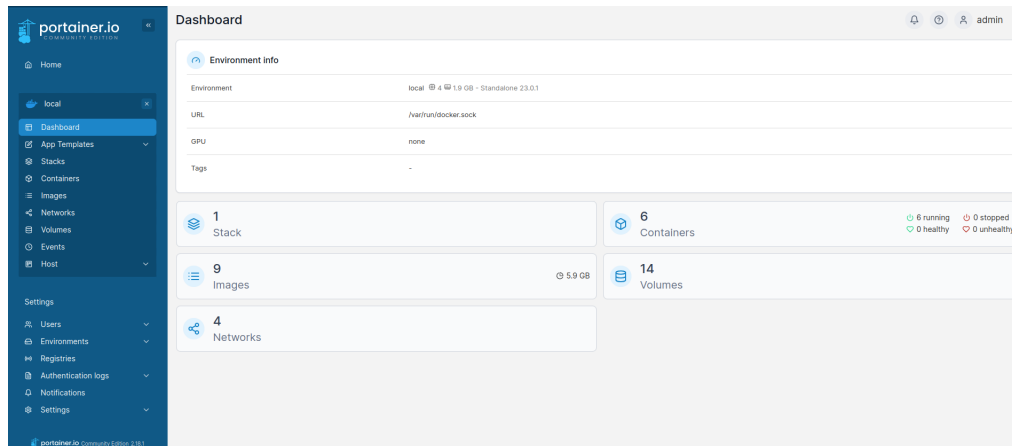


FIGURA 2.5. Arquitectura API Rest.

2.5. Tecnologías frontend

Las tecnologías frontend son aquellas que se utilizan para crear la parte visual y de interacción de una aplicación web o móvil. Estas tecnologías se enfocan en la presentación y manipulación de la interfaz de usuario, en la interacción con el usuario final y actúa como intermediario entre el usuario final y el backend de la aplicación.

Algunas de las tecnologías frontend más comunes son:

- HTML [13] (*HyperText Markup Language*): es el lenguaje de marcado que se utiliza para estructurar y dar formato al contenido de una página web.
- CSS [29] (*Cascading Style Sheets*): es el lenguaje utilizado para dar estilo y diseño a una página web, permitiendo la personalización de fuentes, colores, márgenes, tamaños y otros aspectos visuales.
- JavaScript [30]: es un lenguaje de programación que se utiliza para hacer que la página web sea interactiva y dinámica, permitiendo la manipulación de elementos del DOM [31] (*Document Object Model*), eventos, animaciones y otras acciones en el lado del cliente.
- Frameworks de JavaScript: son bibliotecas que permiten simplificar el desarrollo de aplicaciones web, proporcionando funcionalidades predefinidas y estructuras de organización.
- Bibliotecas de diseño: son herramientas que ofrecen componentes visuales predefinidos y estilos de diseño que permiten crear páginas web con un aspecto más profesional y elegante.
- Herramientas de gestión de paquetes: son programas que permiten gestionar las dependencias de los proyectos y mantener actualizadas las librerías utilizadas en el desarrollo.

2.5.1. Ionic framework

Ionic [32] es un framework de desarrollo de aplicaciones móviles híbridas basado en tecnologías web como HTML, CSS y JavaScript. Permite crear aplicaciones

móviles para iOS, Android y la web utilizando un conjunto de herramientas y bibliotecas predefinidas.

Ofrece una gran cantidad de componentes visuales, animaciones y funcionalidades para crear aplicaciones móviles con una apariencia y experiencia de usuario nativa, similar a las aplicaciones desarrolladas con tecnologías nativas como Java para Android o Swift para iOS. Además, permite la integración con otras tecnologías como Angular [33] y React [34].

El uso de tecnologías web permite el desarrollo de aplicaciones móviles de forma más rápida y sencilla que las aplicaciones nativas, ya que se utiliza un único código base que se puede adaptar para cada plataforma. Además, ofrece una gran cantidad de herramientas y servicios para simplificar el proceso de desarrollo, como Ionic Native (para el acceso a las características nativas del dispositivo) y Ionic Appflow (para la implementación continua y la gestión de versiones).

2.6. Tecnologías de servidor

2.6.1. Docker y Docker Compose

Docker [28] es una plataforma de software libre que se utiliza para desarrollar, implementar y ejecutar aplicaciones en contenedores. Los contenedores son una forma de virtualización que permiten a los desarrolladores empaquetar una aplicación y todas sus dependencias en una imagen de contenedor, que se puede ejecutar en cualquier entorno que tenga Docker instalado. Docker facilita la implementación de aplicaciones en diferentes plataformas, desde servidores locales hasta nubes públicas.

Docker Compose [35] es una herramienta de Docker que se utiliza para definir y ejecutar aplicaciones de múltiples contenedores. Permite definir todos los servicios necesarios para una aplicación en un archivo de configuración YAML, lo que facilita la implementación y el mantenimiento de la aplicación. Docker Compose puede iniciar todos los contenedores necesarios para la aplicación con un solo comando, lo que ahorra tiempo y reduce los errores.

2.6.2. Servidor web Nginx

Un servidor web es un software que procesa solicitudes HTTP de clientes y responde con contenido estático o dinámico. Los servidores web se utilizan para alojar y servir sitios web y aplicaciones web. Un servidor web típicamente aloja varios sitios web y puede gestionar múltiples solicitudes HTTP simultáneamente.

Nginx [36] es un servidor web de código abierto que se utiliza para alojar y servir sitios web y aplicaciones web. Nginx es conocido por su alta escalabilidad, rendimiento y capacidad de manejar múltiples solicitudes HTTP simultáneamente. Es un servidor web ligero y rápido que se puede utilizar como un proxy inverso para distribuir la carga de trabajo a diferentes servidores y balancear la carga de tráfico.

2.7. Hardware utilizado

2.7.1. Raspberry Pi

La Raspberry Pi [37] es una pequeña computadora de placa única (*Simple Board Computer*, por sus siglas en inglés) diseñada para ser utilizada en proyectos de tecnología, educación y prototipos de hardware. Fue desarrollada por la Fundación Raspberry Pi, una organización benéfica con sede en el Reino Unido.

Principales características del modelo Raspberry Pi 4 seleccionado para este trabajo:

- Procesador Broadcom BCM2711 de cuatro núcleos ARM Cortex-A72 a 1,5 GHz.
- Procesador gráfico VideoCore VI con soporte para OpenGL ES 3.x.
- 8 GB de memoria RAM LPDDR4-3200.
- Bluetooth 5.0.
- Wi-Fi de doble banda 802.11ac.
- Gigabit Ethernet.

Puertos:

- Dos puertos micro-HDMI que pueden soportar dos pantallas con resolución de hasta 4K a 60 fps.
- Dos puertos USB 3.0.
- Dos puertos USB 2.0
- Un puerto GPIO de 40 pines
- Un puerto CSI para la conexión de una cámara
- Un puerto DSI para la conexión de una pantalla táctil
- Un puerto de audio de 3,5 mm.

La placa es compatible con diferentes sistemas operativos, incluyendo Raspberry Pi OS (anteriormente llamado Raspbian), Ubuntu, Windows 10 IoT Core y otros sistemas operativos basados en Linux. También es compatible con diferentes lenguajes de programación como Python, C++, Java y más, lo que la hace ideal para proyectos de IoT, robótica, automatización del hogar, entre otros.

2.7.2. NodeMCU Esp32

La NodeMCU ESP32 [38] es una placa de desarrollo basada en el microcontrolador ESP32, que es ampliamente utilizado en proyectos de Internet de las cosas (IoT).

Detalles técnicos del modelo NodeMCU ESP32 WROOM 32 seleccionado para este trabajo:

- Microcontrolador ESP32 de doble núcleo con una velocidad de reloj de hasta 240 MHz y 512 kB de memoria RAM.

- 4 MB de memoria flash integrada para almacenamiento de programas y datos.
- Conectividad inalámbrica Wi-Fi 802.11 b/g/n y Bluetooth 4.2 BLE.
- Interfaz de programación USB integrada para programar la placa y proporcionar una conexión de depuración.
- GPIO de 30 pines para la conexión de sensores, actuadores y otros dispositivos periféricos.
- Interfaces I2C, SPI, UART, PWM y ADC integradas.
- Soporte para el entorno de programación Arduino, así como para MicroPython y Lua.
- Compatible con una amplia gama de bibliotecas y herramientas de desarrollo de código abierto.

2.7.3. Identificación por radiofrecuencia

RFID [39] son las siglas en inglés de *Radio Frequency Identification* o identificación por radiofrecuencia en español. Es una tecnología de identificación automática que utiliza ondas de radio para leer y capturar información almacenada en etiquetas o *tags* RFID.

Esta tecnología consta de tres componentes básicos: el tag RFID que comunmente es una tarjeta o llavero pequeño, el lector RFID y un sistema informático que gestiona la información capturada por el lector.

Los tags RFID contienen antenas y circuitos integrados que permiten la comunicación inalámbrica con los lectores, los cuales envían señales de radio para alimentar y activar los tags, y recibir la información almacenada en ellos. Pueden ser pasivos, activos o semi-activos, dependiendo de si necesitan o no una fuente de alimentación externa.

2.7.4. Módulo RC522

El RFID RC522 [40] es un módulo de lectura-escritura RFID que utiliza la tecnología de comunicación de campo cercano (NFC) para la identificación y el intercambio de datos de manera inalámbrica. Entre sus principales características se encuentran:

- Soporte para frecuencias de operación de 13,56 MHz.
- Comunicación mediante el protocolo SPI (*Serial Peripheral Interface*).
- Capacidad para leer y escribir etiquetas RFID de tipo MIFARE, que son ampliamente utilizadas en sistemas de acceso, control de inventario, pago sin contacto, entre otros.
- Funciones de autenticación y encriptación para mayor seguridad en la transmisión de datos.
- Bajo consumo de energía y fácil integración con microcontroladores y otros sistemas embebidos.

2.7.5. Buzzer sonoro

El módulo buzzer pasivo KY006 [41] es un pequeño dispositivo electrónico que se utiliza para producir sonidos audibles en una variedad de proyectos electrónicos. Se conecta a la placa de control mediante tres pines.

Entre las características técnicas del módulo KY006 se encuentran:

- Tensión de funcionamiento: 5 V DC.
- Corriente de funcionamiento: <25 mA.
- Tipo de sonido: continuo y monótono.
- Frecuencia de resonancia: 2300 ± 300 Hz.
- SPL (nivel de presión sonora): >85 dB a 10 cm de distancia.
- Diámetro: 30 mm.
- Altura: 7,5 mm.
- Peso: 4 gramos.

Capítulo 3

Diseño e implementación

3.1. Arquitectura general del sistema

3.1.1. Funcionamiento

Como se puede observar en la figura 3.1, el sistema está compuesto por nodos los cuales se utilizan para la lectura de tarjetas RFID asignadas a los repuestos de los clientes. Los datos se transmiten en una red lan local, se procesan y almacenan en un servidor con base de datos y son consultados desde la aplicación web en las terminales (PC o *smartphone*).

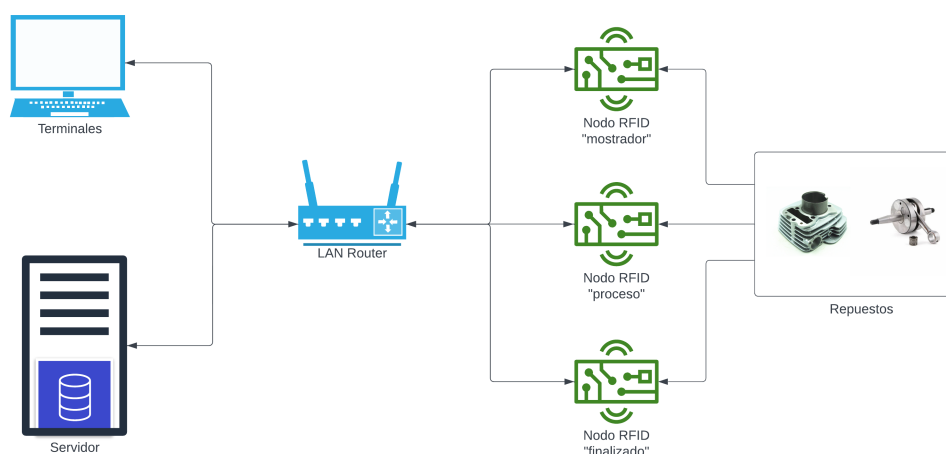


FIGURA 3.1. Diagrama de funciones generales.

3.1.2. Diagrama de bloques

En la figura 3.2 se representa el patrón de modelo conceptual empleado y las tecnologías que se utilizan en cada capa. Se implementó un modelo de 4 capas:

- Capa de percepción.
- Capa de transporte.
- Capa de procesamiento.
- Capa de aplicación.

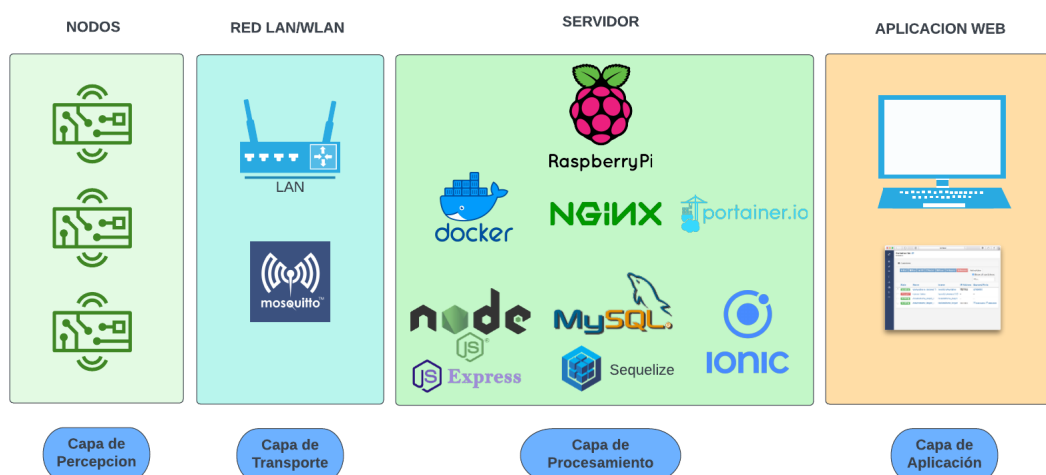


FIGURA 3.2. Diagrama de bloques y tecnologías del sistema.

En la capa de percepción se utilizan los nodos esp32 con lector RFID con los cuales se realiza la lectura de las tarjetas correspondientes. Una vez realizada la lectura, los datos son enviados por medio del protocolo MQTT y a través de la red LAN en la capa de transporte.

La capa de transporte es la encargada de transmitir los mensajes por medio de los protocolos MQTT y HTTP. El broker *Eclipse Mosquitto* distribuye los mensajes publicados a los subscriptores para su procesamiento.

En la capa de procesamiento se realiza la lógica del backend, la base de datos y el frontend. Se implementó un servidor central en una Raspberry Pi 4 con todos los servicios. Cada uno de los servicios está desarrollado de manera individual y fue montado en su propio contenedor de Docker. Todos los contenedores se despliegan utilizando Docker Compose.

Por último, la capa de aplicación otorga el acceso web desarrollado en el framework *Ionic* para el ingreso, registro, administración y egreso de las ordenes de trabajo y también se utiliza el portal de administración de *Portainer* para el monitoreo completo de Docker y del servidor.

3.2. Flujo general del sistema

En esta sección se explica el flujo total de las funciones del sistema desde que el usuario ingresa una nueva orden de trabajo hasta que el producto es retirado por el cliente.

Las comunicaciones y el envío de datos entre los distintos módulos del sistema se realizan en los protocolos HTTP, MQTT y MySQL. Se detallará en cada caso el protocolo utilizado.

3.2.1. Ingreso de repuesto

En la figura 3.3 se puede observar todos los módulos del sistema, el flujo de datos y protocolos que intervienen cuando un usuario carga una nueva orden de trabajo en el sistema.

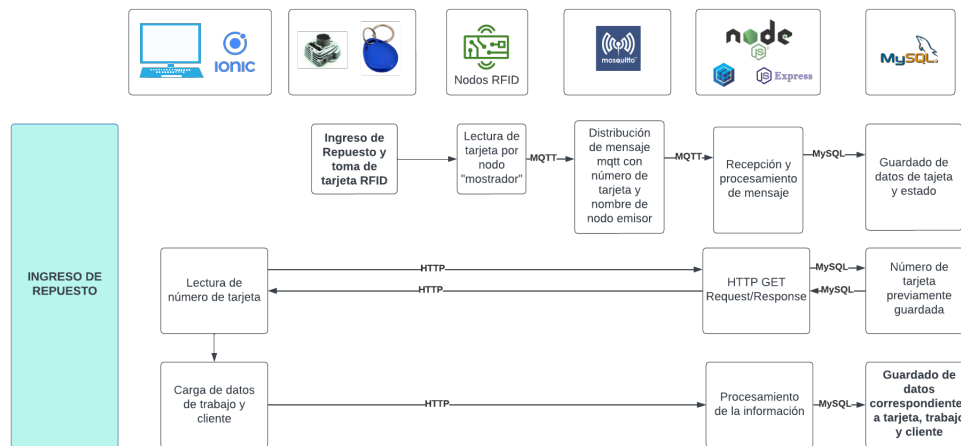


FIGURA 3.3. Flujo en el ingreso de una nueva orden de trabajo.

El flujo inicia cuando el usuario recibe el repuesto y selecciona una tarjeta RFID libre para usarla con ese repuesto. El usuario pasa la tarjeta por el nodo ubicado en la recepción y de esta manera se registra el número de tarjeta para ser utilizada. Luego el usuario carga los datos en la aplicación web, dónde ya está asignada la tarjeta previamente leída, confirma los datos y estos son guardados en la base de datos. La nueva orden de trabajo tiene su número de tarjeta, los datos del cliente y el estado por defecto *en espera*.

3.2.2. Cambio de estado

En la figura 3.4 podemos observar el flujo completo para un cambio de estado de una orden de trabajo.

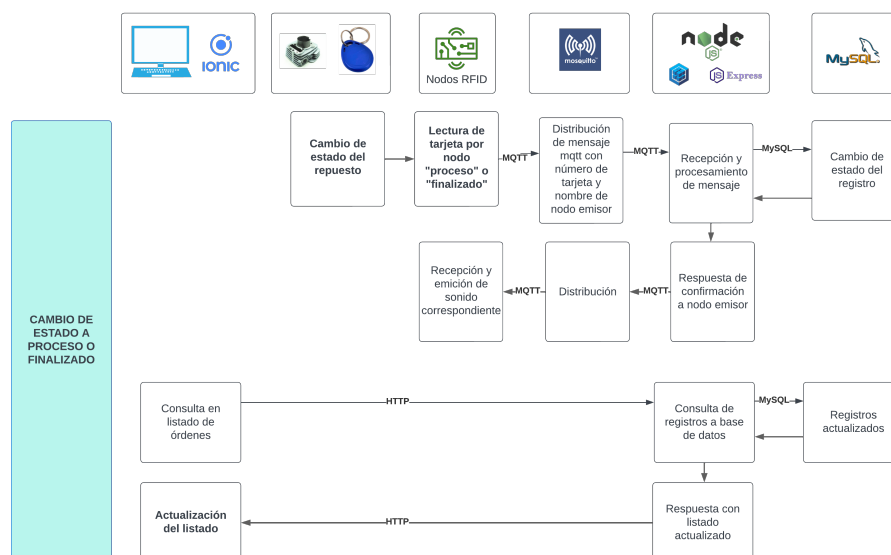


FIGURA 3.4. Flujo en el cambio de estado de una orden de trabajo.

El flujo inicia cuando un trabajador inicia o finaliza una óden, cambiando el estado de la misma a *en proceso* o *finalizada*. Para actualizar el estado de la orden debe pasar la tarjeta RFID asociada al repuesto por el nodo correspondiente, de

esta manera se transmite la información por protocolo MQTT hasta la API de *backend*, esta última realiza la actualización en la base de datos y responde al nodo por medio de MQTT informando si la operación tuvo éxito. El nodo emitirá un sonido que informará al usuario del resultado obtenido (tabla de sonidos en capítulo 3.5.4). Al mismo tiempo el usuario de la aplicación web verá reflejada la actualización del estado de la orden correspondiente en el listado de órdenes, ya que se está constantemente consultando por actualizaciones a la API por medio del protocolo HTTP utilizando el patrón observador de IONIC (detallado en el capítulo 3.7.2).

3.2.3. Retiro de repuesto

En la figura 3.5 podemos observar el flujo completo en el retiro de un repuesto.

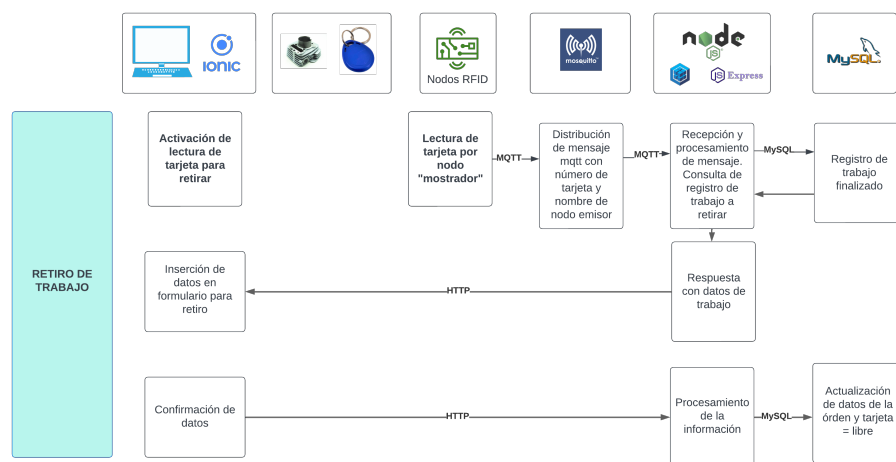


FIGURA 3.5. Flujo en el retiro de una orden de trabajo.

El flujo inicia cuando el usuario de la aplicación web realiza una activación de lectura de tarjeta en la pantalla de retiro. Esto ocasiona que la aplicación active un *modo escucha* hacia la API, esperando a que el usuario pase la tarjeta asociada al repuesto por el nodo *mostrador*. Cuando esto sucede la API recibe por MQTT el número de tarjeta y mediante este recupera los datos de la orden correspondiente en la base de datos, luego devuelve los datos obtenidos a la aplicación web por medio del protocolo HTTP. La aplicación inserta todos estos datos en el formulario de retiro y, cuando el usuario confirma la operación, se realiza una petición HTTP con el método PUT para actualizar los datos de los registros. La tarjeta utilizada queda en estado *libre* para ser reutilizada en una nueva orden de trabajo.

3.3. Arquitectura de datos

En la presente sección se desarrollará la arquitectura en la base de datos MySQL, el diseño y la implementación de las funciones con el ORM Sequelize.

3.3.1. Diagrama de base de datos

En la figura 3.6 se representa un diagrama UML de la base de datos con sus tablas y relaciones.

Para realizar el diseño de la base de datos se realizó un análisis de los requerimientos y los casos de usos o historias de usuario, a partir de esto se inició el diagrama con las tablas principales y sus relaciones, a medida que se avanzaba en el desarrollo de la API y del sistema se fueron añadiendo nuevas tablas y relaciones según las necesidades.

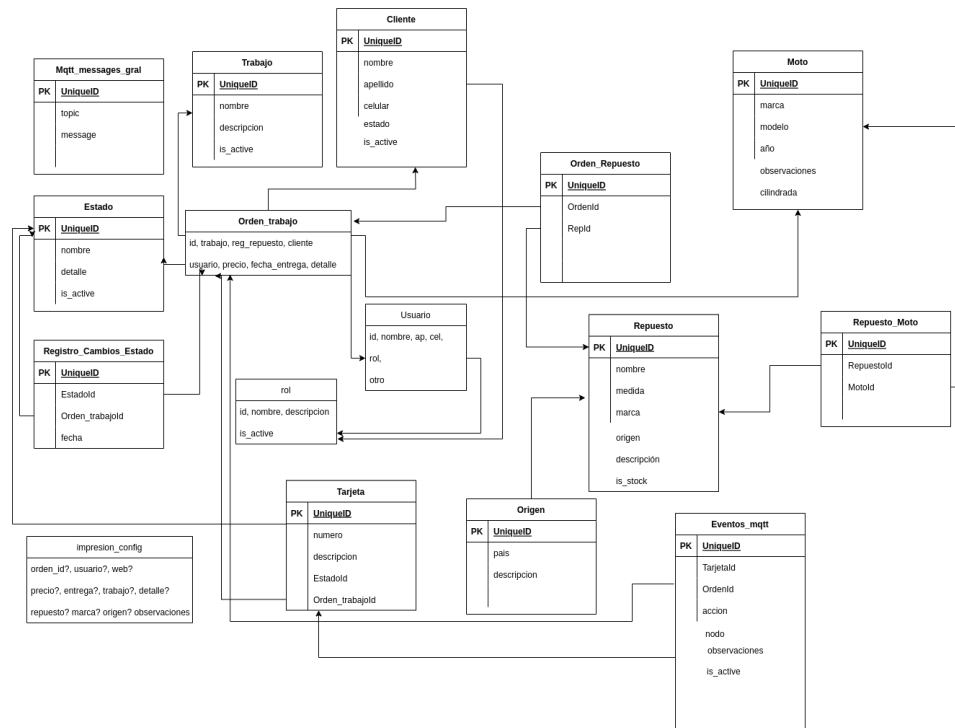


FIGURA 3.6. Diagrama de base de datos.

3.3.2. Estructura de archivos para ORM

Para el desarrollo de la base de datos se utilizó el ORM Sequelize. Como se menciona en el capítulo 2.4.4, existen múltiples ventajas al utilizar un ORM en vez de directamente programar la base de datos en lenguaje SQL, se tuvieron en cuenta esas ventajas a la hora de optar por realizar el desarrollo con este ORM.

La estructura de carpetas y archivos está definida previamente por el ORM pudiendo realizar algunas personalizaciones.

En la figura 3.7 podemos ver la estructura implementada.

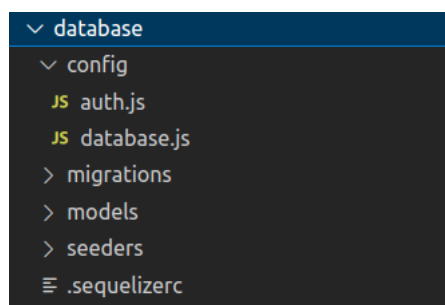


FIGURA 3.7. Estructura de carpetas y archivos para ORM Sequelize.

Dentro de la carpeta *config* se definen los archivos de configuración de Sequelize y el acceso a la base de datos MySQL.

En la carpeta *migrations*, *models* y *seeders* se encuentran todos los archivos Javascript para las migraciones, los modelos y semillas, estos archivos se detallarán en las próximas secciones.

El archivo *.sequelizerc* sirve para definir la ubicación de los modelos, migraciones, semillas y otros archivos generados por Sequelize.

3.3.3. Desarrollo de modelos y tablas

A continuación se presentan algunos fragmentos de código fuente empleados para el desarrollo de la base de datos, siguiendo esta misma modalidad se realizaron todas las tablas de la base de datos, sus modificaciones y sus respectivas migraciones.

El siguiente código se ejecuta en la terminal de linux para crear el archivo para el modelo y el archivo para la migración de la tabla para "Orden de trabajo". En el código se define el nombre del modelo y un atributo *id*.

```
1 npx sequelize-cli model:generate --name mqtt_messages_gral --attributes
  id:integer
```

CÓDIGO 3.1. Código CLI para crear modelo y migración en Sequelize.

El siguiente código se emplea para definir el modelo completo de la tabla:

```
1 'use strict';
2 const {Model} = require('sequelize');
3 module.exports = (sequelize, DataTypes) => {
4   class OrdenTrabajo extends Model {
5
6     static associate(models) {
7       //OrdenTrabajo.X pertenece a X
8       OrdenTrabajo.belongsTo(models.Trabajo)
9       OrdenTrabajo.belongsTo(models.Estado)
10      OrdenTrabajo.belongsTo(models.Cliente)
11      OrdenTrabajo.belongsTo(models.Usuario)
12      OrdenTrabajo.belongsTo(models.Moto)
13
14      //OrdenTrabajo tiene ids en la tabla Eventos_mqtt
15      OrdenTrabajo.hasMany(models.Eventos_mqtt);
16
17      //OrdenTrabajo tiene id en la tabla Tarjeta
18      OrdenTrabajo.hasMany(models.Tarjeta);
19
20      // Orden_trabajo tiene muchos Cambios de Estado N:M
21      OrdenTrabajo.belongsToMany(models.Estado, {
22        through: 'Registro_cambios_estado'
23      })
24
25      // Orden de trabajo tiene muchos Repuestos N:M
26      OrdenTrabajo.belongsToMany(models.Repuesto, {
27        through: 'Orden_Repuesto'
28      })
29
30
31   }
32 }
```

```

33  OrdenTrabajo.init({
34    id: {
35      allowNull: false,
36      autoIncrement: true,
37      primaryKey: true,
38      type: DataTypes.INTEGER
39    },
40    precio: {
41      type: DataTypes.INTEGER
42    },
43    entrega: {
44      type: DataTypes.INTEGER
45    },
46    fecha_entrega_estimada: {
47      type: DataTypes.DATE
48    },
49    detalle: {
50      type: DataTypes.TEXT('long')
51    },
52    tarjeta: {
53      type: DataTypes.STRING
54    },
55    ordenPapel: {
56      type: DataTypes.STRING
57    },
58    informado: {
59      type: DataTypes.BOOLEAN
60    },
61    is_active: {
62      type: DataTypes.BOOLEAN
63    },
64    createdAt: {
65      allowNull: false,
66      type: DataTypes.DATE
67    },
68    updatedAt: {
69      allowNull: false,
70      type: DataTypes.DATE
71    }
72  }, {
73    sequelize,
74    modelName: 'OrdenTrabajo',
75    tableName: 'OrdenTrabajo',
76  });
77  return OrdenTrabajo;
78  };

```

CÓDIGO 3.2. Código para un modelo en Sequelize.

En las líneas 1 a la 4 se realizan las declaraciones de la librería y el nombre del modelo, luego en las líneas 6 a la 31 se definen las relaciones que tendrá el modelo, los tipos de relaciones pueden ser 1 a 1, 1 a muchos o muchos a muchos. Luego a partir de la línea 33 se definen los campos de la tabla o modelo, se definen también los atributos de los campos y el tipo de datos. En las líneas 74 y 75 se define el nombre personalizado que tendrá la tabla en la base de datos y el nombre del modelo que reconocerá el ORM. Por último en la línea 77 se retorna la Clase creada, de esta manera podrá ser utilizada en otras partes del código fuente.

3.3.4. Desarrollo de migraciones

Como se menciona en la sección anterior, el archivo con la estructura inicial de migraciones se crea a través del CLI de Sequelize al momento de crear el modelo para una tabla, luego hay que personalizar esta estructura para que quede igual al modelo definido previamente.

A continuación se representa el código *javascript* para la migración del modelo "Orden de trabajo":

```

1 'use strict';
2 module.exports = {
3   async up(queryInterface, Sequelize) {
4     await queryInterface.createTable('OrdenTrabajo', {
5       id: {
6         allowNull: false,
7         autoIncrement: true,
8         primaryKey: true,
9         type: Sequelize.INTEGER
10      },
11      nombre: {
12        type: Sequelize.STRING
13      },
14      precio: {
15        type: Sequelize.INTEGER
16      },
17      entrega: {
18        type: Sequelize.INTEGER
19      },
20      fecha_entrega_estimada: {
21        type: Sequelize.DATE
22      },
23      detalle: {
24        type: Sequelize.TEXT('long')
25      },
26      tarjeta: {
27        type: Sequelize.STRING
28      },
29      ordenPapel: {
30        type: Sequelize.STRING
31      },
32      informado: {
33        type: Sequelize.BOOLEAN,
34        defaultValue: false
35      },
36      TrabajoId: {
37        type: Sequelize.INTEGER,
38        references: { model: 'Trabajo', key: 'id' }
39      },
40      EstadoId: {
41        type: Sequelize.INTEGER,
42        references: { model: 'Estado', key: 'id' }
43      },
44      ClienteId: {
45        type: Sequelize.INTEGER,
46        references: { model: 'Cliente', key: 'id' }
47      },
48
49      UsuarioId: {
50        type: Sequelize.INTEGER,
51        references: { model: 'Usuario', key: 'id' }
52      },

```

```

53     MotoId: {
54       type: Sequelize.INTEGER,
55       references: { model: 'Moto', key: 'id' }
56     },
57     is_active: {
58       type: Sequelize.BOOLEAN,
59       defaultValue: true,
60     },
61     createdAt: {
62       allowNull: false,
63       type: Sequelize.DATE
64     },
65     updatedAt: {
66       allowNull: false,
67       type: Sequelize.DATE
68     }
69   });
70 },
71   async down(queryInterface, Sequelize) {
72     await queryInterface.dropTable('OrdenTrabajo');
73   }
74 };

```

CÓDIGO 3.3. Código para migración en Sequelize.

Como se puede notar, el código es muy parecido al del modelo, con la diferencia que en este caso, además de los campos y sus atributos, sólo se deben definir directamente las claves que representan a las relaciones 1 a muchos que tendrá esta tabla, dejando de lado el resto de relaciones.

Otra de las particularidades principales del archivo de migración es que contiene dos funciones que se ejecutan de manera asíncrona, la función `up` y `down`, la primera creará la tabla y la segunda se ejecuta en caso de alguna falla y elimina la tabla.

3.3.5. Interacción con la base de datos

En Sequelize se pueden realizar todas las funciones necesarias para interactuar con la base de datos, ya sea para insertar, actualizar o eliminar registros, como así también los métodos para leer registros utilizando filtros o condiciones particulares.

A continuación se representan a modo de ejemplo algunos fragmentos de código Javascript empleados para tal fin:

Código para traer todas las órdenes de trabajo existentes, incluyendo otros modelos relacionados a este:

```

1  /**
2   *
3   * @method getOrdenesTrabajo
4   * @description
5   * Traer todas las ordenes de trabajo existentes
6   * @returns
7   * listado de todas las ordenes de trabajos existentes
8   */
9  const getOrdenesTrabajo = async (req, res) => {
10    const ordenesTrabajo = await OrdenTrabajo.findAll({
11      include: [
12        { model: Cliente },

```

```

13         {model: Estado},
14         {model: Moto},
15         {model: Trabajo},
16         {model: Usuario},
17     ]
18 });
19 return res.json(ordenesTrabajo)
20 }

```

CÓDIGO 3.4. Código para traer datos en Sequelize.

En el siguiente código se representa de manera resumida cómo crear un nuevo registro, se quitaron algunas partes del código que son relevantes a esta sección:

```

1  /**
2   * Crear nueva orden de trabajo
3   */
4  const nuevaOrdenTrabajo = async (req, res) => {
5      const nuevaOrdenTrabajo = await OrdenTrabajo.create(
6          req.body,
7      );
8
9
10     return res.json(nuevaOrdenTrabajo);
11
12     } catch (error) {
13         return res.status(400).json({
14             error: error.message,
15             message: 'Error cargando orden',
16             context: 'api > controllers > ordenTrabajoController >
nuevaOrdenTrabajo'
17         });
18     }
19
20 }

```

CÓDIGO 3.5. Código resumido para crear nuevo registro en la base de datos.

En las líneas 5 y 6 es dónde se realiza la creación del registro, pasándole los datos que se reciben como parámetros en el *body* de la petición HTTP.

Por último se muestra cómo actualizar un registro de la base de datos, previamente trayendo el objeto por su id:

```

1
2  // Traigo la Orden de trabajo
3  let orden = await OrdenTrabajo.findOne({
4      where: {
5          id: req.params.id_orden,
6      }
7  });
8
9  // Modifico estado, precio, detalle y entrega de la Orden
10  await orden.update(
11      {
12          EstadoId : estado.id,
13          precio: req.body.precio,
14          entrega: req.body.entrega,
15          detalle: req.body.detalle
16      });

```

CÓDIGO 3.6. Código resumido para actualizar un registro en la base de datos.

Podemos observar que en la línea 3 se busca el registro mediante su id y se guarda el mismo en una variable, luego se utiliza esa variable para realizar la actualización de los datos, por lo que esa variable pasa a ser un “objeto de Sequelize” que puede ser tratado directamente como entidad única de la base de datos, facilitando su uso mediante el ORM.

3.4. Desarrollo API REST

La lógica del backend se centra principalmente en la API REST desarrollada, en la misma se utilizaron diferentes tecnologías las cuales están descritas en el capítulo 2.4.

A continuación se detalla su desarrollo e implementación.

3.4.1. Patrón de desarrollo

Se implementó una arquitectura de software orientada a eventos, nativa de Node.js, y la organización modular del código también llamado “*event-driven*”.

En la organización modular se estructura el código de la aplicación en pequeñas piezas reutilizables. Esto hace que el código sea más fácil de mantener y actualizar a medida que la aplicación crece y evoluciona.

En la figura 3.8 se presenta la estructura de carpetas utilizada siguiendo el modelo mencionado.

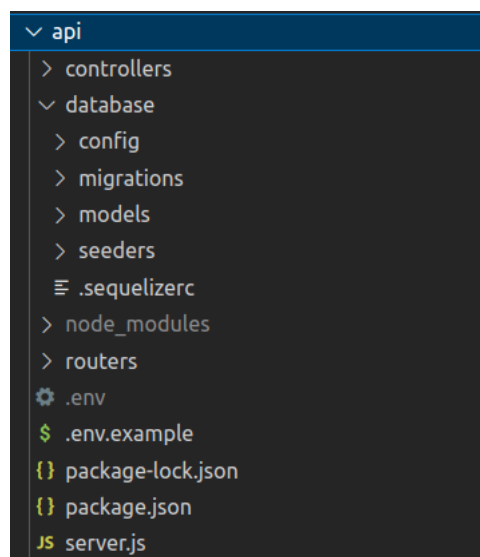


FIGURA 3.8. Estructura de carpetas en patrón de desarrollo modular.

Dentro de cada una de las carpetas se encuentran los archivos Javascript correspondientes. El archivo *server.js* es el punto de partida de la API y donde se realizan las configuraciones de conexión a bases de datos con Sequelize, configuración de Express, se definen los puertos a utilizar, la conexión MQTT, se declaran las rutas o *routes*, entre otras configuraciones.

En el archivo *package.json* se encuentran configuraciones y datos fundamentales para la aplicación, entre estas están: el nombre de la aplicación, la versión, descripción del proyecto, los scripts de inicio y ejecución, las dependencias del proyecto, bibliotecas y paquetes de terceros utilizados, las versiones específicas de las dependencias requeridas y la información de licencia. En este archivo se define, por ejemplo, el uso de Sequelize y Express y sus respectivas versiones.

El archivo *.env* es un archivo que no se versiona, esto significa que no se resguarda el historial de cambios del archivo, esto se debe a que contiene información sensible que debe mantenerse segura y no debe versionarse ni subirse a internet. En cada entorno donde la aplicación sea instalada se deberá crear este archivo manualmente y escribir las definiciones de manera manual. Se declara, por ejemplo, usuario y contraseña para la conexión a base de datos, credenciales para conexión a *broker* MQTT, direcciones IP o URL, conexión a Wi-Fi, entre otros datos sensibles.

En las siguientes secciones se detallan las carpetas *routers* y *controllers*.

3.4.2. Ruteo de la API

En la carpeta *routers* se encuentran los archivos Javascript que corresponden a las rutas que manejará Express para recibir las peticiones o *requests* HTTP provenientes desde fuera de la API, generalmente desde el cliente *frontend*.

En la figura 3.9 se muestran los archivos de ruteo que se crearon siguiendo el patrón modular, separando las rutas según su entidad o funcionalidad.

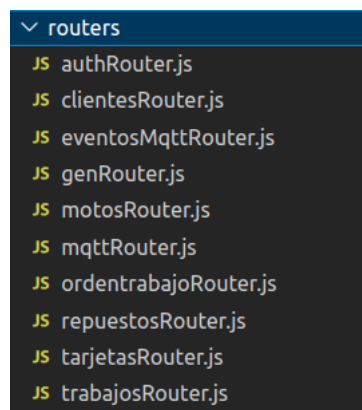


FIGURA 3.9. Estructura de archivos en carpeta de rutas.

En el archivo *ordentrabajoRouter.js* se manejan las peticiones a rutas referentes a las órdenes de trabajo, de la misma manera para los demás archivos.

A modo de ejemplo se representa un fragmento de código de ruteo correspondientes a órdenes de trabajo:

```

1  /**
2   * Nueva orden de trabajo
3   * @params data, estado = espera
4   * Recibe datos para la orden de trabajo
5   * incluido el numero de tarjeta rfid
6   */
7  ordentrabajoRouter.post('/nueva',
8    ordentrabajoCtrl.nuevaOrdenTrabajo);

```

CÓDIGO 3.7. Código de ruteo para órdenes de trabajo.

En la declaración de esta ruta podemos observar que la dirección de la ruta en cuestión es */nueva* y se define con el método *post()* en un objeto de router llamado *ordentrabajoRouter*. Este método indica que la ruta es accesible mediante una solicitud HTTP POST, que es utilizada para enviar información a un servidor para crear un recurso.

Además, se especifica un controlador de ruta (*handler*). El controlador es una función que se ejecuta cuando se realiza una solicitud HTTP a la ruta especificada. En este caso, el controlador de ruta se llama *nuevaOrdenTrabajo* y se define en el archivo *ordentrabajoCtrl*, más adelante veremos en profundidad la carpeta *controllers* y sus características.

Cuando se realiza una solicitud HTTP POST a la ruta */nueva*, Express ejecutará automáticamente la función *nuevaOrdenTrabajo* definida en el controlador de ruta *ordentrabajoCtrl*.

De esta manera se logra separar la lógica de ruteo de la lógica de procesos y acceso a datos, siguiendo el patrón modular mencionado previamente.

3.4.3. Controladores de la API

Como se mencionó en la sección previa, los controladores son funciones que se ejecutan luego de resolver la ruta a la cual están asociados.

En la figura 3.10 se puede ver como en la carpeta *controllers* se definieron los controladores siguiendo el patrón modular de desarrollo, teniendo en cuenta la entidad o la funcionalidad que se desea procesar.

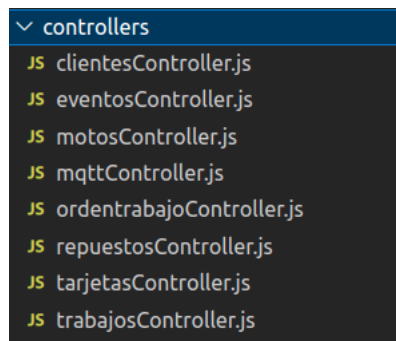


FIGURA 3.10. Estructura de archivos en la carpeta *controllers*.

A modo de ejemplo se muestra a continuación un fragmento de código del controlador para obtener un listado de todas las motos de la base de datos:

```

1  const {Moto} = require('../database/models/index');
2
3  const { Op, Sequelize } = require("sequelize");
4  const { response } = require('express');
5
6  //controller
7  const listarMoto = async (req, res)=>{
8      try {
9          const listadoMotos = await Moto.findAll({});
10
11         return res.status(200).json({
12             listadoMotos
13         });
14     }

```

```

15     } catch (error) {
16         return res.status(400).json({
17             error: error.message,
18             message: 'Error listando motos'
19         })
20     }
21
22 };

```

CÓDIGO 3.8. Código de controlador para obtener listado de motos.

La primera línea del código importa el modelo *Moto* desde el archivo principal para los modelos de Sequelize *index.js* ubicado en la carpeta *models*. En la segunda línea, se importan las funciones *Op* y *Sequelize* desde la librería Sequelize, estas funciones se utilizan para realizar operaciones y consultas en la base de datos. Luego, se define el controlador *listarMoto* que se encarga de listar todas las motos de la base de datos. Dentro del controlador, se utiliza el método *findAll()* de Sequelize para buscar todos los registros. Si la consulta a la base de datos se realiza con éxito, se devuelve un objeto *JSON* con el listado de motos y un código de estado HTTP 200. En caso de que la consulta falle, se devuelve un objeto *JSON* con el mensaje de error y un código de estado HTTP 400.

Finalmente, se exporta el controlador *listarMoto* para que pueda ser utilizado en otras partes de la aplicación.

De manera similar se desarrollaron todos los controladores para las diferentes rutas de la aplicación.

3.4.4. Endpoints HTTP

Para la organización del desarrollo se realizó una tabla con todos los datos necesarios para los *endpoints* o rutas de la API.

En la figura 3.11 se puede apreciar una parte de esta tabla correspondiente a las rutas para las órdenes de trabajo.

METODO	URL	FUNCION	DETALLE	BODY
GET	ordentrabajo/todos	Obtener ordenes de trabajo	todas las ordenes	
GET	ordentrabajo/filtrado	Filtrado por campo	endpoint para traer ordenes segun filtro: fecha, estado,	data filter
GET	ordentrabajo/cliente	filtrado orden de trabajo por dni del cliente	para la pantalla busqueda orden de trabajo del cliente.	
ORDENES DE TRABAJO				
POST	ordentrabajo/nueva	insertar nueva orden	Datos de la orden con numero de tarjeta incluida, front busca en Eventos, muestra msj si desea marcar retirado o finalizado, si el msj se acepta envia put para marcar estado de esa orden en retirado/finalizado, si es finalizado debe enviar sms/email.	data, obligatorio es parámetro "estado", se envia un string con el nombre del estado. Por defecto enviar "espera"
PUT	ordentrabajo/cambiareestado	cambiar estado orden y tarjeta		id orden
PUT	ordentrabajo/editar	cambiar datos de la orden	Cambiar otros datos de la orden como precio, detalle, etc.	data
PUT	eventos/activos	consultar si hay evento activo, si hay desactivarlo	front consulta si hay evento activo, se ocupará para cuando se pase una tarjeta y se deban agregar datos en front, por ejemplo cargar una orden nueva o confirmar el retiro de un repuesto. Si hay evento is_active en True, se hace un update para pasar a False y se trae ese registro con todos sus datos. ***Resolver que hacer si hay varios eventos activos, ya que no debería, si es así se puede traer el sólo el último y pasar todos a false,	

FIGURA 3.11. Tabla descriptiva de *endpoints* o rutas de la API.

A medida que se avanzó en el desarrollo se fueron marcando en verde las casillas de las rutas finalizadas, además se fue agregando más detalles o parámetros según requerimientos o necesidades.

3.5. Comunicación MQTT

3.5.1. Diagrama MQTT

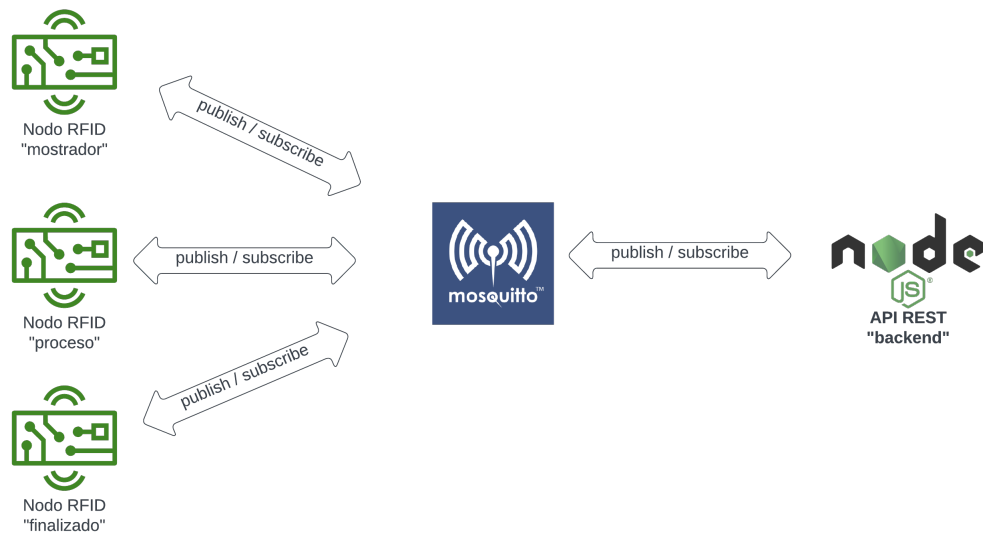


FIGURA 3.12. Diagrama de funciones MQTT.

En la figura 3.12 se pueden apreciar los distintos actores que intervienen en la comunicación MQTT del sistema. Los nodos RFID representan a cada ESP32 con el módulo lector RC522 mencionados en el capítulo 2.7.2, estos se encargan de marcar el estado específico de una orden de trabajo. En el centro se ubica el broker Mosquitto el cual se encarga de la distribución de los mensajes a los demás módulos y a la derecha está la API REST desarrollada en Node.js la cual recibe los mensajes provenientes de los nodos y realiza el proceso correspondiente, además de enviar publicaciones a modo de información y para *logs*.

3.5.2. Topics MQTT

Como se observa en la figura 3.13, se organizan los *topics* MQTT en una tabla, en este caso, según el nodo o sensor y podemos ver en la cabecera de la tabla la información que representaremos para cada mensaje, el *topic*, si es una publicación o suscripción, el mensaje en cuestión y una breve explicación de cual es la función que cumple la comunicación.

Método	topic	pub	sub	message	funcion
TODO LOS SENSORES					
SUB	*	todos	api	messages	Guardar mensaje mqtt en tabla general
SENSOR 1					
PUB	nodo/discriminar	NODO MOSTRADOR		tarjeta, nodo, estado: discriminar	Enviar numero de tarjeta, nodo y estado
					API: busco estado actual de la tarjeta, si es libre va accion nueva , si es otra va accion en_uso . Guardo en un modelo "eventos" el numero de tarjeta, la accion, y si corresponde los datos de la orden a retirar. <i>Tambien podria guardar el estado actual de la tarjeta.</i>
SUB	nodo/discriminar		API		
PUB	api/cargaoretira	API		tarjeta, accion	envio el numero de tarjeta y la accion a ejecutar: cargar o retirar, segun lo q se recibió por mqtt
SUB			Front		recibe tarjeta y accion a ejecutar: cargar o retirar
SENSOR 2					
PUB	nodo/cambiarestado	NODO PROCESO		tarjeta, nodo, estado: proceso	enviar numero de tarjeta, nodo y estado a cargar "proceso"
					API: busco estado actual, si es distinto libre asigno en proceso a tarjeta y orden. Caso contrario, si es libre, mqtt error a nodo que inicio el mensaje
SUB			API		
PUB	api/error/[nodo_nombre]	API			envio topic error
SUB			NODO PROCESO	error	ejecuto beep error

FIGURA 3.13. Tabla de *topics* MQTT.

La lista completa de *topics* implementados es la siguiente:

- *: API *backend* se suscribe a todos los mensajes.
- nodo/discriminar: emite nodo "mostrador" para ingresar o retirar órdenes de trabajo. Recibe API *backend*.
- nodo/cambiarestado: emite nodo "proceso" o "finalizado" para cambiar el estado de una orden de trabajo. Recibe API *backend*.
- api/error/mostrador: emite la API de *backend* para informar un error en el proceso. Recibe nodo "mostrador".
- api/error/proceso: emite la API de *backend* para informar un error en el proceso. Recibe nodo "proceso".
- api/error/finalizado: emite la API de *backend* para informar un error en el proceso. Recibe nodo "finalizado".
- api/confirmacion/mostrador: emite la API de *backend* para informar proceso exitoso. Recibe nodo "mostrador".
- api/confirmacion/proceso: emite la API de *backend* para informar proceso exitoso. Recibe nodo "proceso".
- api/confirmacion/finalizado: emite la API de *backend* para informar proceso exitoso. Recibe nodo "finalizado".

3.5.3. Broker MQTT

Para la implementación en el servidor del sistema en la Raspberry Pi del *broker* MQTT "Eclipse Mosquitto" se utilizó un servicio de Docker Compose el cual permite levantar el *broker* y sus configuraciones.

En la figura 3.14 podemos observar la estructura de carpetas y archivos que se implementa en el contenedor de Docker para este servicio.

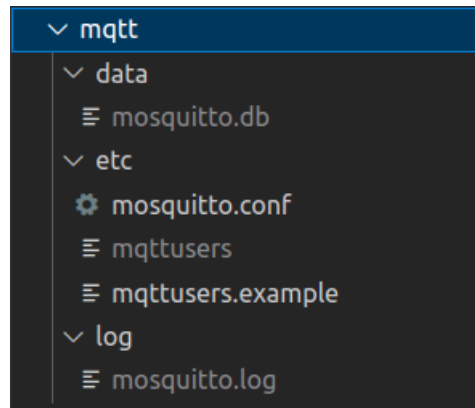


FIGURA 3.14. Estructura de carpetas del servicio Eclipse Mosquitto en Docker.

- En la carpeta *data* se encuentra el archivo de base de datos *mosquitto.db* donde se pueden persistir los mensajes MQTT según la configuración de persistencia implementada.
- En la carpeta *etc* se encuentra el archivo de configuración principal del *broker*: *mosquitto.conf*, aquí se declaran todos los parámetros con los cuales Docker Compose levantará el servicio. El archivo *mqttusers* persiste de manera encriptada las credenciales de acceso para los clientes MQTT, estos usuarios son administrados por línea de comando en la terminal *bash* ingresando al contenedor específico. Por último se agregó un archivo con el ejemplo de comandos para administrar usuarios: *mqttusers.example*.
- La carpeta *log* contiene el archivo *mosquitto.log* donde se registran todos los logs del servicio y que sirven para hacer un *debug* o *test* del mismo.

3.5.4. MQTT en nodos

Para implementar las comunicaciones MQTT en los nodos ESP32 se utilizó la biblioteca de código abierto PubSubClient [42] la cual permite conexiones con servidores MQTT, publicar mensajes y suscribirse a los mensajes recibidos.

Como se describió en el capítulo 1 Cada nodo representa uno o más estados específicos en la cadena de procesos que se realizan en la empresa. Estos estados son declarados en el código fuente del microcontrolador ESP32, específicamente en un archivo de entorno, de manera que pueda ser modificado fácilmente en caso de ser necesario.

A continuación se describen los nodos y sus estados utilizados:

- Nodo 1, “mostrador”: este nodo cumple la función comunicar a la API REST por medio de MQTT el número de tarjeta que ha sido leída en el mostrador de atención a clientes de la empresa. De esta manera la API REST se encargará de asignar los estados “espera” o “retirado” a las órdenes de trabajo según corresponda.
- Nodo 2, “proceso”: este nodo se encarga de comunicar a la API REST que debe cambiar el estado de la orden de trabajo a “proceso”, la API REST reconoce que el mensaje proviene de este nodo por el “topic” por el cual se

envía el mensaje, y consulta a la base de datos qué orden de trabajo tiene el número de tarjeta recibida para ejecutar la actualización.

- Nodo 3, “finalizado”: de la misma manera que en el nodo 2, en este sensor se informa a la API REST que debe cambiar el estado de la orden de trabajo asociada a la tarjeta leída a “finalizado”.

De esta manera el ciclo de lecturas de los nodos para una orden de trabajo es el siguiente:

1. Nodo “mostrador”, se asigna el estado “espera”.
2. Nodo “proceso”, se asigna el estado “proceso”.
3. Nodo “finalizado”, se asigna el estado “finalizado”.
4. Nodo “mostrador”, se asigna el estado “retirado”.

Cuando un nodo envía o recibe una señal MQTT emite alertas sonoras para informar al usuario si la operación correspondiente tuvo éxito o fracasó. En la figura 3.15 se puede observar como se implementaron estas señales sonoras en los nodos. Además podemos ver que existen dos opciones de sonidos, una es en forma de una sola nota denominada en la tabla *Sonidos Clásicos* y otra es en forma de una melodía denominada *Melodías*. Se configura en cada nodo la opción deseada desde las variables de entorno. Además se realiza en el código una validación para asegurarse que el tipo de *buzzer* instalado es compatible con la opción *Melodía*, en caso que no lo sea se ejecuta por defecto la opción *Sonidos Clásicos*

Sonidos Clasicos			
Milisegundos	Cantidad	Indica	Detalles
200	2	Conexion a WIFI exitosa o Confirmacion API	Cuando se enciende el nodo = coneccion a wifi ok Cuando se pasa tarjeta = la API recibió y procesó el mensaje correctamene
500	1	Tarjeta leida ok, envío mensaje ok	Se envió correctamente el mensaje a la API
Melodias			
Tapa Tapita Tapon	1	Conexion a wifi exitosa	Cuando se enciende el nodo = coneccion a wifi ok
Tapa Tapita	1	Tarjeta leida ok, envío mensaje	Se envió correctamente el mensaje a la API
Tapon	1	Confirmacion API	La API recibió y procesó el mensaje correctamene
Errores			
150	3	Error en lectura de tarjeta	Cuando la tarjeta no puede ser leída por el nodo.
150	6	Error al procesar tarjeta enviada	Cuando enviamos desde un nodo al servidor una tarjeta y este no puede procesarla por algun motivo, ej: la tarjeta ya está en uso

FIGURA 3.15. Tabla de detalles de sonidos de nodos.

3.5.5. MQTT en API REST

Se definió en la estructura de la API REST, descrita en el capítulo 3.4.1, una sola ruta de entrada en el archivo principal de configuración *server.js*, dónde se realiza la conexión al *broker* MQTT y se subscribe a todos los *topics* utilizando para ello el símbolo #.

Configuración y conexión a *broker* MQTT:

```

1 // mqtt config
2 const mqtt = require('mqtt')
3 const host = process.env.MQTT_SERVER

```



```

4   const port = process.env.MQTT_PORT
5   const clientId = `api_mqtt_${Math.random().toString(16).slice(3)}`
6
7   // mqtt connect function
8   const connectUrl = `mqtt://${host}:${port}`
9   const mqtt_client = mqtt.connect(connectUrl, {
10     clientId,
11     clean: true,
12     connectTimeout: 4000,
13     username: process.env.MQTT_USER,
14     password: process.env.MQTT_PASSWORD,
15     reconnectPeriod: 3000,
16   })

```

CÓDIGO 3.9. Configuración y conexión a *broker* MQTT en API REST.

Como se puede observar en el código 3.9 se traen la mayoría de los parámetros desde las variables de entorno, esto se realiza para mantener un código limpio y que sea fácil de mantener, en caso de requerir algún cambio se realiza directamente en el archivo de variables de entorno evitando tener que cambiar en varias partes del código.

Primero se importa la librería MQTT y se definen las variables correspondientes al *host* y puerto del *broker*, así como un *clientId* generado aleatoriamente. Luego se define la función de conexión al *broker*, utilizando la URL compuesta por el *host* y *puerto* definidos anteriormente, y se especifican las opciones de conexión, incluyendo el *clientId* generado, la limpieza de sesión en la conexión, un tiempo de espera de conexión de 4 segundos, el nombre de usuario y contraseña para la conexión, y un período de reconexión de 3 segundos.

Subscripción a todos los *topics*:

```

1 // subscribe to topics
2 const topic = process.env.MQTT_TOPIC_ALL;
3 mqtt_client.on('connect', () => {
4   console.log('mqtt client Connected')
5   mqtt_client.subscribe([topic], () => {
6     console.log('API Subscribe to topic `${topic}`')
7   })
8 })

```

CÓDIGO 3.10. Subscripción a *topics* en API REST.

En el código 3.10, en la línea 2 se trae desde las variables de entorno el *topic* correspondiente. Luego se inicia la conexión con el método *on* y se realiza la subscripción con el método *subscribe*.

Luego los mensajes se rutean a un controlador MQTT que procesa todos los mensajes y siguiendo un flujo condicional realiza las acciones correspondientes.

A continuación se presenta un diagrama del flujo del código fuente para resolver los procesos según el mensaje MQTT recibido:

Como puede observarse en la figura 3.16 las validaciones dependen tanto del *topic* recibido como del estado de la tarjeta. De acuerdo a eso se realizan los procesos correspondientes y se devuelve siempre un mensaje MQTT al nodo emisor el cual, de acuerdo al mensaje recibido, emitirá una señal sonora a través del *buzzer* para informar al usuario si la acción tuvo éxito o no.

3.6. API para mensajería

Para el envío de mensajes por la aplicación *WhatsApp* [25] se utilizó una API adicional [26] la cual fue personalizada para las necesidades de este sistema.

En esta sección se detalla como se implementó una API de mensajería para *WhatsApp*.

Esta API permite al usuario de la aplicación conectarse a *WhatsApp* con su teléfono móvil y enviar mensajes desde la aplicación a los clientes de manera automática.

3.6.1. Implementación

Para implementar esta API se creó un conenedor en Docker de manera tal de separar este servicio del resto de los demás siguiendo una arquitectura de tipo microservicios [43].

El patrón de desarrollo es similar al de la API REST detallada en el capítulo 3.4 y también está desarrollada en Node.js y Express, por lo que su personalización fue sencilla para este trabajo.

Se modificó el código fuente de la API para poder autenticarse desde el frontend al servicio de *WhatsApp*, haciendo que el frontend pueda realizar una petición por medio del protocolo HTTP y que la API devuelva un código QR [44] en formato SVG [45] para que el usuario pueda realizar la conexión.

En el ruteo del proyecto se añadió un endpoint para obtener un código QR de autenticación:

```
1 router.get("/", leadCtrl.getQrCode);
2 router.get("/regenerateqr", leadCtrl.regenerateQrCode);
```

CÓDIGO 3.11. Endpoint para obtener código QR.

Luego en el controlador se realiza la lógica correspondiente:

```
1 public getQrCode = async(req: Request, res: Response)=>{
2   const path = `${process.cwd()}/tmp`;
3   res.setHeader('Content-Type', 'image/svg+xml');
4   res.sendFile(`${path}/qr.svg`);
5 }
6
7 public regenerateQrCode = async(req: Request, res: Response)=>{
8   console.log("logout test");
9   const response = await this.leadCreator.logoutSrv();
10  res.send(response);
11 }
```

CÓDIGO 3.12. Controlador de ruta para obtener código QR.

La primera función, llamada *getQrCode*, es un controlador que se encarga de mostrar el código QR para autenticar la sesión de *WhatsApp* en la aplicación. Para hacerlo, el código utiliza la función *res.sendFile()* para enviar el archivo *qr.svg* que se encuentra en la carpeta *tmp* del directorio actual del proyecto. Además, se establece el encabezado de respuesta *Content-Type* a *image/svg+xml* para indicar que se está enviando un archivo SVG.

La segunda función, llamada *regenerateQrCode*, es otro controlador que se encarga de cerrar la sesión de *WhatsApp* actual y volver a generar un nuevo código

QR. Para hacerlo, se llama a una función *logoutSrv()* que se encuentra en la clase *leadCreator*. Luego, se envía la respuesta al cliente con la respuesta recibida de la función *logoutSrv()*.

3.7. Desarrollo frontend

Para el desarrollo de las interfaces de usuario se utilizó el framework Ionic [32], implementando de esta manera el lenguaje Javascript tanto en el *backend* como en el *frontend*.

3.7.1. Patrón de desarrollo

Se implementó el patrón de diseño MVC [46] (*Modelo-Vista-Controlador*) para estructurar la aplicaciones web. Este patrón divide la aplicación en tres capas principales: el modelo, el cual representa los datos y la lógica de negocio, la vista, que representa las interfaces de usuario, y el controlador, que actúa como intermediario entre la vista y el modelo.

También se utilizaron otros patrones de diseño, como el patrón de “Inyección de Dependencias” [47] *Dependency Injection Pattern* y el patrón de “Observador” [48] *Observer Pattern*. Estos patrones son utilizados para mejorar la escalabilidad, la flexibilidad y la mantenibilidad de las aplicaciones desarrolladas con Ionic.

En la capa de la vista se utilizaron las páginas, componentes y módulos de Ionic. Las páginas son plantillas en HTML o lenguaje IONIC. Los componentes son elementos visuales que se desarrollan en lenguajes HTML, CSS y Javascript y son implementados en las páginas. Los módulos son grupos de páginas y componentes.

En la capa de controlador se utilizó un archivo Javascript con las funciones y la lógica para cada componente, dándole funcionalidad al mismo.

En la capa de modelo se implementaron los servicios de Ionic, estos son clases que contienen métodos y lógica de negocio para recuperar datos de la API.

Además se utilizó el manejo de rutas para mapear las URL a las páginas. Se asignó a cada módulo un archivo de rutas independiente para el manejo fluido de las páginas.

En la figura 3.17 podemos observar una representación del flujo utilizando este patrón.

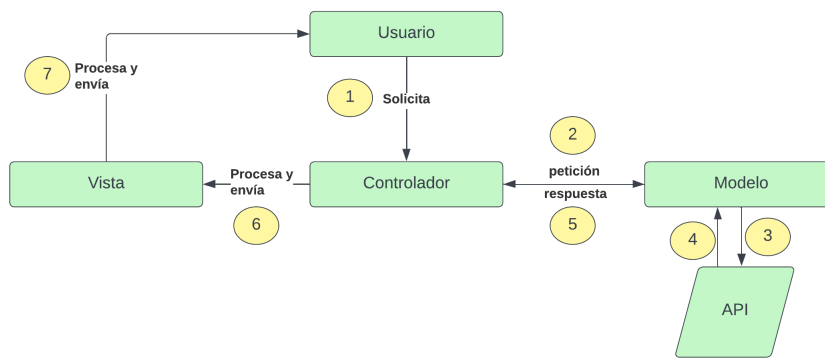


FIGURA 3.17. Flujo en MVC.

Por ejemplo, para el desarrollo del módulo de listar órdenes de trabajo implementando este patrón, el flujo sería el siguiente:

1. Usuario solicita a través de su navegador la URL *ordenestrabajo/listar*.
2. El *routing* mapea la URL al controlador *listar.page.ts* que se encuentra en el módulo *listar.module.ts*. El controlador llama al servicio *ordentrabajo.services.ts*.
3. El servicio hace una petición HTTP por medio del método GET a una ruta de la API de *backend*.
4. La API responde con los datos de la lista en formato JSON.
5. El servicio entrega esos datos al controlador en formato JSON.
6. El controlador procesa los datos, realiza validaciones y filtros necesarios, ordena los datos, etc.
7. La vista procesa los datos, los inserta en una tabla, aplica los diseños CSS, etc. Luego envía la vista con los datos al usuario quien la visualizará en su navegador.

De esta manera se implementó este patrón de desarrollo para todos los componentes y módulos del *frontend*.

3.7.2. Interfaces de usuario

Teniendo en cuenta el flujo general del sistema presentado en el capítulo 3.2 y los requerimientos para un sistema que sea amigable con el usuario y de diseño adaptable a diferentes dispositivos, se desarrollarán las interfaces de usuario utilizando los componentes nativos de Ionic para diseño de interfaces junto a CSS para personalizar estilos y javascript para funcionalidades.

Las interfaces de usuario que se desarrollaron fueron:

1. Nueva: para añadir una nueva orden de trabajo.
2. Listar: para listar todas las órdenes de trabajo.
3. Retirar: para retirar una orden de trabajo.
4. Clientes: de tipo modal, para añadir o buscar un cliente.

5. Motos: de tipo modal, para añadir o buscar una motocicleta.
6. QR: para mostrar el código QR para autenticación en *Whatsapp*.

Nueva orden de trabajo

En la figura 3.18 podemos observar los primeros campos del formulario a rellenar para cargar una nueva orden de trabajo.

NUEVA ORDEN DE TRABAJO				
Tarjeta 378952445		<button>LEER TARJETA</button> <button>CANCELAR</button> <button>LIMPIAR</button>		
N° Orden Papel: 8557955				
Rectificación ▾				
CLIENTE (*)				
Nombre	Apellido	Celular	DNI	
pablo	arancibia	3624101031	0	
MOTO (*)				
Marca	Modelo	Cilindrada	Año	Observaciones
Honda	New Titán		2020	modelo nuevo, 2020-2022

FIGURA 3.18. Interfaz de usuario para nueva orden de trabajo.

El usuario deberá hacer clic en el botón “LEER TARJETA” y luego pasar la tarjeta RFID por el lector “mostrador”. De esta manera el sistema trae el número de tarjeta que fue seleccionada para asignar a la orden. Luego deberá cargar, si existe, un número de orden de manera manual el cual corresponde a un *ticket* que usa actualmente la empresa.

Para cargar el tipo de trabajo, datos del cliente y la motocicleta se utilizan interfaces de tipo modal, las cuales se presentan a continuación:

En la figura 3.19 se presenta la interfaz de tipo modal para seleccionar el tipo de trabajo a asignar a la orden.

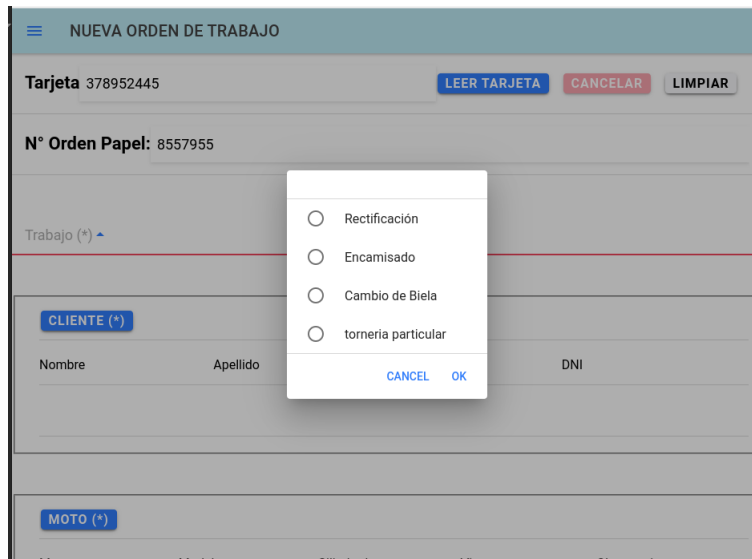


FIGURA 3.19. Interfaz de usuario de tipo modal para seleccionar tipo de trabajo.

Para cargar o buscar los datos de clientes se utilizó el mismo componente, reutilizando así parte del código fuente. Como podemos ver en las figuras 3.20 y 3.21, también se usa el tipo de interfaz modal.

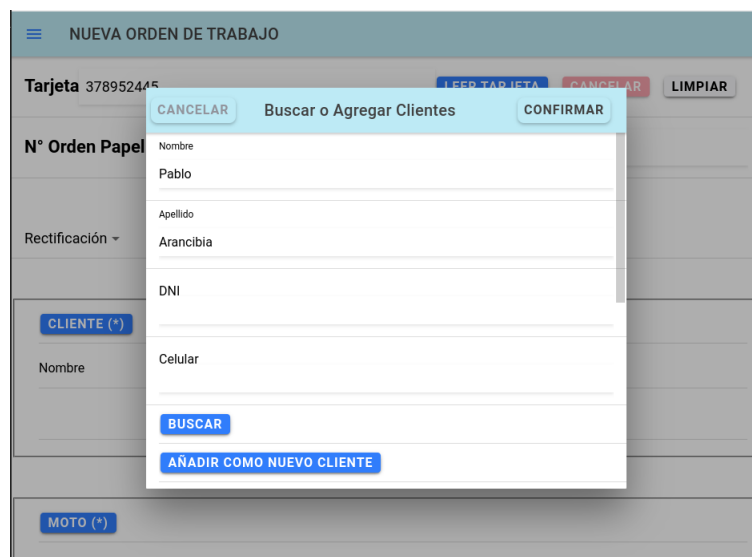


FIGURA 3.20. Interfaz de usuario de tipo modal para seleccionar o buscar cliente.

En la figura 3.21 podemos observar el resultado de una búsqueda de clientes. Se selecciona el que se desea asignar y se hace click en “CONFIRMAR”.

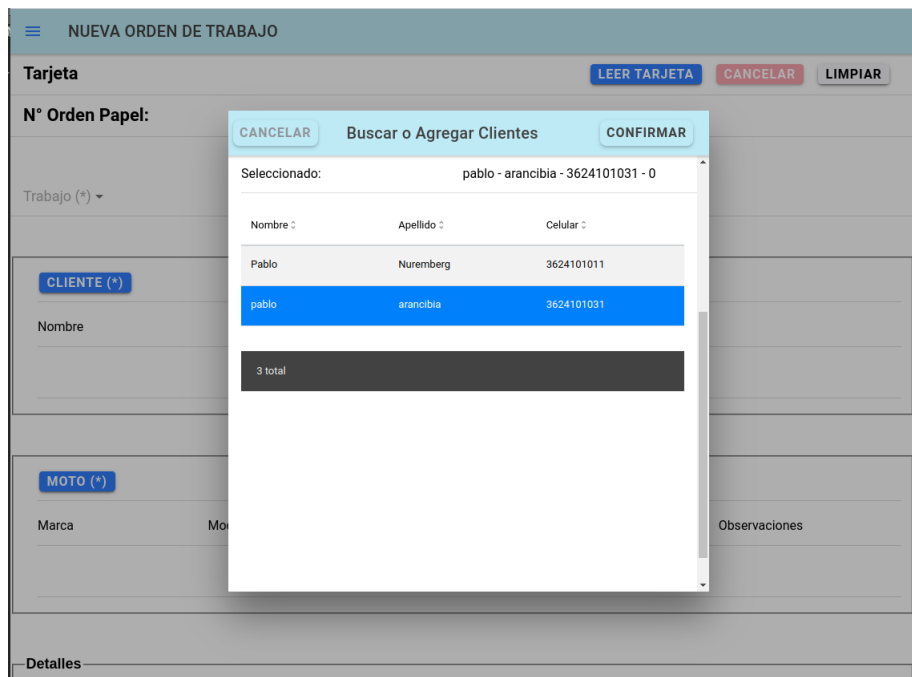


FIGURA 3.21. Interfaz de usuario de tipo modal con el resultado de búsqueda de clientes.

De la misma manera se puede seleccionar o crear un registro de datos de una nueva motocicleta para asignar a la orden. En las figuras 3.22 y 3.23 vemos las interfaces modales.

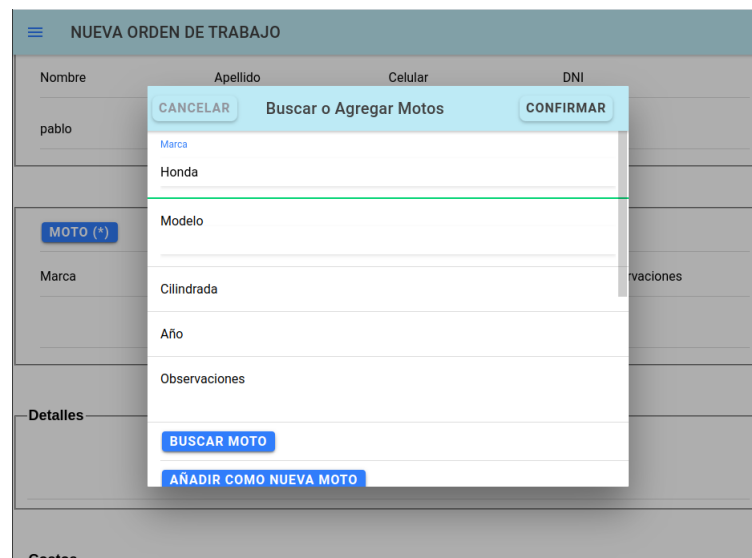


FIGURA 3.22. Interfaz de usuario de tipo modal para seleccionar o buscar motocicleta.



FIGURA 3.23. Interfaz de usuario de tipo modal con el resultado de búsqueda de motocicletas.

Volviendo a la segunda sección del formulario para agregar una nueva orden, ya habiendo cargado el número de tarjeta, el tipo de trabajo, el cliente y la motocicleta, continuamos con los campos que se representan en la figura 3.24.

NUEVA ORDEN DE TRABAJO

Honda

New Titán

2020

modelo nuevo, 2020-2022

Detalles

Agrego algunos detalles para la orden

Costos

Precio \$

1000

Entrega \$

550

Saldo \$

450

Fecha de entrega estimada

19 de jun de 2023

2:00 p. m.

GUARDAR Y LIMPIAR

GUARDAR Y SEGUIR AÑADIENDO

LIMPIAR FORMULARIO

FIGURA 3.24. Interfaz de usuario para nueva orden de trabajo.

En esta sección podemos agregar detalles, costos y fecha de entrega estimada.

- **Detalles:** para agregar algunas observaciones a la orden en formato de texto
- **Costos:** están separados en 3 campos: el precio total de la orden, la entrega que realiza el cliente cuando deja el repuesto y el saldo restante.

- Fecha de entrega estimada: al seleccionar la fecha saldrá un calendario de tipo modal para seleccionar el día y la hora de entrega.

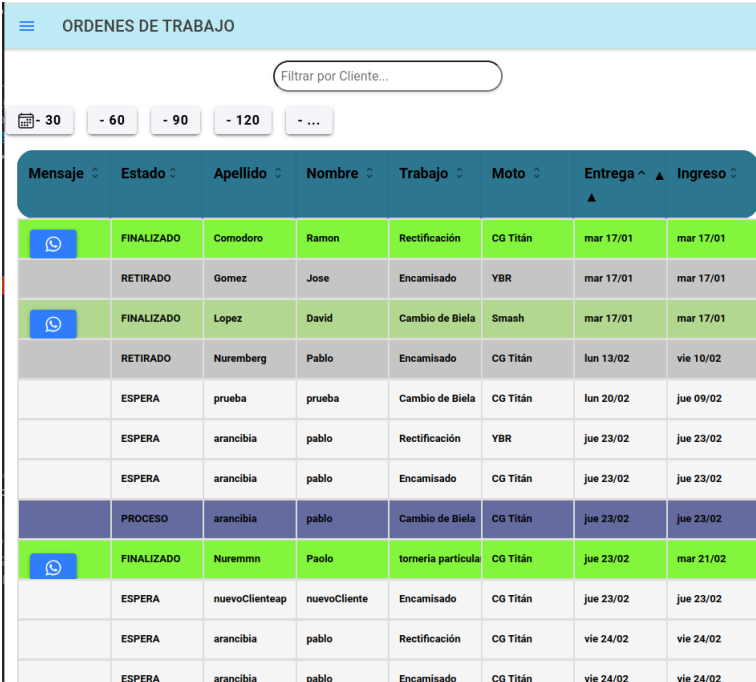
Por último sólo queda guardar la orden, para ello se pueden realizar dos acciones:

- GUARDAR Y LIMPIAR: esta opción se utilizará cuando el cliente sólo deje un repuesto para trabajar de manera tal que se guarda el registro y se limpia completamente el formulario
- GUARDAR Y SEGUIR AÑADIENDO: cuando el mismo cliente tiene varios repuestos para dejar, de esta manera se guarda el registro pero se mantienen los datos del cliente en el formulario.

La última opción corresponde a “LIMPIAR FORMULARIO” que borra los datos del formulario sin guardar ningún registro.

Listar órdenes de trabajo

En la figura 3.25 podemos observar la interfaz gráfica para donde se listan las órdenes de trabajo según las opciones seleccionadas.



The screenshot shows a web interface titled "ORDENES DE TRABAJO". It features a search bar "Filtrar por Cliente...", a date range selector with buttons for 30, 60, 90, 120, and an ellipsis, and a table of work orders. The table has columns: Mensaje, Estado, Apellido, Nombre, Trabajo, Moto, Entrega, and Ingreso. The data is as follows:

Mensaje	Estado	Apellido	Nombre	Trabajo	Moto	Entrega	Ingreso
[WhatsApp icon]	FINALIZADO	Comodoro	Ramon	Rectificación	CG Titán	mar 17/01	mar 17/01
	RETIRADO	Gomez	Jose	Encamisado	YBR	mar 17/01	mar 17/01
[WhatsApp icon]	FINALIZADO	Lopez	David	Cambio de Biela	Smash	mar 17/01	mar 17/01
	RETIRADO	Nuremberg	Pablo	Encamisado	CG Titán	lun 13/02	vie 10/02
	ESPERA	prueba	prueba	Cambio de Biela	CG Titán	lun 20/02	jue 09/02
	ESPERA	arancibia	pablo	Rectificación	YBR	jue 23/02	jue 23/02
	ESPERA	arancibia	pablo	Encamisado	CG Titán	jue 23/02	jue 23/02
	PROCESO	arancibia	pablo	Cambio de Biela	CG Titán	jue 23/02	jue 23/02
[WhatsApp icon]	FINALIZADO	Nuremn	Paolo	torneria particula	CG Titán	jue 23/02	mar 21/02
	ESPERA	nuevoClienteap	nuevoCliente	Encamisado	CG Titán	jue 23/02	jue 23/02
	ESPERA	arancibia	pablo	Rectificación	CG Titán	vie 24/02	vie 24/02
	ESPERA	arancibia	pablo	Encamisado	CG Titán	vie 24/02	vie 24/02

FIGURA 3.25. Interfaz de usuario para listar las órdenes de trabajo.

Las opciones en esta interfaz son:

- Filtrado: se puede filtrar por cualquier campo de la tabla, una vez que se filtran los resultados, la información se sigue actualizando en tiempo real, por lo que si una fila cambia de estado, se verá de todas maneras el cambio.
- Rango de días: en la parte superior de la tabla se pueden observar 5 botones, estos representan al rango de días hacia atrás que se desea mostrar los registros. Además la tabla muestra al usuario una paginación navegable en la parte inferior si los registros son mayores a 25. El último botón de este grupo corresponde a mostrar el historial completo de registros.

- Ordenamiento por campos: al hacer clic en la cabecera de la tabla se puede ordenar los elementos de manera múltiple, por ejemplo en primer orden la fecha de entrega y en segundo orden el apellido del cliente, así sucesivamente en múltiples campos.
- Orden de columnas: se pueden ordenar los campos haciendo clic y arrastrando el campo deseado al orden correspondiente. Por ejemplo se puede arrastrar el campo “Estado” en la primer columna.

Para una mejor visualización de los estados de los trabajos se asignó un color específico para cada estado, de esta manera es más fácil identificar rápidamente el estado de los trabajos. Se utilizó la siguiente paleta de colores para los estados:

- Espera: color verde.
- Proceso: color púrpura.
- Finalizado: color verde brillante o verde opaco.
- Retirado: color gris.

Además se asigna un botón en la columna “Mensaje”, solamente a las filas de los trabajos que están en estado “FINALIZADO”. Este botón tiene el logo de la aplicación *WhatsApp* y se utiliza para enviar un mensaje al cliente informando que su trabajo está finalizado y listo para retirarse. En caso que el cliente todavía no fue alertado el color de la fila será verde brillante, si el cliente ya fue avisado el color de la fila será verde opaco. El usuario puede reenviar un mensaje a un cliente si lo desea sólo que el sistema lo alertará previamente.

En las figuras 3.26 y 3.27 vemos los alertas de tipo modal que se utilizan para confirmar cuando el usuario va a enviar o reenviar un mensaje a un cliente.

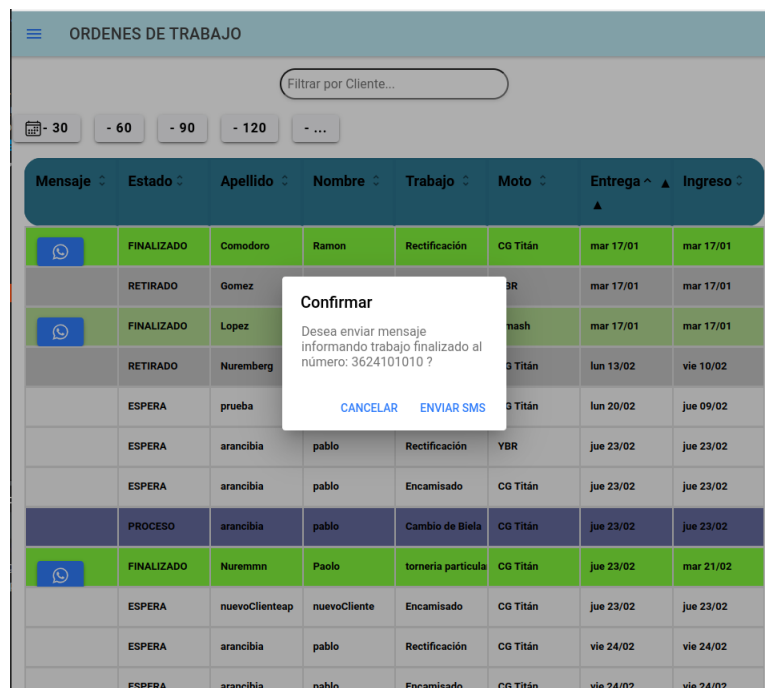


FIGURA 3.26. Confirmar envío de mensaje de texto al cliente.

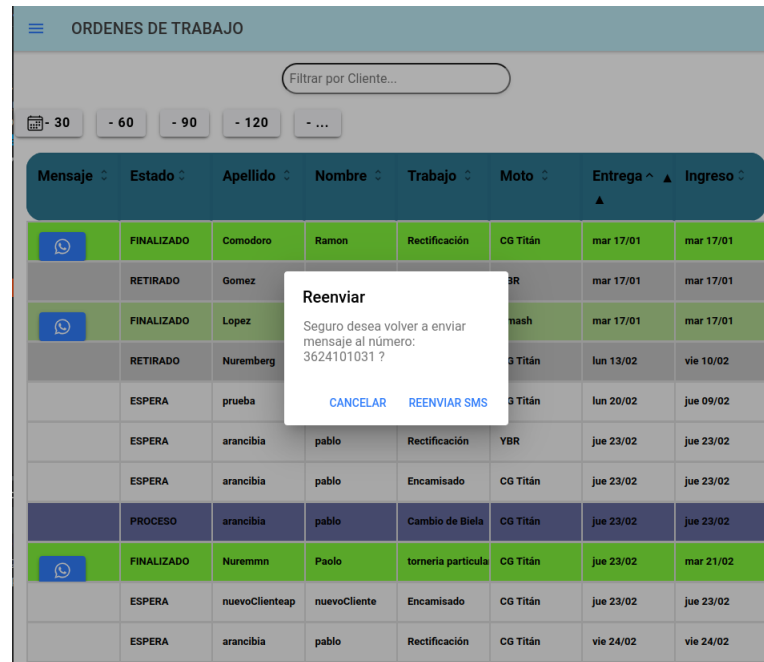


FIGURA 3.27. Confirmar reenvío de mensaje de texto al cliente.

Como se detalló en el capítulo 3.6 el mensaje llegó al cliente en su aplicación *WhatsApp* como si fuera enviado desde el teléfono móvil del usuario, por lo que mostrará su número o el nombre por el cual el cliente lo tenga agendado, ya que la API de mensajería utiliza los servicios de *WhatsApp* para hacer el envío y el usuario debe autenticarse previamente con su teléfono móvil y el código QR para utilizar el servicio.

Retirar orden de trabajo

Para retirar una orden se utiliza una metodología similar a la de agregar una nueva orden de trabajo. El usuario va hacer clic en un botón para activar la búsqueda de un evento, este evento será pasar la tarjeta RFID del repuesto por el nodo mostrador, una vez hecho esto el número de la tarjeta con todos los datos de la orden se cargará en la pantalla.

El usuario puede modificar el precio de la orden en este punto y se actualizarán los datos de saldo. También puede agregar detalles para el retiro de la orden que quedarán registrados en la base de datos.

Una vez confirmado el retiro se guardan los registros y estados correspondientes y se asigna el estado “libre” a la tarjeta para que pueda ser reutilizada para una futura orden de trabajo.

De esta manera finaliza el ciclo de un repuesto que ingresa a la empresa, es procesado y retirado luego por el cliente.

En las figuras 3.28 y 3.29 vemos la interfaz para retirar una orden de trabajo.

RETIRAR TRABAJO

LEER TARJETA

CANCELAR

Tarjeta N° 1111

Orden Papel N° 111222

DATOS CLIENTE

Nombre	DNI
David Lopez	33123456

TRABAJO

Cambio de Biela

DATOS REPUESTOS

Nombre	Marca	Medida	Origen
Camisa	RMD	025	Brasil
Biela	DKM	075	Taiwan

FIGURA 3.28. Formulario para retirar orden de trabajo.

RETIRAR TRABAJO

Biela	DKM	075	Taiwan
-------	-----	-----	--------

MOTO

Marca	Modelo	Cilindrada	Año
Guerrero	Smash	110	2000

PRECIOS

Precio: 3500

MODIFICAR PRECIO

Entrega Realizada: 3500

Saldo Pendiente: 0

Cancela con:

CANCELAR TOTAL

Detalles de la Orden de Trabajo:

Observaciones:

RETIRAR TRABAJO

LIMPIAR FORMULARIO

FIGURA 3.29. Confirmar retiro de orden de trabajo.

Autenticación a API de mensajería

Para la autenticación a la API de mensajería se utiliza un código QR que se obtiene consultando dicha API por protocolo HTTP. La misma devuelve un archivo SVG que se muestra en la interfaz gráfica del usuario para que pueda ser escaneada por su teléfono móvil y de esta manera acceder a las funciones de envío de mensajes a clientes.

En la figura 3.30 podemos ver la interfaz para autenticarse mediante código QR. También se puede observar el botón “ACTUALIZAR” para refrescar el código por uno nuevo’.



FIGURA 3.30. Interfaz para autenticación en API de mensajería.

3.8. Arquitectura de servidor

Para el despliegue de los servicios desarrollados se implementó una arquitectura de contenedores utilizando Docker y Docker Compose en una Raspberry Pi 4.

Para cada proyecto se creó un archivo Dockerfile para el manejo del contenedor correspondiente, a su vez se centraliza el despliegue de estos contenedores en un archivo *yml* de Docker Compose, donde también se levantan servicios directamente sin utilizar un Dockerfile particular.

En esta sección se detalla la implementación de contenedores y servicios en el servidor.

3.8.1. Implementación de contenedores

A continuación se representa el código fuente utilizado en Docker Compose para el despliegue de todos los contenedores de cada proyecto o servicio utilizado:

```
1 version: '3'
2
3 services:
4   db:
5     container_name: mysqlldb
6     restart: always
7     image: mysql
8     env_file:
9       - .env_mysql/.mysql.env
10    volumes:
```

```

11         - ./sql-data/db:/var/lib/mysql
12     ports:
13         - "3306:3306"
14     api:
15         depends_on:
16             - db
17         container_name: backendapi
18         restart: always
19         build:
20             context: .
21             dockerfile: Dockerfile
22         command: bash -c 'while !</dev/tcp/db/3306; do sleep 1; done;
npm run dev'
23     ports:
24         - "3001:3001"
25     env_file:
26         - ./api/.env
27     volumes:
28         - ./:/app
29     mosquito:
30         image: eclipse-mosquitto
31         container_name: mosquittobroker
32         ports:
33             - "1883:1883"
34             - "9001:9001"
35         volumes:
36             - './mqtt/etc/mosquitto.conf:/mosquitto/config/mosquitto.conf'
37             - './mqtt/etc/mqttusers:/mosquitto/config/mqttusers'
38             - './mqtt/data:/mosquitto/data'
39             - './mqtt/log:/mosquitto/log'
40         restart: always
41     web:
42         container_name: web
43         restart: always
44         build:
45             context: .
46             dockerfile: Dockerfile-frontend
47         ports:
48             - "80:80"
49     messenger:
50         container_name: messenger
51         restart: always
52         build:
53             context: api-whatsapp-ts
54             dockerfile: Dockerfile
55         ports:
56             - "3003:3003"
57         volumes:
58             - ./:/messenger
59     portainer:
60         image: portainer/portainer-ce
61         restart: always
62         ports:
63             - "9000:9000"
64         volumes:
65             - /var/run/docker.sock:/var/run/docker.sock
66             - portainer_data:/data
67 volumes:
68     portainer_data:

```

CÓDIGO 3.13. Código de Docker Compose.

A continuación se describen cada uno de los servicios para la aplicación:

- **db:** este servicio utiliza la imagen de Docker para crear un contenedor de base de datos MySQL. Se le asigna un nombre de contenedor *mysqldb*, se establece una opción de reinicio siempre que se detiene, se define un archivo *.env* para configurar las variables de entorno, se mapea un volumen local a la ruta de datos de MySQL y se expone el puerto 3306.
- **api:** este servicio es la API de la aplicación para el *backend*. Su ejecución depende del servicio de la base de datos *db* y utiliza un archivo *.env* para configurar las variables de entorno. Se construye a partir de un archivo Dockerfile. Se mapea al puerto 3001.
- **mosquitto:** este servicio utiliza la imagen de Docker *eclipse-mosquitto* para crear un contenedor de un servidor MQTT. Se le asigna un nombre de contenedor *mosquittobroker*, se expone los puertos 1883 y 9001, se mapea la configuración y los datos de MQTT a un volumen local y se establece una opción de reinicio siempre que se detiene.
- **web:** este servicio es la interfaz *web* de la aplicación la cual utiliza NGINX para servir la aplicación desarrollada en IONIC. Se construye a partir de un archivo Dockerfile en el directorio actual y se mapea el puerto 80.
- **messenger:** este servicio utiliza un archivo Dockerfile en el directorio de la API de mensajería para crear un contenedor y servir la API. Se le asigna un nombre de contenedor *messenger* y se mapea al puerto 3003.
- **portainer:** este servicio utiliza la imagen de Docker *portainer/portainer-ce* para crear un contenedor del administrador de Docker *Portainer*. Se mapea el puerto 9000 y se utiliza un volumen para almacenar los datos. Se describe esta herramienta en el capítulo 2.4.6.

Al desplegar Docker Compose se levantan todos los servicios como se muestra en la figura 3.31.

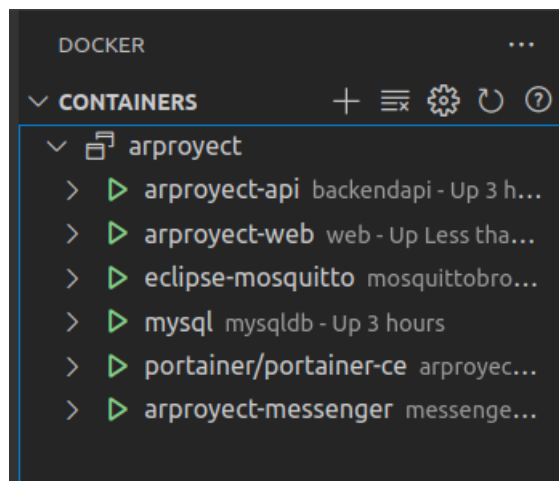


FIGURA 3.31. Contenedores de Docker.

3.9. Implementación de Hardware

A continuación se describe la implementación del *hardware* utilizado y detallado en el capítulo 2.7.

Para el manejo de la red local se configuró un *router* con una red Wi-Fi con filtrado de MAC Address y un servidor DHCP con entrega IP fija a los dispositivos del sistema.

En la figura 3.32 podemos observar el diagrama de red del sistema, detallando que tipo de conexión realiza cada dispositivo.

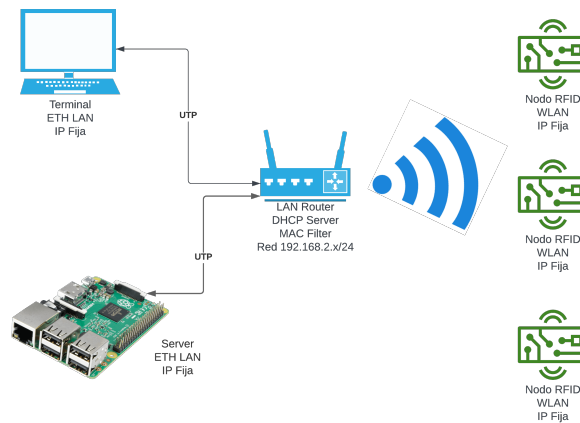


FIGURA 3.32. Diagrama de red del sistema.

En el servidor se implementó una placa Raspberry Pi 4. Para esta placa se imprimió un gabinete personalizado utilizando una impresora de tipo 3D, se incluyó además un espacio adicional en el gabinete para agregar un disco tipo SSD para ampliar las capacidades de la placa.

Para los nodos se utilizó el microcontrolador NodeMCU Esp32, el sensor de radiofrecuencia MRC522 y un buzzer sonoro. El diseño del circuito en un *software* especializado y la impresión de la placa PCB estuvo a cargo de colaboradores.

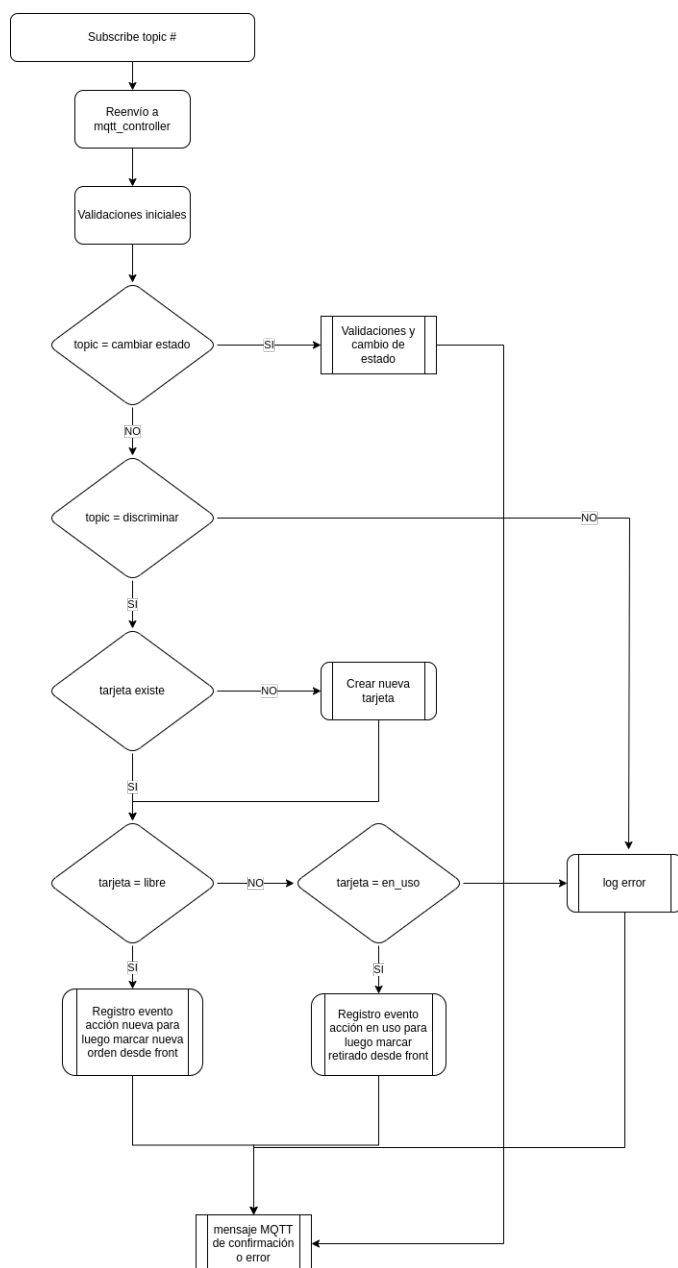


FIGURA 3.16. Flujo de código fuente para MQTT en API REST.

Capítulo 4

Ensayos y resultados

4.1. Pruebas funcionales del hardware

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

Capítulo 5

Conclusiones

5.1. Conclusiones generales

La idea de esta sección es resaltar cuáles son los principales aportes del trabajo realizado y cómo se podría continuar. Debe ser especialmente breve y concisa. Es buena idea usar un listado para enumerar los logros obtenidos.

Algunas preguntas que pueden servir para completar este capítulo:

- ¿Cuál es el grado de cumplimiento de los requerimientos?
- ¿Cuán fielmente se pudo seguir la planificación original (cronograma incluido)?
- ¿Se manifestó algunos de los riesgos identificados en la planificación? ¿Fue efectivo el plan de mitigación? ¿Se debió aplicar alguna otra acción no contemplada previamente?
- Si se debieron hacer modificaciones a lo planificado ¿Cuáles fueron las causas y los efectos?
- ¿Qué técnicas resultaron útiles para el desarrollo del proyecto y cuáles no tanto?

5.2. Próximos pasos

Acá se indica cómo se podría continuar el trabajo más adelante.

Bibliografía

- [1] Lautaro Regis. «Tornería». En: *Esc. de Ed. Tec. Nuestra Señora de la Guarda* (2018).
- [2] MecanicaFácil. *Rectificado del Bloque del Motor*.
http://www.mecanicafacil.info/Rectificado_del_Bloque_del_Motor.html.
Sep. de 2012. (Visitado 06-09-2022).
- [3] Gilda Liliana Ballivian Rosado. «Reparación del Motor». En: *Instituto de Educación Superior Tecnológico Público* (2016).
- [4] Robert Bosch. «Manual de la técnica automóvil». En: Editorial Reverte S.A., 1999, págs. 387-389.
- [5] William H. Crouse. *Mecánica de la Motocicleta*. Marcombo, 1992, pág. 321.
- [6] Ediciones Necochea. *Tapas de Cilindros*. Ediciones Necochea, 2021.
- [7] Dipole. ¿Qué es RFID? <https://www.dipolerfid.es/blog-rfid/que-es-rfid>.
(Visitado 06-09-2022).
- [8] Enciclopedia EcuRed. *Sistemas radioeléctricos*.
<https://www.ecured.cu/UHF>. (Visitado 06-09-2022).
- [9] Telectrónica. *Web oficial*. <https://telectronica.com/soluciones-rfid/>.
(Visitado 06-09-2022).
- [10] Cámara Argentina de IoT. *Documentos | Cámara Argentina de IoT*.
https://iot.org.ar/es_ES/category/documentos/. (Visitado 05-04-2023).
- [11] Mozilla. *Generalidades del protocolo HTTP*.
<https://developer.mozilla.org/es/docs/Web/HTTP/Overview>. (Visitado 06-09-2022).
- [12] Oscar Blancarte. *Arquitectura cliente-servidor*.
<https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/cliente-servidor>. (Visitado 06-09-2022).
- [13] Mozilla. *HTML: Lenguaje de etiquetas de hipertexto*.
<https://developer.mozilla.org/es/docs/Web/HTML>. (Visitado 06-09-2022).
- [14] Cloudflare. ¿Qué es el modelo OSI? <https://www.cloudflare.com/es-es/learning/ddos/glossary/open-systems-interconnection-model-osi/>.
(Visitado 06-09-2022).
- [15] Rfc. *Specification of internet transmission control program*.
<https://www.rfc-editor.org/rfc/rfc675>. (Visitado 06-09-2022).
- [16] Paessler. ¿Qué es MQTT?
<https://www.paessler.com/es/it-explained/mqtt>. (Visitado 06-09-2022).
- [17] Amazon. *Pub/Sub Messaging*.
<https://aws.amazon.com/es/pub-sub-messaging/>. (Visitado 06-09-2022).
- [18] Eclipse. *Eclipse Mosquitto™ An open source MQTT broker*.
<https://mosquitto.org/>. (Visitado 06-09-2022).
- [19] C. J. Date. *An Introduction to Database Systems*. 8th ed. Boston, MA: Addison-Wesley Professional, 2004.
- [20] Oracle Corporation. *MySQL :: MySQL Documentation*.
<https://dev.mysql.com/doc/>. (Visitado 30-03-2023).

- [21] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, CA: University of California, Irvine, 2000. URL: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> (visitado 27-03-2023).
- [22] Node.js contributors. *Node.js Documentation*. Visitado: 2023-03-27. 2021. URL: <https://nodejs.org/docs/latest-v14.x/api/>.
- [23] Express contributors. *Express Documentation*. Visitado: 2023-03-27. 2021. URL: <https://expressjs.com/>.
- [24] Sequelize contributors. *Sequelize Documentation*. Visitado: 2023-03-27. 2021. URL: <https://sequelize.org/master/>.
- [25] WhatsApp Inc. *WhatsApp FAQ - Creating a link with a pre-filled message*. Visitado: 2022-05-02. 2021. URL: <https://faq.whatsapp.com/general/chats/how-to-use-click-to-chat/?lang=en>.
- [26] Leifer Mendez. *api-whatsapp-ts*. <https://github.com/leifermendez/api-whatsapp-ts>. Visitado: 2023-05-03. 2021.
- [27] Portainer.io. *Portainer Documentation*. <https://documentation.portainer.io>. Visitado: May 4, 2023. 2021.
- [28] Docker contributors. *Docker Documentation*. Visitado: 2023-03-27. 2021. URL: <https://docs.docker.com/>.
- [29] W3C. *Cascading Style Sheets (CSS) specifications*. Visitado: 2023-03-27. 2018. URL: <https://www.w3.org/Style/CSS/>.
- [30] ECMA International. *ECMAScript® Language Specification*. Visitado: 2023-03-27. 2021. URL: <https://www.ecma-international.org/ecma-262/12.0/>.
- [31] World Wide Web Consortium. *Document Object Model (DOM) Specification*. <https://www.w3.org/TR/dom/>. (Visitado 06-09-2022).
- [32] Ionic contributors. *Ionic Framework Documentation*. Visitado: 2023-03-27. 2021. URL: <https://ionicframework.com/docs/>.
- [33] Angular contributors. *Angular Documentation*. Visitado: 2023-03-27. 2021. URL: <https://angular.io/docs>.
- [34] Facebook. *React Documentation*. Visitado: 2023-03-27. 2021. URL: <https://reactjs.org/docs/getting-started.html>.
- [35] Docker contributors. *Docker Compose Documentation*. Visitado: 2023-03-27. 2021. URL: <https://docs.docker.com/compose/>.
- [36] Nginx, Inc. *Nginx Documentation*. <https://docs.nginx.com/>. (Visitado 27-03-2023).
- [37] Raspberry Pi Foundation. *Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*. <https://www.raspberrypi.org/>. (Visitado 05-04-2023).
- [38] Espressif Systems. *ESP32 Datasheet*. 2021. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (visitado 12-10-2022).
- [39] *RFID Technology*. Impinj website. 2021. URL: <https://www.impinj.com/what-is-rfid>.
- [40] NXP Semiconductors. *MFRC522*. NXP Semiconductors. Eindhoven, The Netherlands, 2011. URL: <https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf>.
- [41] ArduinoModules. *KY-006 Passive Buzzer Module*. <https://arduinomodules.info/ky-006-passive-buzzer-module/>. Visitado: 2023-03-22. 2016.

- [42] Nick O’Leary. *PubSubClient - A client library for the MQTT protocol*. <https://github.com/knolleary/pubsubclient>. (Visitado 02-05-2023).
- [43] Martin Fowler. *Microservices*. 2014. URL: <https://martinfowler.com/articles/microservices.html> (visitado 03-05-2023).
- [44] QR Code. *QR Code*. Visitado: 2022-05-02. 2021. URL: <https://www.qrcode.com/en/>.
- [45] W3C. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. Visitado: 2022-05-02. 2011. URL: <https://www.w3.org/TR/SVG11/>.
- [46] Glenn Krasner y Stephen Pope. *A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80*. ParcPlace Systems. 1988. URL: https://www.researchgate.net/publication/221008215_A_Cookbook_for_Using_the_Model-View_Controller_User_Interface_Paradigm_in_Smalltalk-80.
- [47] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. URL: <https://martinfowler.com/articles/injection.html>.
- [48] Erich Gamma y col. *Observer*. 1995. URL: https://sourcemaking.com/design_patterns/observer.