Proyecto Fin de Grado
Ingeniería de Telecomunicación

Desarrollo de aplicación móvil para telelectura de contadores de agua

Autor: Fernando Rodríguez Mateo
Tutor: Jose Manuel Fornés Rumbao

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

Sevilla, 2021

Proyecto Fin de Grado
Ingeniería de Telecomunicación

Desarrollo de aplicación móvil para telelectura de contadores de agua

Autor:
Fernando Rodríguez Mateo

Tutor:
Jose Manuel Fornés Rumbao
Profesor titular

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Proyecto Fin de Carrera: Desarrollo de aplicación móvil para telelectura de contadores de agua

Autor: Fernando Rodríguez Mateo

Tutor: Jose Manuel Fornés Rumbao

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2013

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Quiero agradecer en estas líneas a todos los que me apoyan día a día y me han apoyado durante estos años para llegar aquí. A mi familia por estar siempre conmigo, confiar en mí, apoyarme y animarme siempre para seguir luchando ya que ellos saben cuánto he sufrido por llegar hasta aquí. A mis compañeros por ayudarme de muchas maneras en Sevilla, especialmente a Pablo, Manu y Fran, que sin su apoyo no habría podido seguir adelante en este proyecto. A mis amigos por hacer más ameno el día a día y haber estado en los buenos y en los malos momento. Por último, a mi tutor por dejarme hacer realidad este proyecto.

Fernando Rodríguez Mateo

Sevilla, 2021

Resumen

Actualmente se está produciendo un aumento considerable de dispositivos IoT que se disponen para mejorar el nivel de vida actual de las personas. Uno de los aspectos que actualmente se encuentran en desarrollo es el del control y mantenimiento de la cantidad de agua que se consumen en el hogar. Debido al cambio climático, es fundamental que se racione la cantidad de agua que consume la población. Además, este raciocinio ayuda a economizar el gasto que se genera del mismo, por lo que es una clara ventaja para la población.

Por otro lado, el aumento de dispositivos móviles que actualmente se utilizan aporta una clara ventaja al objetivo marcado anteriormente. Sin embargo, no existen muchas aplicaciones móviles destinadas al control y la telelectura de contadores de agua.

El objetivo de este proyecto es el desarrollo de una aplicación que permita la lectura de contadores de agua en el dispositivo móvil, permitiendo el manejo de cualquier contador de agua IoT gracias al servicio de una plataforma que permite integrar diferentes tipos de contadores y unificarlos, permitiendo una rápida accesibilidad y manejabilidad.

Abstract

There is currently a considerable increase in the number of IoT devices that are available to improve people's current standard of living. One of the aspects currently under development is the monitoring and maintenance of the amount of water consumed in the home. Due to climate change, it is essential to ration the amount of water consumed by the population. In addition, this rationing helps to economize the expenditure generated from it, so it is a clear advantage for the population.

On the other hand, the increase in the number of mobile devices currently in use brings a clear advantage to the objective outlined above. However, there are not many mobile applications for the control and remote reading of water meters.

The objective of this project is the development of an application that allows the reading of water meters on the mobile device, allowing the management of any IoT water meter thanks to the service of a platform that allows integrating different types of meters and unifying them, allowing quick accessibility and manageability.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

API	Interfaz de Programación de Aplicaciones (Application Programming Interface)
CoAP	Constrained Application Protocol
HTTP	Protocolo de transferencia de hipertexto (Hypertext Transfer Protocol)
IoT	Dispositivos de las Cosas (Internet of Things)
JSON	Notación de Objeto de JavaScript (JavaScript Object Notation)
MQTT	Protocolo de mensajería estándar de para IoT
REST	Transferencia de Estado Representacional (Representational State Transfer)
RPC	Llamada a procedimiento remoto (Remote Procedure Call)
SQL	Lenguaje de Consulta Estructurada (Structured Query Language)

1 INTRODUCCIÓN

No hay un sustituto para el trabajo duro.

- Thomas Edison -

A medida que se va expandiendo el alcance de los dispositivos de IoT, la demanda de los mismos también aumenta, lo que hace que se desarrollen tecnologías que permitan el acceso y control a dichos dispositivos. Actualmente, el avance de la tecnología IoT se encuentra centralizado en los suministros básicos (Agua, electricidad…), por lo que se espera que haya una gran cantidad de datos a gestionar. Debido a esto, se espera que las empresas desarrollen diferentes estrategias para abordarlo.

Para ello mostraremos cómo se encuentran los dispositivos IoT en general, así como el aumento de medidores digitales de agua que hay en el mercado.

1.1 Estado Actual

Debido al gran aumento de las tecnologías a nivel global, las empresas deben modernizarse para poder seguir siendo competitivos, esto se ha incrementado debido a la introducción de los dispositivos IoT.

El término IoT se ha acuñado en 1999 por Kevin Ashton, aunque no ha sido hasta el 2010 cuando se ha visto la utilidad a los dispositivos IoT. Desde entonces ha habido un aumento exponencial de la cantidad de dispositivos utilizados, llegando a tener más del 50% de los dispositivos conectados actualmente. [1]

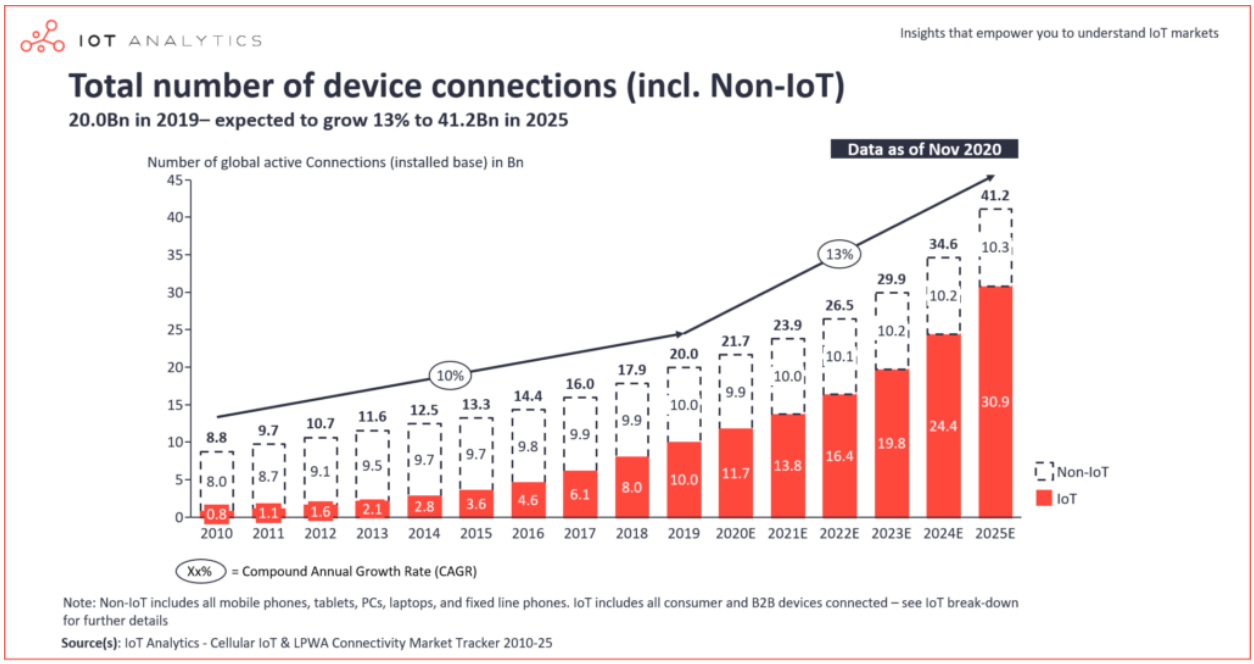

Figura 1-1. Número de dispositivos conectados

El aumento de dispositivos IoT ha sido aún más grande en el año 2020 debido a la pandemia COVID-19. [2] Ya que la mayoría de los trabajos se han tenido que realizar desde casa, se ha vuelto acuciante que se puedan manejar los dispositivos y aparatos externamente. Esto da lugar a implementar soluciones IoT.

Sin embargo, hoy en día la cantidad de dispositivos es menor que la previsión que se tenía. [3] Aún falta millones de dispositivos para conectar. Además, se debe tener en cuenta la conectividad de los aparatos conectados, así como su interoperabilidad.

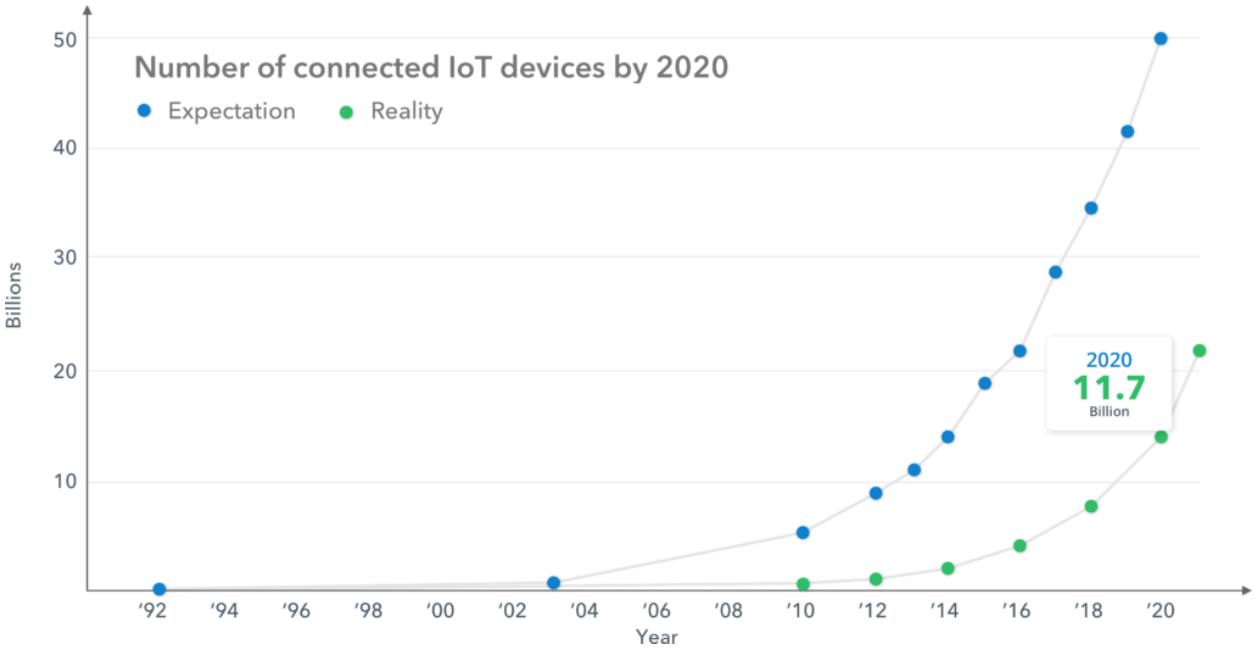

Figura 1-2. Evolución esperada frente a la real.

Este aumento provoca que los gobiernos y las empresas se apresuren por digitalizar elementos básicos como puedan ser el agua, la electricidad y el gas. Esto se realizará mediante la introducción de medidores inteligentes que utilizan la base IoT para el análisis y la gestión de los datos generados por estos medidores.

En 2026 se prevé que haya en torno a 400 millones de medidores inteligentes de agua instalados, esto proporciona un gran aumento en el mercado de este tipo de tecnologías.

Estos medidores servirán para mejorar tanto el consumo de agua a nivel nacional como el consumo de agua a nivel de usuario. Lo que permite a los gobiernos gestionar más eficientemente la cantidad de recursos hídricos que se consumen. [4]

A pesar del aumento de los dispositivos instalados, existen muy pocas aplicaciones para dispositivos móviles que puedan manejar la información recibida. Entre las más conocidas se encuentran hidroConta y GalileoIyS. Ambas aplicaciones sólo pueden manejar unos pocos dispositivos seleccionados por la propia empresa. De modo que este proyecto sirve para poner en el mercado una aplicación que pueda manejar cualquier medidor de agua inteligente.

2 OBJETIVOS

No hay un sustituto para el trabajo duro.

- Thomas Edison -

Como objetivo principal, este proyecto va a desarrollar una aplicación para dispositivos móviles que gestione las telelecturas de los contadores de agua conectados mediante IoT, que se pueda controlar el suministro de agua y que se pueda comprobar en tiempo real la cantidad de agua consumida por los usuarios.

Para ello, este proyecto va a seguir la siguiente estructura, que pondremos en diferentes capítulos:

- Estudiaremos las diferentes plataformas que permiten la integración de dispositivos IoT, centrándonos en la plataforma ThingsBoard, que es la que se va a emplear en este proyecto
- Mostraremos como se ha desarrollado la aplicación, de modo que mostraremos los diferentes aspectos de este y su utilización, de modo que se pueda utilizar de manera eficiente y correcta.
- Por último, mostraremos las conclusiones a las que se ha llegado al realizar el proyecto, mostrando aspectos de mejora y una valoración final.

La aplicación está construida mediante el software React Native, de modo que podamos crear una aplicación multiplataforma que permita ser utilizada en los diferentes sistemas operativos como Android, iOS, Google…

3 THINGSBOARD

No hay un sustituto para el trabajo duro.

- Thomas Edison -

ThingsBoard es una plataforma de gestión de dispositivos IoT de código abierto que aporta una solución local o en la nube de los diferentes proyectos de gestión IoT en la que habilita una infraestructura del lado del servidor para que sirva de apoyo en las aplicaciones desarrolladas.

Primero, en este apartado veremos las diferentes plataformas que existen en el mercado, de modo que podamos ver por encima las principales diferencias que poseen.

Luego, desarrollaremos la plataforma ThingsBoard, que es la plataforma que se va a realizar en este proyecto, mostrando las principales características, ventajas y desventajas. Mostraremos también las diferentes modalidades de la plataforma, para ver cuál es la que se va a utilizar.

3.1 Plataformas de Integración IoT

Para que distintos dispositivos de diferentes marcas puedan comunicarse con la aplicación, es necesario que los datos se recojan previamente en una plataforma de integración que reúna los datos y que lo presente de una manera uniforme para que pueda ser interpretado por la aplicación.

Aunque no es objeto de este proyecto la comparativa de las diferentes plataformas, se va a realizar una breve descripción de las posibilidades que ofrece el mercado. Después, nos centraremos en la plataforma ThingsBoard, que es la que se ha elegido para este proyecto.

Entre las plataformas que existen en el mercado, las más conocidas son: [5]

- **Amazon Web Services IoT**: Es una plataforma de pago que recopila los datos en la nube de Amazon Web Services y que facilita la conexión con otros dispositivos para facilitar la conexión con las aplicaciones que los consumen. Se conectan mediante HTTPS, WebSockets o MQTT.
- **ThingSpeak**: Plataforma gratis de código abierto, para poder comunicarse con la plataforma se utilizan las API REST proporcionada. El análisis de datos se utiliza mediante MATLAB.
- **Azure IoT Hub:** Plataforma de pago que administra la información de manera bidireccional entre los distintos dispositivos y el back-end de las distintas soluciones como pueden ser IoT Hub, WebApps, bases de datos y blobs de almacenamiento.
- **Oracle Internet of Things Cloud Service:** Plataforma de pago que recopila datos de forma segura y fiable desde cualquier dispositivo inteligente, analiza los datos en tiempo real mediante Big Data y permite utilizar las plataformas Oracle.

3.2 Plataforma ThingsBoard

Como se ha descrito anteriormente, ThingsBoard es una plataforma IoT de código abierto que está centrada en el rápido desarrollo, gestión y escalado de proyectos que se basan en la tecnología IoT. Es compatible con los protocolos de IoT más utilizados: MQTT, CoAP y HTTP.

Esto es debido a que utiliza un servidor de puerta de enlace que se encarga de la conexión con los dispositivos conectados a la red, permitiendo que la gestión sea ágil y se mantenga siempre actualizada. Gracias a esto,

consigue una gran tolerancia a los fallos y un buen rendimiento a la hora de gestionar y procesar los datos recolectados.

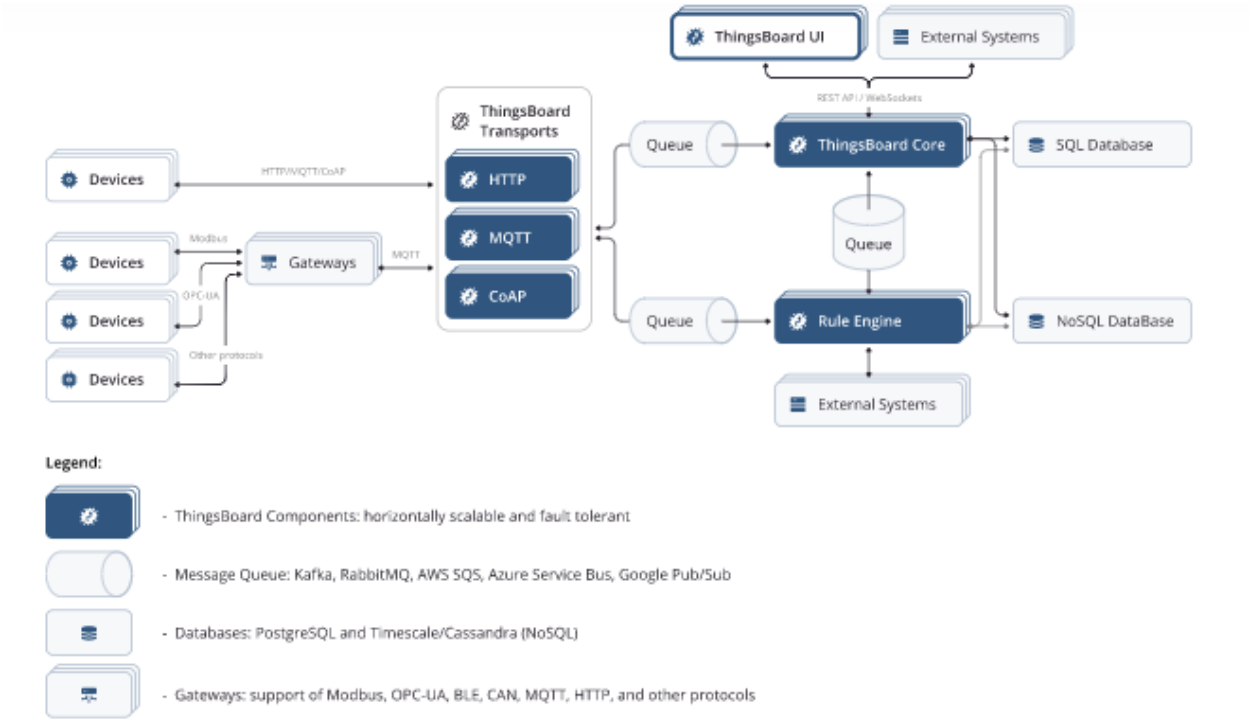

Figura 3-1. Arquitectura ThingsBoard

3.2.1 Características

La plataforma cuenta con una serie de características que debemos conocer para que podamos comprenderlo a fondo. Cada uno de ellos pensado para mejorar su usabilidad y utilidad, así como para simplificar el escalado de la infraestructura y adecuarla a las necesidades.

Con ThingsBoard se puede:

- Añadir dispositivos, activos y clientes, y definir las relaciones entre ellos.
- Recopilar y visualizar datos de dispositivos y activos.
- Analizar la telemetría entrante y activar alarmas con un procesamiento de eventos complejo.
- Controlar sus dispositivos mediante llamadas a procedimiento remoto (RPC).
- Crear flujos de trabajo basados en un evento del ciclo de vida del dispositivo, un evento de API REST, una solicitud de RPC, etc.
- Diseñar paneles de control dinámicos y receptivos y presentar información y telemetría de dispositivos o activos a sus clientes.
- Habilitar funciones específicas de casos de uso mediante cadenas de reglas personalizables.
- Enviar los datos del dispositivo a otros sistemas.

3.2.2 Arquitectura

Para profundizar más en la plataforma ThingsBoard, en este apartado vamos a estudiar la arquitectura básica en la que se compone la plataforma. De este modo, seremos capaces de saber cómo conectarnos a ella de modo que podamos extraer la información que se necesita para la aplicación.

3.2.2.1 Transporte ThingsBoard

ThingsBoard proporciona API basadas en MQTT, HTTP, CoAP y LwM2M que están disponibles para las aplicaciones / firmware de su dispositivo. Cada uno de los protocolos API es proporcionado de forma independiente y es parte de la "Capa de transporte" de ThingsBoard.

Una vez que el transporte recibe el mensaje del dispositivo, se analiza y se envía a la cola de mensajes. La entrega del mensaje al dispositivo se produce solo después de que la cola de mensajes reconozca el mensaje correspondiente.

3.2.2.2 Núcleo ThingsBoard

ThingsBoard Core es responsable de manejar las llamadas a la API REST y las suscripciones a WebSocket. También es responsable de almacenar información actualizada sobre las sesiones activas del dispositivo y monitorear el estado de conectividad del dispositivo. El núcleo ThingsBoard usa un sistema de actores bajo el capó para implementar actores para entidades principales: inquilinos y dispositivos.

3.2.2.3 Motor de reglas

El motor de reglas de ThingsBoard es el corazón del sistema y es responsable de procesar los mensajes entrantes. El motor de reglas implementa actores para las entidades principales: cadenas de reglas y nodos de reglas. Los nodos del motor de reglas son responsables de ciertas particiones de los mensajes entrantes.

El motor de reglas se suscribe a la alimentación de datos entrantes de la(s) cola(s) y reconoce el mensaje una vez que se procesa. Hay múltiples estrategias disponibles que controlan el orden o procesamiento del mensaje y los criterios de reconocimiento del mensaje.

El motor de reglas de ThingsBoard puede funcionar en dos modos: compartido y aislado. En el modo compartido, el motor de reglas procesa los mensajes que pertenecen a varios inquilinos. En modo aislado, el motor de reglas se puede configurar para procesar mensajes solo para inquilinos específicos.

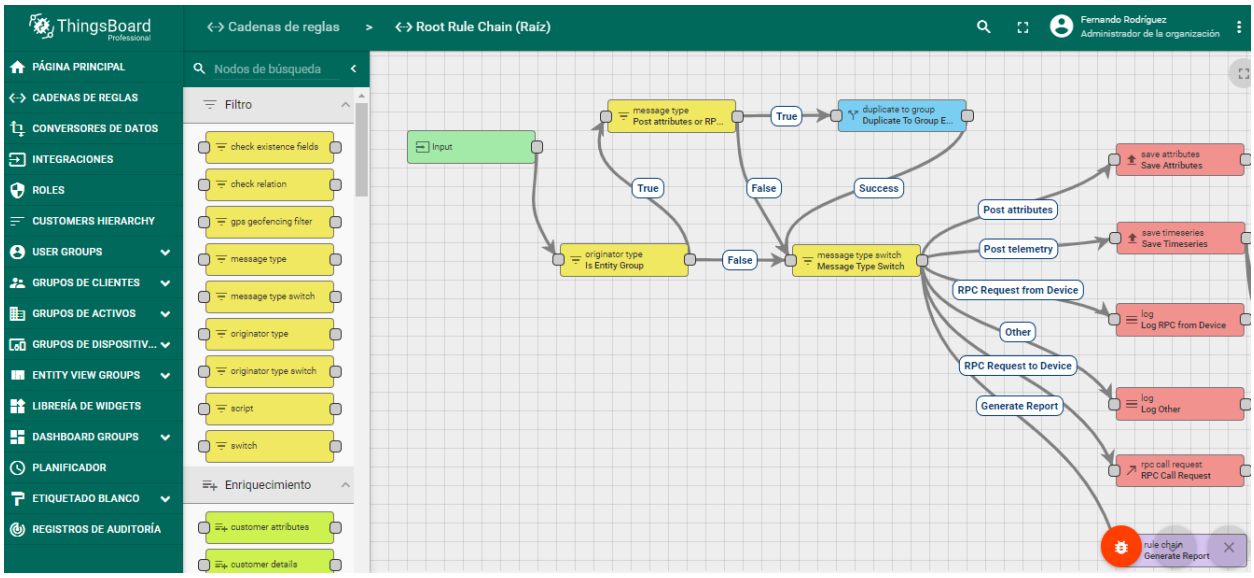

Figura 3-2. Motor de Reglas de ThingsBoard

3.2.2.4 Interfaz de Usuario de Red

ThingsBoard proporciona una interfaz que permite manejar de manera rápida y eficaz las diferentes utilidades de la plataforma. Además, nos permite visualizar claramente las diferentes entidades que se van creando, así como las acciones que se pueden ir tomando sobre ellos.

Una vez que se carga, la aplicación comienza a usar la API REST y la API WebSockets proporcionada por el núcleo de ThingsBoard.

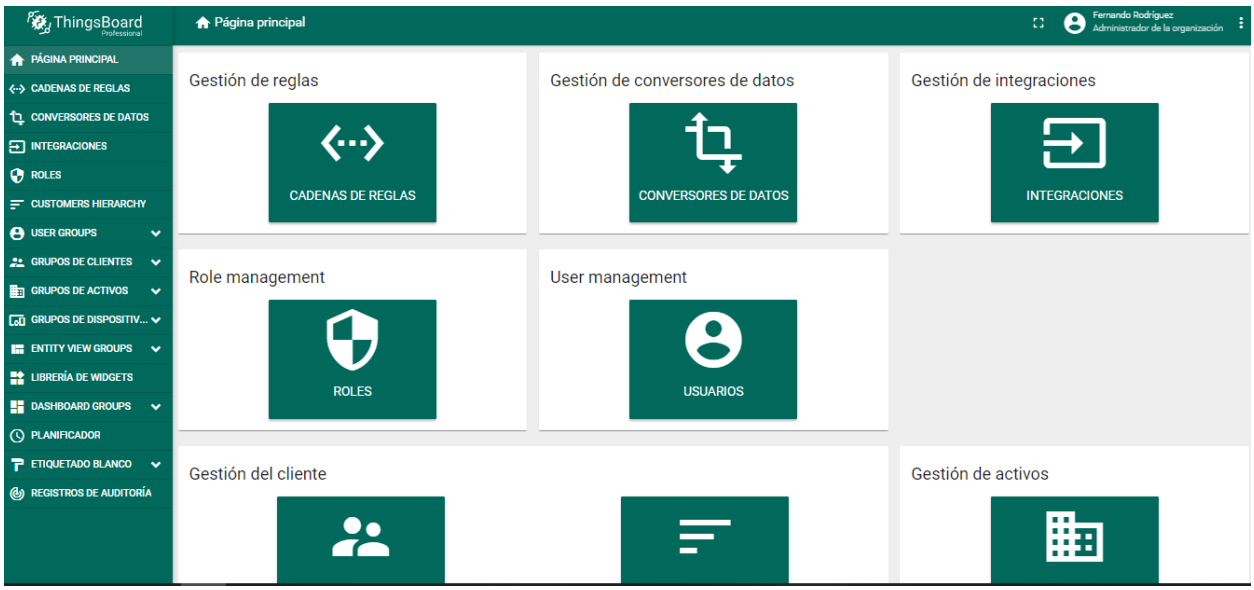

Figura 3-3. Interfaz de Web de ThingsBoard

3.2.2.5 Arquitectura monolítica vs microservicios

Al iniciar ThingsBoard, es posible ejecutar la plataforma como una aplicación monolítica o como un conjunto de microservicios.

Aproximadamente el 80% de las instalaciones de la plataforma todavía utilizan el modo monolítico debido a los esfuerzos mínimos del conocimiento y los recursos de hardware para realizar la configuración y el bajo mantenimiento.

Sin embargo, los microservicios son útiles si necesitamos alta disponibilidad o tenemos millones de dispositivos. Esto nos permite protegernos de:

- Picos de carga impredecibles;
- Mala configuración de la cadena de reglas;
- Dispositivos únicos que abren miles de conexiones simultáneas debido a errores de firmware.

3.2.2.6 Bases de datos

ThingsBoard utiliza una base de datos para almacenar entidades (dispositivos, activos, clientes, paneles de control, etc.) y datos de telemetría (atributos, lecturas de sensores de series temporales, estadísticas, eventos). La plataforma admite tres opciones de base de datos en este momento:

- SQL: almacena todas las entidades y la telemetría en la base de datos SQL. Los autores de ThingsBoard recomiendan usar PostgreSQL y esta es la base de datos SQL principal que admite ThingsBoard.
- NoSQL (obsoleto): almacena todas las entidades y la telemetría en la base de datos NoSQL. Los autores de ThingsBoard recomiendan usar Cassandra y esta es la única base de datos NoSQL que ThingsBoard admite en este momento.
- Híbrido (PostgreSQL + Cassandra): almacena todas las entidades en la base de datos PostgreSQL y los datos de series temporales en la base de datos Cassandra.
- Híbrido (PostgreSQL + TimescaleDB): almacena todas las entidades en la base de datos PostgreSQL y los datos de series temporales en la base de datos Timescale.

3.2.3 Conceptos clave

ThingsBoard posee una serie de elementos que son fundamentales para la comprensión de la plataforma y que en este apartado vamos a desglosar para su mayor comprensión. Estos conceptos sirven para que la aplicación se comunique con el servidor y nos ayude a diseñar la estructura de esta.

3.2.3.1 Entidades y Relaciones

ThingsBoard provee a la interfaz de usuario y a las API REST administrar múltiples tipos de entidades y sus relaciones en su aplicación de IoT. Las entidades admitidas son:

- Tenant: Es un individuo u organización que posee o produce dispositivos y activos. El tenant puede tener varios usuarios administradores de tenant y millones de clientes, dispositivos y activos;
- Clientes: El cliente también es una entidad comercial separada, individuo u organización que compra o usa dispositivos y/o activos del tenant; El cliente puede tener varios usuarios y millones de dispositivos y/o activos;
- Usuarios: los usuarios pueden navegar por paneles y administrar entidades;
- Dispositivos: entidades básicas de IoT que pueden producir datos de telemetría y manejar comandos RPC. Por ejemplo, sensores, actuadores, interruptores;
- Activos: entidades abstractas de IoT que pueden estar relacionadas con otros dispositivos y activos. Por ejemplo, fábrica, campo, vehículo;
- Vistas de entidad: útil si desea compartir solo una parte de los datos del dispositivo o de los activos con los clientes;
- Alarmas: eventos que identifican problemas con sus activos, dispositivos u otras entidades;
- Paneles de control: visualización de sus datos de IoT y capacidad para controlar dispositivos particulares a través de la interfaz de usuario;
- Nodo de reglas: unidades de procesamiento para mensajes entrantes, eventos del ciclo de vida de la entidad, etc.
- Cadena de reglas: define el flujo del procesamiento en el motor de reglas. Puede contener muchos nodos de reglas y enlaces a otras cadenas de reglas.

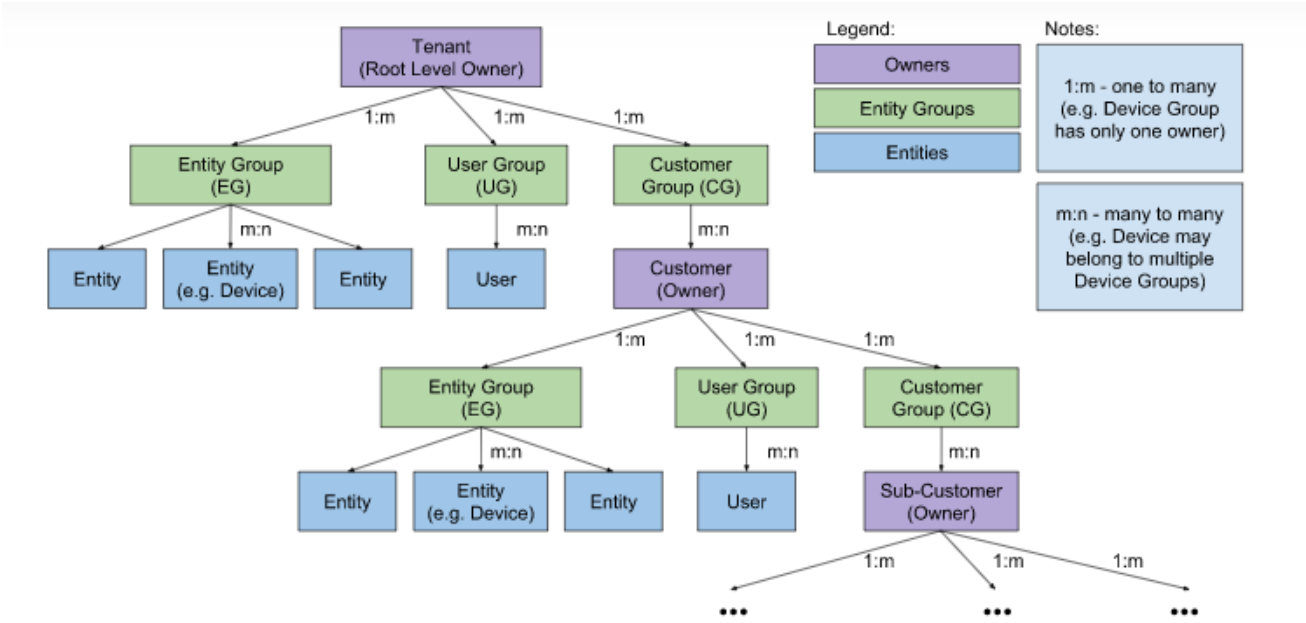

Figura 3-4. Estructura de entidades

Cada entidad posee varias características, que son las siguientes:

- Atributos: Son conceptos clave-valor estáticos y semiestáticos que describen características de las entidades. Por ejemplo, el número de serie, modelo, firmware…
- Datos de serie de tiempos (Timeseries): Datos asociados a un tiempo concreto disponibles para la visualización, consulta y almacenamiento. Por ejemplo, temperatura, humedad…
- Relaciones: Conexiones entre dos entidades, pueden ser de producir, administrar, poseer…

A continuación, vamos a observar las relaciones. Las relaciones se definen entre dos entidades de este tenant, de modo que tienen un tipo arbitrario que puede ser contiene, soporta, gestiona… Las relaciones de ThingsBoard son del tipo direccionales, similar a las relaciones de la programación orientada a objetos.

Las relaciones ayudan a entender las relaciones de las entidades del mundo físico en ThingsBoard. En el siguiente diagrama se muestran las posibles relaciones.

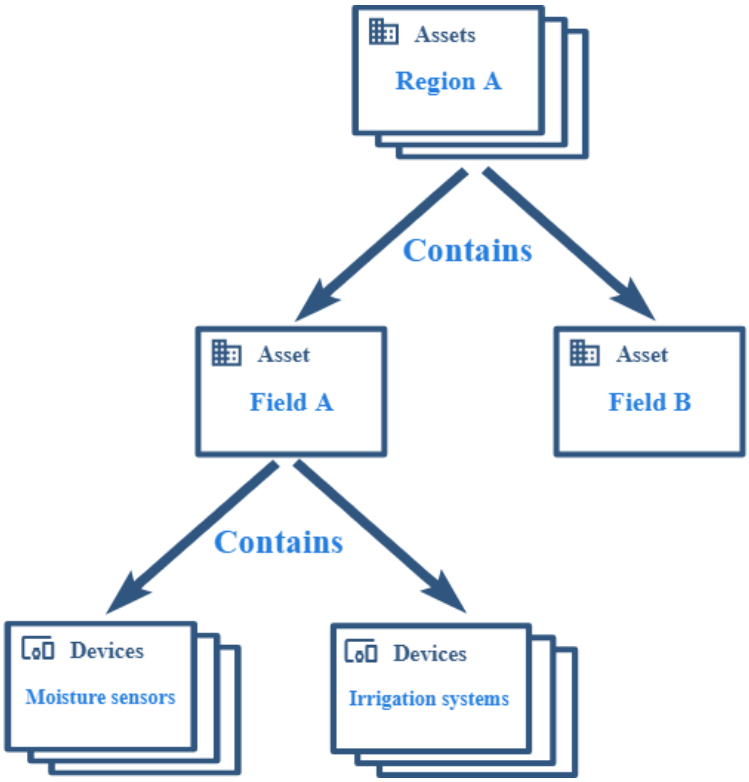

Figura 3-5. Posibles Relaciones de entidades

3.2.3.2 Atributos

ThingsBoard permite asignar atributos personalizables a sus entidades y trabajar con ellos. Estos atributos se pueden almacenar en la base de datos para que se puedan utilizar en el procesamiento de datos o visualizar más adelante.

Se muestran mediante un par de clave-valor. Este formato permite una mayor flexibilidad e integración de los dispositivos IoT existentes en el mercado. La clave es siempre una cadena y es básicamente un nombre de atributo, mientras que el valor del atributo puede ser cadena, booleano, doble, entero o JSON.

Los nombres de los atributos son totalmente personalizables, sin embargo, ThingsBoard recomienda utilizar el formato CamelCase ya que permite una mayor facilidad de integración a las funciones JavaScript.

En ThingsBoard existen tres tipos de atributos.

3.2.3.2.1 Atributos del lado del servidor

Es compatible con cualquier entidad que tiene la plataforma (clientes, usuarios, dispositivos, activos…). Son las que se pueden configurar a través de la interfaz del usuario o de las API. Sin embargo, el dispositivo no

puede acceder al atributo.

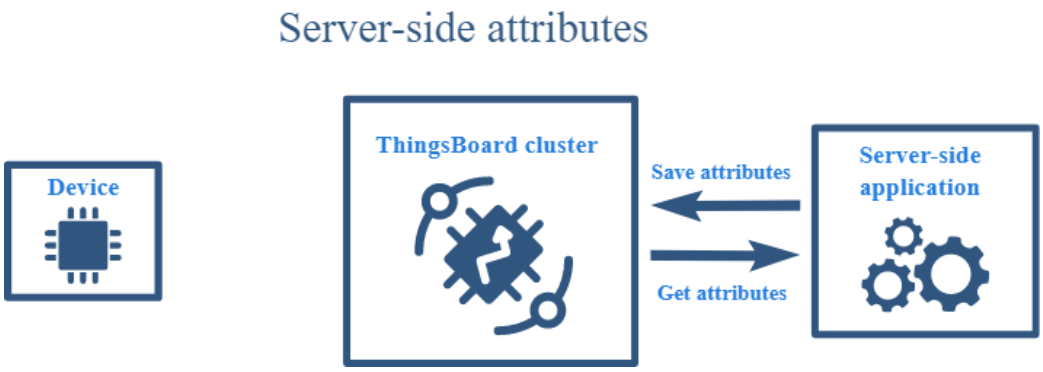

Figura 3-6. Atributos del lado del servidor

3.2.3.2.2 Atributos compartidos

Este tipo de atributos solo está disponible para los dispositivos. Es similar a los atributos del lado del servidor, pero el dispositivo puede solicitar el valor del atributo o suscribirse a las actualizaciones de este. Los dispositivos que se comunican mediante MQTT u otros protocolos de comunicación bidireccionales pueden suscribirse a las actualizaciones del atributo en tiempo real. Mientras, los dispositivos con HTTP u otros protocolos de comunicación solicitud-respuesta pueden pedir periódicamente el valor del atributo.

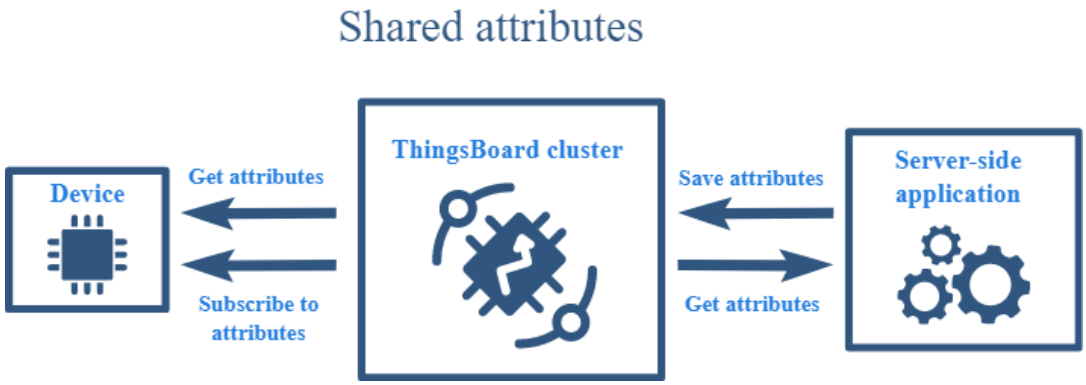

Figura 3-7. Atributos compartidos

3.2.3.2.3 Atributos del lado del cliente

Estos atributos solo están disponibles para los dispositivos, se utiliza para informar de varios datos semiestáticos desde el dispositivo a ThingsBoard. Es similar a los atributos compartidos, pero el dispositivo puede enviar valores a la plataforma.

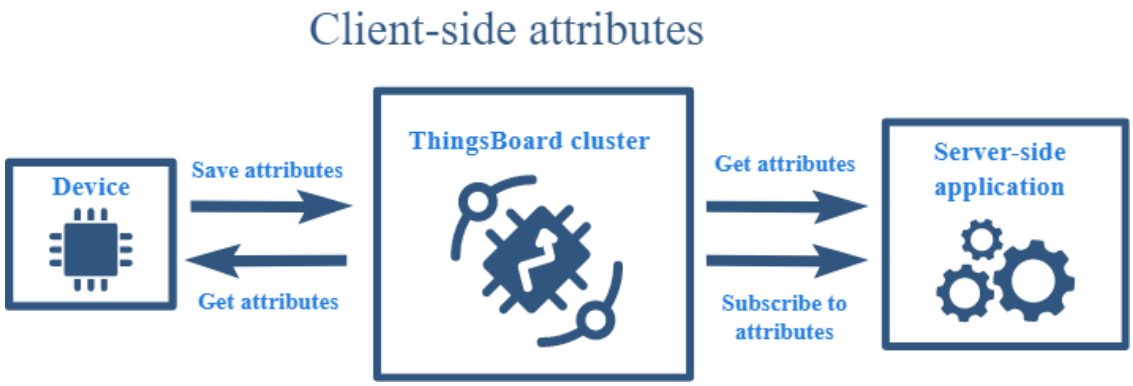

Figura 3-8. Atributos del lado del cliente

3.2.3.3 Datos de serie de tiempos (Timeseries)

ThingsBoard trabaja con datos de timeseries como pares clave-valor con marca de tiempo. Como los atributos descritos en el apartado anterior, las claves son cadenas de caracteres mientras que el valor puede ser una cadena de caracteres, booleano, doble, entero o JSON.

ThingsBoard proporciona una amplia gama de funciones relacionadas como pueden ser:

- Recopilar los datos de los dispositivos mediante varios protocolos e integraciones.
- Almacenar los datos en bases de datos SQL y NoSQL.
- Consultar el dato más reciente o consultar los datos en un rango de tiempo más específico.
- Visualizar los datos en dashboards configurables y personalizables.
- Filtrar y analizar los datos mediante el motor de reglas.
- Generar alarmas basadas en datos recopilados
- Reenviar datos a sistemas externos.

3.2.3.4 Alarmas

La plataforma permite crear y administrar alarmas relacionadas con sus entidades (dispositivos, activos, clientes…). Se puede configurar, por ejemplo, para que salte cuando cierto sensor esté por encima de cierto umbral. Este es un caso muy sencillo, pero puede llegar a ser mucho más complejo.

Las alarmas tienen varias características principales que son las siguientes:

3.2.3.4.1 Autor

El originador de la alarma es aquella entidad que la provoca.

3.2.3.4.2 Tipo

El tipo de alarma ayuda a identificar cuál es la causa de que la alarma salte.

3.2.3.4.3 Gravedad

Cada alarma tiene una gravedad que puede ser Crítica, Mayor, Menor, Advertencia o Indeterminada (clasificadas por prioridad en orden descendente).

3.2.3.4.4 Ciclo de Vida

La alarma puede estar activada o borrada. Cuando ThingsBoard crea la alarma, persiste la hora de inicio y finalización de esta. De forma predeterminada, la hora de inicio y la hora de finalización son las mismas. Si la condición de activación de la alarma se repite, la plataforma actualiza la hora de finalización. ThingsBoard puede borrar automáticamente la alarma cuando ocurre un evento que coincide con una condición de borrado de alarma. Un usuario puede borrar la alarma manualmente.

Además del estado de alarma activa y borrada, ThingsBoard también realiza un seguimiento de si alguien ha reconocido la alarma.

Para resumir, hay 4 valores posibles del campo "estado":

- Activa no reconocida (ACTIVE_UNACK): la alarma no se borra y aún no se reconoce.
- Activo reconocido (ACTIVE_ACK): la alarma no se borra, pero ya se reconoció.
- Borrado no reconocido (CLEARED_UNACK): la alarma ya se borró, pero aún no se reconoció.
- Borrado reconocido (CLEARED_ACK) - la alarma ya fue borrada y reconocida.

3.2.3.4.5 Unicidad de la alarma

ThingsBoard reconoce la alarma usando una combinación del autor, tipo y la hora de inicio de esta. Por lo tanto, solo puede haber una alarma con el mismo autor, tipo y hora.

3.2.3.4.6 Propagación de la alarma

ThingsBoard admite la propagación de la alarma. Cuando se crea la alarma, podemos especificar si debe ser visible para las entidades principales o no. También podemos especificar opcionalmente las relaciones que deben estar presentes entre las entidades padre y el autor para que se propague la alarma.

3.2.4 Motor de reglas

Es un marco de ThingsBoard para crear flujos de trabajos basados en eventos. Con el motor de reglas, podemos filtrar, enriquecer y transformar los mensajes entrantes originados por dispositivos de IoT y activos relacionados. También podemos activar varias acciones, por ejemplo, notificaciones o comunicación con sistemas externos. Se compone de tres elementos principales.

- Mensaje: Cualquier evento entrante. Puede ser un dato de dispositivo, eventos de ciclo de vida del dispositivo, eventos de API REST, solicitud de RPC, etc.
- Nodo de reglas: una función que se ejecuta en un mensaje entrante. Hay muchos tipos de nodos diferentes que veremos más adelante.
- Cadena de reglas: los nodos están unidos entre sí mediante relaciones, por lo que el mensaje saliente del nodo de reglas se envía a los siguientes nodos conectados.

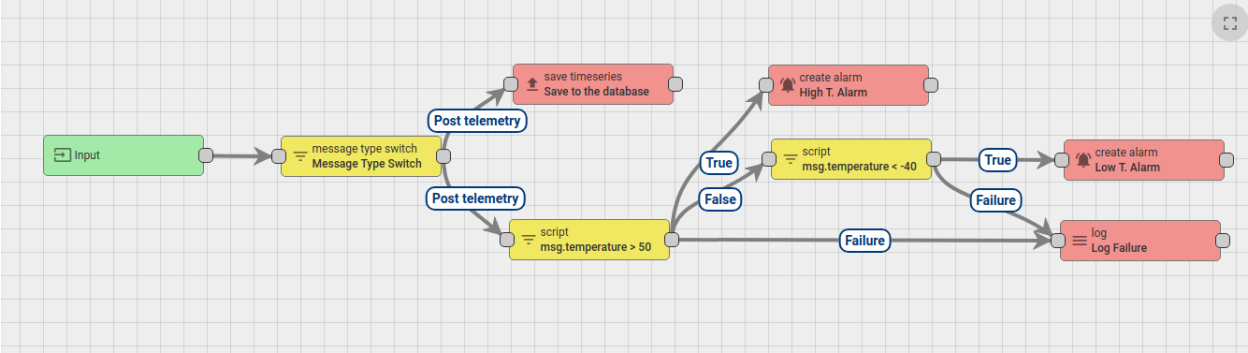

Figura 3-9. Cadena de reglas

El motor de reglas de ThingsBoard es un marco altamente personalizable. Algunos casos de uso comunes que se pueden configurar a través de las cadenas de reglas son:

- Validar y modificar datos para la telemetría o los atributos antes de guardarlos en la base de datos.
- Copiar la telemetría o los atributos de los dispositivos a los activos relacionados para que pueda agregar telemetría.
- Crear / actualizar / borrar alarmas según las condiciones definidas.
- Activar acciones basadas en eventos del ciclo de vida del dispositivo.
- Cargar los datos adicionales necesarios para el procesamiento.
- Activar llamadas a la API REST a sistemas externos.
- Enviar correos electrónicos cuando ocurra un evento.
- Tener en cuenta las preferencias del usuario durante el procesamiento del evento.

3.2.4.1 Mensaje

El mensaje del motor de reglas es una estructura de datos que puede representar varios mensajes en el sistema. Contiene la siguiente información:

- Id del mensaje: Identificador único universal basado en el tiempo.
- Originador del mensaje: Entidad que produce el mensaje (Dispositivo, Activo u otra entidad).

- Tipo de mensaje: Característica del mensaje que representa la acción de este.
- Carga útil del mensaje: Cuerpo JSON con carga útil real del mensaje.
- Metadatos: lista de pares clave-valor con datos adicionales sobre el mensaje.

Los diferentes tipos de los mensajes se muestran en la siguiente tabla:

Tabla 3-1. Tipos de mensajes

Tipo de Mensaje	Nombre mostrado	Descripción
POST_ATTRIBUTES_REQUEST	Atributos de publicación	Solicitud del dispositivo para publicar atributos del lado del cliente
POST_TELEMETRY_REQUEST	Publicar telemetría	Solicitud del dispositivo para publicar telemetría
TO_SERVER_RPC_REQUEST	Solicitud de RPC desde el dispositivo	Solicitud de RPC desde el dispositivo
RPC_CALL_FROM_SERVER_TO_DEVICE	Solicitud de RPC al dispositivo	Solicitud de RPC del servidor al dispositivo
ACTIVITY_EVENT	Evento de actividad	Evento que indica que el dispositivo se activa
INACTIVITY_EVENT	Evento de inactividad	Evento que indica que el dispositivo se vuelve inactivo
CONNECT_EVENT	Conectar evento	Evento producido cuando el dispositivo está conectado
DISCONNECT_EVENT	Desconectar evento	Evento producido cuando se desconecta el dispositivo
ENTITY_CREATED	Entidad creada	Evento producido cuando se creó una nueva entidad en el sistema
ENTITY_UPDATED	Entidad actualizada	Evento producido cuando se actualizó la entidad existente
ENTITY_DELETED	Entidad eliminada	Evento producido cuando se eliminó la entidad existente
ENTITY_ASSIGNED	Entidad asignada	Evento producido cuando la entidad existente fue asignada al cliente
ENTITY_UNASSIGNED	Entidad sin asignar	Evento producido cuando la entidad existente no se asignó al cliente
ADDED_TO_ENTITY_GROUP	Agregado al grupo	Evento producido cuando la entidad se agregó al Grupo de entidades
REMOVED_FROM_ENTITY_GROUP	Eliminado del grupo	Evento producido cuando la entidad se eliminó del Grupo de entidades
ATTRIBUTES_UPDATED	Atributos actualizados	Evento producido cuando se realizó la actualización de los atributos de la entidad
ATTRIBUTES_DELETED	Atributos eliminados	Evento producido cuando se eliminaron algunos de los atributos de la entidad
ALARM	Evento de alarma	Evento producido cuando se creó, actualizó o eliminó una alarma

3.2.4.2 Nodo de regla

El nodo de regla es un componente básico que procesa un solo mensaje entrante a la vez y produce uno o más

mensajes de salida. Es una unidad lógica básica que puede filtrar, transformar o añadir mensajes entrantes, realizar acciones o comunicarse con sistemas externos.

Pueden estar relacionados con otros nodos de reglas. Cada relación tiene un tipo, una etiqueta que se usa para identificar el significado lógico de la relación. Cuando el nodo de reglas produce el mensaje saliente, siempre se especifica el tipo de relación que se usa para enrutar el mensaje a los siguientes nodos.

Las relaciones típicas de los nodos son "Éxito" y "Fallo". Los nodos de reglas que representan operaciones lógicas utilizan "Verdadero" o "Falso". Algunos nodos de reglas específicos pueden usar tipos de relación más complejos.

Los diferentes nodos de reglas se agrupan de acuerdo con su naturaleza:

- Los nodos de filtro: se utilizan para el filtrado y enrutamiento de mensajes.
- Los nodos de enriquecimiento: se utilizan para cambiar campos de mensajes como el originador, tipo, carga útil, metadatos.
- Los nodos de acción: ejecutan acciones basados en el mensaje entrante.
- Los nodos externos: se utilizan para interactuar con sistemas externos.

3.2.4.3 Cadena de reglas

La cadena de reglas es un grupo lógico de nodos de reglas y sus relaciones. El tenant administrador puede definir una cadena de reglas raíz y, opcionalmente varias cadenas de reglas adicionales. La cadena de regla raíz maneja todos los mensajes entrantes y puede reenviarlas a otras cadenas para procesamiento adicional. Las otras cadenas también pueden reenviar mensajes.

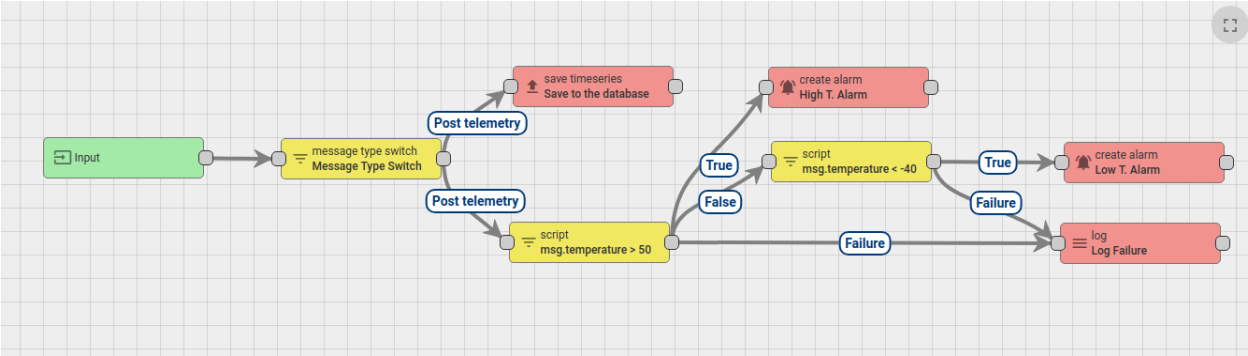

Figura 3-10. Cadena de reglas

Hay tres posibles resultados del procesamiento de mensajes en una cadena de reglas: éxito, error y tiempo de espera. El intento de procesamiento de mensajes sale como "Éxito" si el nodo ha procesado correctamente el mensaje. Si no lo ha procesado correctamente se produce un "Error" y no hay nodo para manejar este error. Si el procesamiento sobrepasa un cierto umbral configurable.

3.2.5 Dashboards

ThingsBoard brinda la capacidad de crear y administrar cuadros de mando, estos cuadros de mando posibilitan al usuario mostrar de manera eficaz y visual las diferentes entidades y los datos asociados a estos. Los diferentes paneles se asignan a los diferentes clientes.

Para crear un panel, el usuario puede añadir diferentes tipos de widgets de distintos tipos que muestran los diferentes datos asociados a las entidades que se quieran mostrar. Además, es posible exportar un tablero o un widget específico como un archivo de configuración en formato JSON. Puede utilizar este archivo para transferir la configuración de su panel o widget a otra instancia.

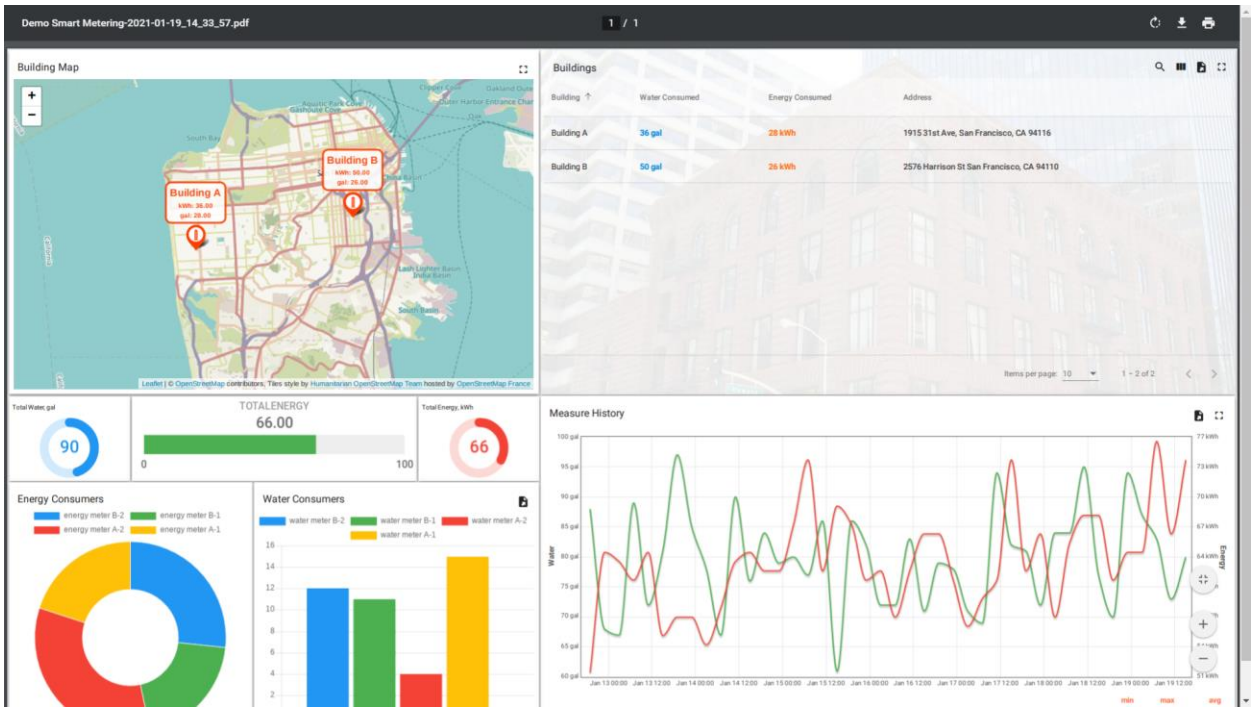

Figura 3-11. Dashboard

Para determinar que entidades se muestran en el panel, ThingsBoard configura diferentes alias que sirven como una referencia a las entidades que se pueden mostrar. Estos alias pueden ser estáticas o dinámicas.

Un ejemplo de alias estático es el alias de entidad única. Una entidad se configura una vez en el cuadro de diálogo de alias. Todos los usuarios ven los mismos datos si tienen permiso para acceder a este dispositivo.

Un ejemplo de un alias dinámico es el alias de tipo de dispositivo, que muestra todos los dispositivos de un tipo determinado. Este alias es dinámico porque la lista de dispositivos depende del usuario que usa el panel. Si ha iniciado sesión como tenant administrador, este alias se resolverá en todos los dispositivos de ese tipo. Sin embargo, si ha iniciado sesión como usuario del Cliente, este alias se resolverá en los dispositivos que sean asignados/sean propiedad de ese Cliente.

3.2.5.1 Widgets

Todos los paneles se construyen utilizando widgets de ThingsBoard que se muestran en la biblioteca. Cada widget proporciona diferentes funciones para el usuario final, como visualización de datos, control remoto de dispositivos, gestión de alarmas y visualización de contenido.

Hay cinco tipos de widgets:

- Los widgets de timeseries muestran datos para una ventana de tiempo específica. La ventana de tiempo puede ser en tiempo real (por ejemplo, durante las últimas 24 horas) o histórica (diciembre de 2020).
- Los widgets de valores más recientes muestran los valores más recientes de un atributo particular o claves de timeseries.
- Los widgets de control le permiten enviar comandos RPC a sus dispositivos.
- Los widgets de alarma le permiten mostrar alarmas.
- Los widgets estáticos están diseñados para mostrar datos estáticos.

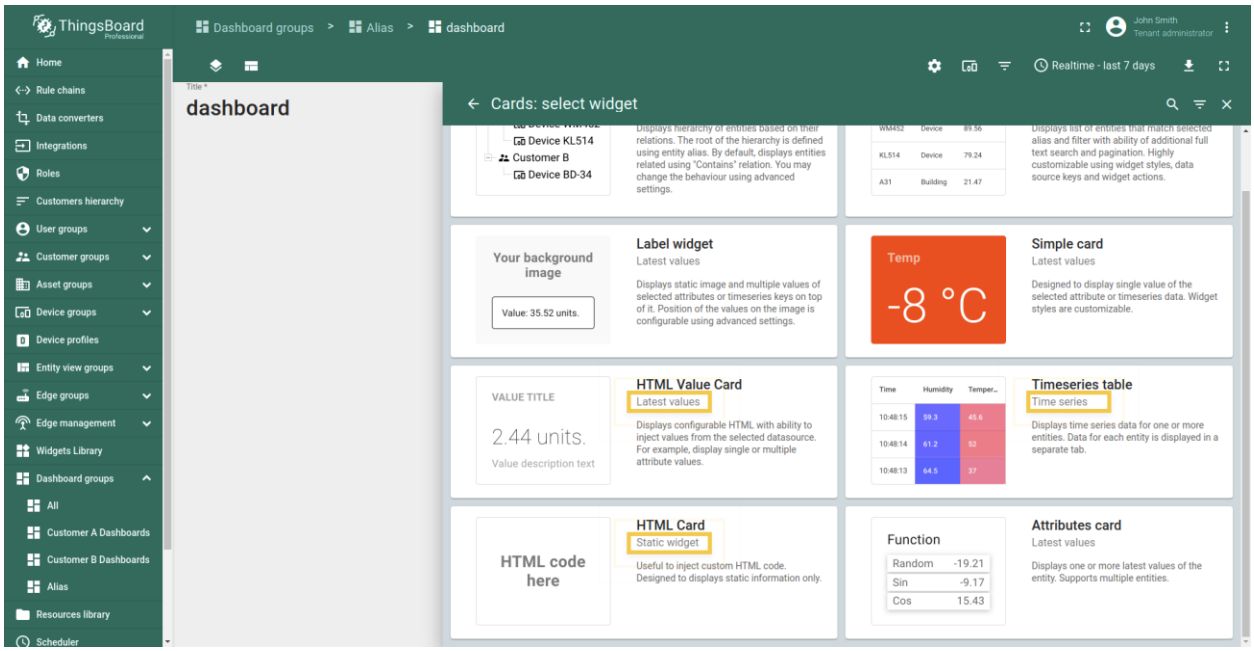

Figura 3-12. Biblioteca de Widgets

3.2.6 API REST

La API REST de ThingsBoard se puede explorar mediante la interfaz de usuario de Swagger. Puede explorar la API REST del servidor usando la siguiente URL: http://207.180.221.115:8080/swagger-ui.html#/, que es la dirección IP proporcionada por el profesor.

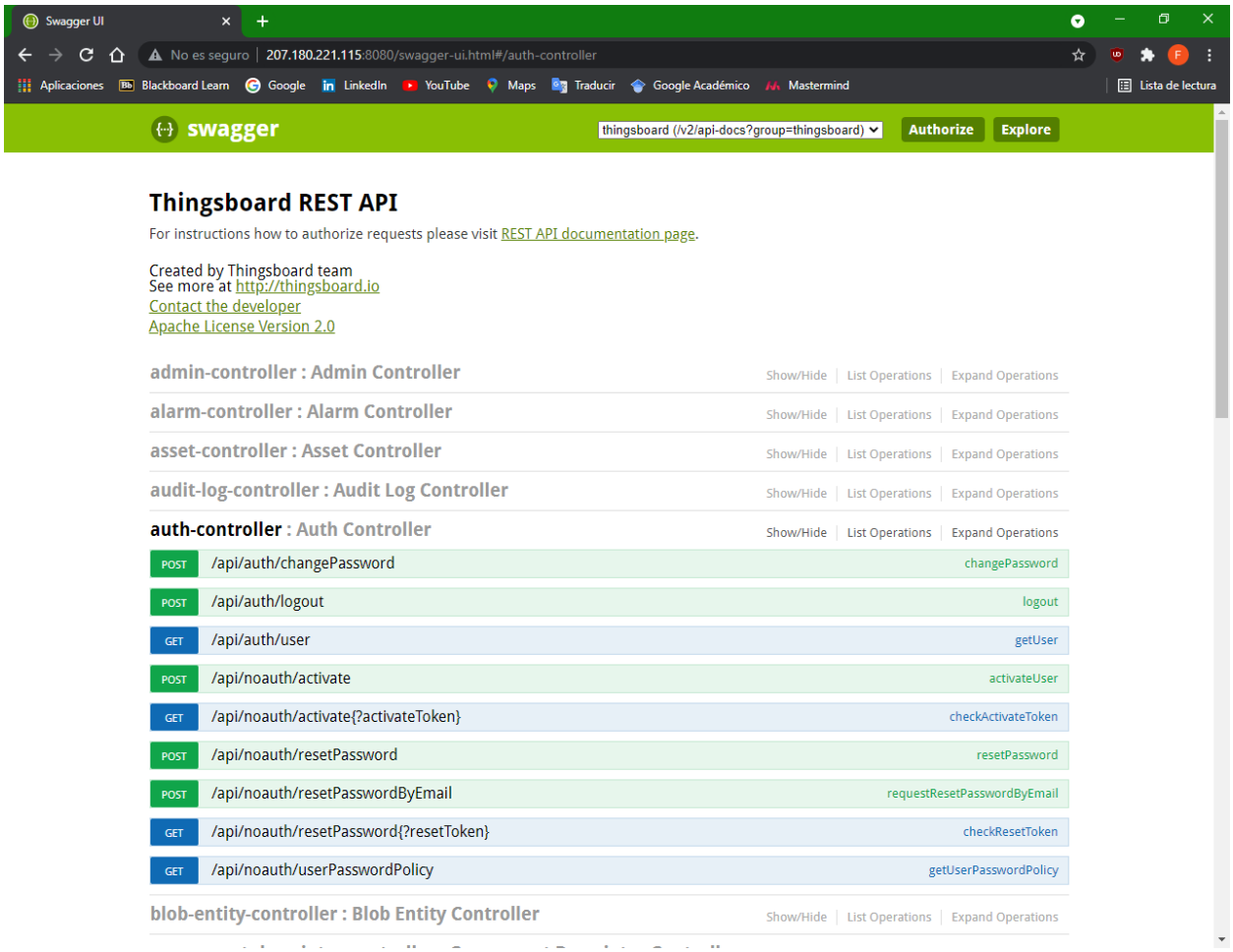

Figura 3-13. API REST de ThingsBoard

ThingsBoard utiliza JWT para la autenticación de solicitudes. Para que las solicitudes a la API sean correctas deberemos completar el encabezado "X-Authoritation". Para solicitar el código JWT debemos solicitar las credenciales mediante la API.

API login:

```
http://THINGSBOARD_URL/api/auth/login'
```

Donde THINGSBOARD_URL es la dirección IP dada por el profesor 207.180.221.115:8080. Para recibir el token JWT, tenemos que añadir a la API mencionada los datos del email del usuario y la contraseña correspondiente. Recibiremos un JSON con el token correspondiente que añadiremos a las posteriores solicitudes para autorizar la solicitud.

3.2.6.1 Atributos

Una vez que ya estamos autorizados, podemos solicitar o cambiar los atributos de las distintas entidades mediante las distintas APIs:

API Atributos del servidor:

```
http://207.180.221.115:8080/api/plugins/telemetry/$ENTITY_TYPE/$ENTITY_ID/SERVER_SCOPE
```

API Atributos compartidos:

```
http://207.180.221.115:8080/api/plugins/telemetry/$ENTITY_TYPE/$ENTITY_ID/SHARED_SCOPE
```

API Atributo del lado cliente:

```
http://207.180.221.115:8080/api/plugins/telemetry/$ENTITY_TYPE/$ENTITY_ID/CLIENT_SCOPE
```

Donde se puede realizar la solicitud POST enviando un JSON del atributo o realizar un GET, donde ThingsBoard te envía un JSON con los clave-valor de los distintos atributos solicitados. Se pueden realizar en todos los casos excepto en el del cliente, donde sólo se puede realizar la solicitud GET.

En la solicitud tendremos que poner el ENTITY_TYPE que corresponde al tipo de entidad del cual queremos sacar los atributos y el ENTITY_ID que es el identificador único de la entidad que se va a comprobar.

3.2.6.2 Timeseries

ThingsBoard proporciona una serie de APIs para obtener los diferentes datos de una entidad. Podemos obtener todos los datos de la entidad deseada u obtener los datos concretos de una clave específica que queramos manejar. Además, nos permite obtener el último valor obtenido o los diferentes valores históricos de esa clave.

3.2.6.2.1 Obtener claves de datos de timeseries para una entidad específica

Puede obtener una lista de todas las claves de datos de timeseries para un tipo de entidad en particular y un ID de entidad usando la solicitud GET a la siguiente API.

API solicitado:

```
http(s)://207.180.221.115:8080/api/plugins/telemetry/{entityType}/{entityId}/keys/timeseries
```

Los tipos de entidad admitidos son: TENANT, CUSTOMER, USER, DASHBOARD, ASSET, DEVICE, ALARM, ENTITY_VIEW.

3.2.6.2.2 Obtener valores de datos de timeseries para una entidad específica

Puede obtener una lista de todos los valores más recientes para un tipo de entidad en particular y un ID de entidad usando la solicitud GET a la siguiente API.

API solicitado:

```
http(s)://207.180.221.115:8080/api/plugins/telemetry/{entityType}/{entityId}/values/timeseries?keys=key1,key2,key3
```

Los tipos de entidad admitidos son: TENANT, CUSTOMER, USER, DASHBOARD, ASSET, DEVICE, ALARM, ENTITY_VIEW.

3.2.6.2.3 Obtener valores de datos históricos de timeseries para una entidad específica

Puede obtener una lista de todos los valores históricos para el tipo de entidad en particular y la identificación de la entidad usando la solicitud GET a la siguiente API.

API solicitado:

```
http(s)://207.180.221.115:8080/values/timeseries?keys=key1,key2,key3&startTs=1479735870785&endTs=1479735871858&interval=60000&limit=100&agg
```

```
=AVG
```

Los parámetros admitidos se describen a continuación:

- keys: Lista separada por comas de claves de telemetría para recuperar.
- startTs: Marca de tiempo de Unix que identifica el inicio del intervalo en milisegundos.
- endTs: Marca de tiempo de Unix que identifica el final del intervalo en milisegundos.
- interval: El intervalo de agregación, en milisegundos.
- agg: La función de agregación. Uno de MIN, MAX, AVG, SUM, COUNT, NONE.
- límit: La cantidad máxima de puntos de datos para devolver o intervalos para procesar.

ThingsBoard usará startTs, endTs e interval para identificar particiones de agregación o subconsultas y ejecutar consultas asincrónicas a la base de datos que aprovechan las funciones de agregación integradas.

3.2.6.3 Motor de reglas

ThingsBoard proporciona API para enviar llamadas API REST personalizadas al motor de reglas, procesar la carga útil de la solicitud y devolver el resultado del procesamiento en el cuerpo de la respuesta. Esto es útil para varios casos de uso. Por ejemplo:

- extender la API REST existente de la plataforma con llamadas API personalizadas;
- enriquecer la llamada API REST con los atributos de dispositivo / activo / cliente y reenviar al sistema externo para un procesamiento complejo;
- proporcione una API personalizada para sus widgets personalizados.

Para ejecutar la llamada a la API REST, puede utilizar la regla-motor-controlador API REST:

Figura 3-14. API para motor de reglas

3.2.7 ThingsBoard PE vs ThingsBoard CE

Dentro de ThingsBoard existen principalmente dos tipos de productos principales: ThingsBoard CE (Community Edition) y ThingsBoard PE (Professional Edition). En este apartado vamos a ofrecer una pequeña comparativa para poder discernir las diferencias principales.

ThingsBoard CE permite realizar las funcionalidades básicas de ThingsBoard. Que permite manejas los activos y los datos recolectados, junto con las cadenas de reglas y los paneles. Permite también el transporte en los protocolos establecidos y los componentes del motor de reglas son básicas. Además, no permite que tenga múltiples inquilinos dentro de inquilinos, a diferencia de la PE.

ThingsBoard PE tiene, además de las funcionalidades básicas mencionadas anteriormente, permite planificar las acciones, configurar el menú completamente y crear grupos de entidades. Además, podemos integrar acciones de otras plataformas.

4 APLICACIÓN

Actualmente la forma de contabilizar el consumo de agua se encuentra realizado de forma clásica, por lo que puede ser tedioso y llevar mucho tiempo. Para poder controlar el consumo tenemos que ir a los cuartos de mantenimiento o en los armarios destinados a ello, por lo que tenemos que acceder físicamente a cada aparato para comprobar cuanto llevamos consumido. Además, para poder cortar el acceso al agua, tenemos que girar las llaves de acceso, que suelen ser de difícil acceso.

No existen muchas aplicaciones que puedan contabilizar el consumo de agua que se ha llevado y menos aplicaciones que accedan al contador para poder cortar el acceso al mismo. Por ello, esta aplicación quiere solucionar estos dos puntos de forma sencilla y eficaz.

Para conseguirlo incorpora las siguientes características:

- Un sistema de registro de usuario para que la aplicación sea personal e intransferible.
- Un sistema de acceso a la aplicación que permite acceder a tus datos de manera segura.
- Una pantalla de registro del contador para que puedas agregar tu dispositivo.
- Un apartado de perfil personal en el que se muestran los datos del usuario y del contador.
- Un apartado de configuración en el que se puede modificar los datos del usuario o del contador y que se pueden realizar acciones como cambiar de contador o de usuario.
- Un apartado de configuración en el que se puede modificar el nivel de agua consumida en el que se genere una alerta.
- Una pantalla para visualizar el agua consumida durante el día, la semana y el mes. Además, podemos cortar o restaurar el acceso agua.
- Una pantalla para poder visualizar el histórico de agua consumida, aplicando una comparativa con periodos anteriores.
- Un apartado donde podemos visualizar todas las alarmas generadas por el consumo excesivo.

4.1 Gráfico global del Proyecto

Antes de desarrollar la aplicación, primero hemos de considerar todas las partes del proyecto y como estas interactúan unas con otras. En este apartado vamos a explicar, junto con una gráfica, como interactúan cada una de ellas y en qué aspectos del proyecto se van a desarrollar.

La gráfica global del proyecto es el siguiente:

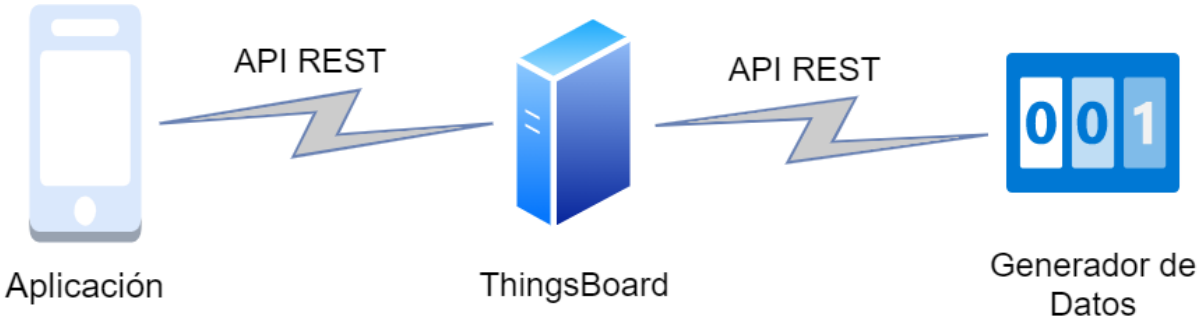

Figura 4-1. Gráfico Global del Proyecto

Donde vamos a dividir el proyecto en tres partes :

- **La aplicación**: Aspecto del proyecto que se comunica con el cliente y en el que se generan las gráficas y las operaciones con los datos que se recogen con ThingsBoard. Para comunicarnos con

ThingsBoard utilizaremos peticiones HTTP a las diferentes API REST de ThingsBoard para que este guarde o intercambie información con la aplicación.

- **ThingsBoard**: Servidor del proyecto donde se almacenan los distintos datos que provienen tanto del cliente como del generador de datos, y generan las distintas alarmas con los datos recogidos. Además, si alguno de los elementos requiere de algún dato, ThingsBoard se los proporciona.
- **El generador de datos**: Produce aleatoriamente datos simulando el funcionamiento de un contador, que entregará posteriormente a ThingsBoard mediante una petición HTTP utilizando las distintas API REST del servidor.

A continuación, veamos con más detalle los distintos elementos del proyecto.

4.1.1 La aplicación

La aplicación sirve fundamentalmente como un elemento de contacto entre el cliente y el servidor ThingsBoard, para ello solicita al cliente los distintos datos fundamentales para mostrarle los diferentes aspectos del contador. Es decir, le solicita al cliente los datos del contador y del usuario suficientes y para crearlos y guardarlos en el servidor. También le reclama al servidor los datos del contador para mostrarle al cliente el agua consumida, las comparativas de los días y las alarmas producidas por el consumo excesivo.

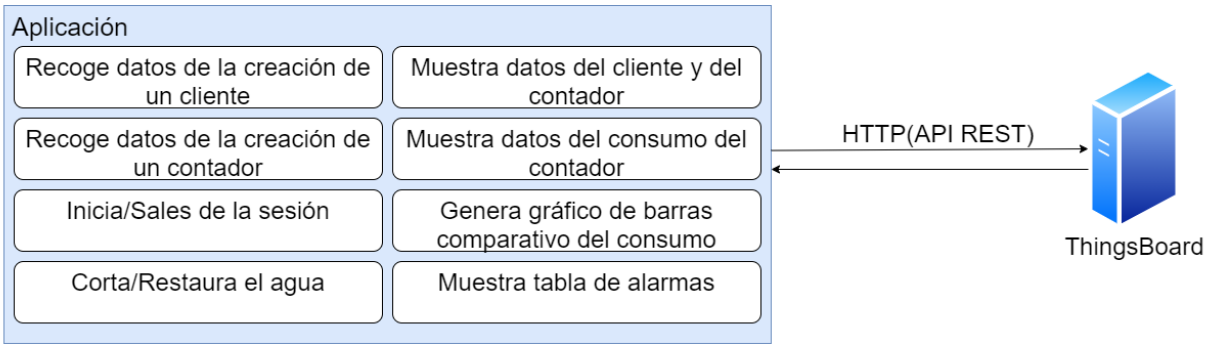

Figura 4-2. Gráfico de la Aplicación

Una vez realizado todas las acciones le muestra al cliente los resultados.

4.1.2 ThingsBoard

ThingsBoard es el servidor que fundamentalmente almacena los datos que recibe de la aplicación y del generador de datos. Una vez recibe los datos, los procesa y los almacena de manera representativa en su base de datos interna, de modo que podamos trabajar con ellos a posteriori.

ThingsBoard recibe por parte de la aplicación los datos que sirven para crear los clientes y el contador. Una vez creado en la base de datos guarda los atributos asociados al contador, así como las cadenas de reglas que se utilizan para generar las alarmas que se producen por el consumo excesivo por parte del cliente.

Por parte del generador recibe las distintas telemetrías que se asocian a un contador en específico y lo guarda en la base de datos, también se asegura que la telemetría pase por la cadena de reglas para ver si se ha producido una alarma.

Tanto la aplicación como el generador necesitan datos de los clientes y de los dispositivos para su funcionamiento. Por lo tanto, ThingsBoard les proporciona los datos necesarios para ello.

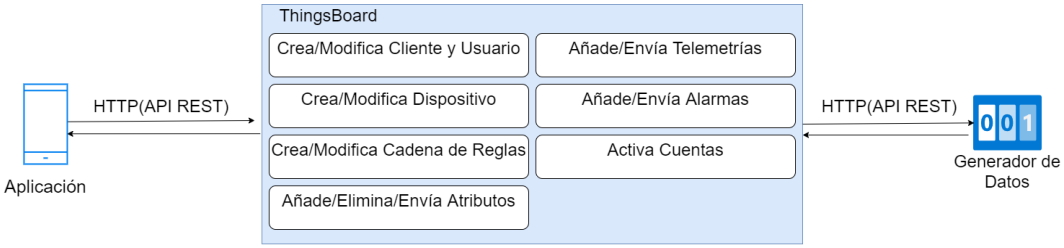

Figura 4-3. Gráfico del Servidor

4.1.3 El generador de datos

Por último, el generador de datos simula el comportamiento de un contador de agua generando datos aleatorios que se envían al servidor ThingsBoard para su almacenamiento. Para ello envía mediante una petición HTTP a la API REST del servidor los datos generados.

Para que el generador funcione, tenemos que pedirle al servidor las credenciales de nuestro usuario para poder enviarle los datos al contador asociado. Luego, comprobamos si está activo mirando el atributo correspondiente. Si todo es correcto, le envío los datos.

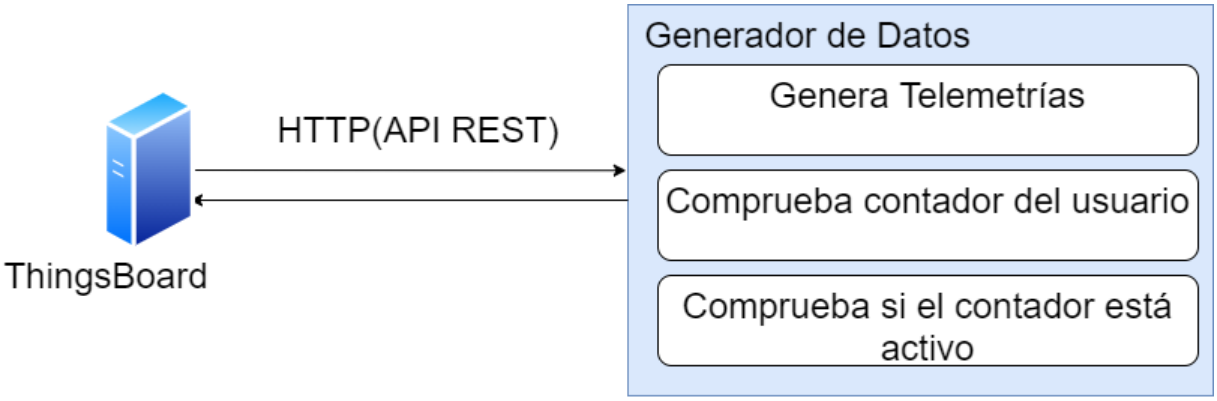

Figura 4-4. Gráfico del Generador de datos

4.2 Desarrollo de React Native en la aplicación

La aplicación se encuentra desarrollada en React Native y para que se lleve a cabo se ha desarrollado en una serie de archivos, diferenciados en la utilidad que le aporta a la aplicación. Por lo tanto, hemos estructurado la aplicación en la siguiente manera:

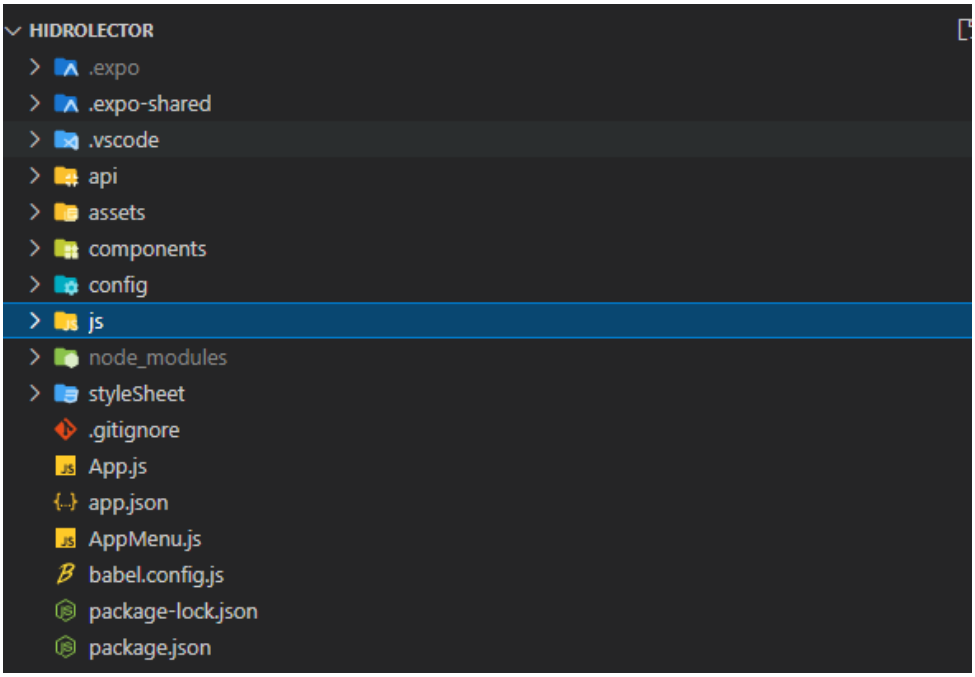

Figura 4-5. Estructura de la aplicación en React Native

Dentro de esta estructura dividimos la aplicación en los siguientes módulos:

- Ficheros generales que aportan la navegación en la aplicación.
- El directorio “js”: que contiene todos los ficheros que componen las diferentes pantallas dentro de la

aplicación.

- El directorio “styleSheet”: que contiene los ficheros de estilo que manejan las pantallas y que aportan el diseño de la aplicación.
- El directorio “config” : que contiene los ficheros que aportan los elementos constantes que se usan en todos los demás ficheros.
- El directorio “components”: que contiene los ficheros que manejan diferentes componentes que se usan dentro la aplicación.
- El directorio “api”: que contiene los ficheros que manejan la conexión entre el dispositivo móvil y el servidor ThingsBoard

A continuación, se explicará en mayor detalle los elementos de React Native que se utilizan en los distintos directorios y ficheros y que utilidad tienen cada uno de ellos.

4.2.1 Ficheros de navegación

Dentro de la aplicación, tenemos dos ficheros (App.js y AppMenu.js), que aportan la navegación entre las diferentes pantallas de la aplicación. Para ello utilizaremos el módulo de React Native que se denomina React Navigation. [6]

Para ello instalamos en la aplicación el paquete mediante el siguiente paquete:

Paquete de React Navigation:

```
expo install @react-navigation/native
expo install @react-navigation/stack
```

Este módulo permite crear una estructura de navegación dentro de la aplicación, antes de instalar el paquete hay que instalar las dependencias que son fundamentales a la hora de que el paquete funcione. Estas dependencias son “react-native-screens” y “react-native-safe-area-context.”

Instalación de dependencias:

```
expo install react-native-screens react-native-safe-area-context
```

Una vez instalado el paquete React Navigation, ya se puede importar en el proyecto. La importación se realiza importando los elementos necesarios en nuestro proyecto. En nuestro caso se realiza incorporando a los ficheros la siguiente línea:

```
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
```

Figura 4-6. Líneas de import del proyecto

Una vez que ya tenemos importado los elementos necesarios, el React Navigation proporciona una forma para que la aplicación haga la transición entre pantallas y administre el historial de navegación. Para ello la aplicación empuja y saca elementos de la pila de navegación a medida que los usuarios interactúan con ella, y esto hace que el usuario vea diferentes pantallas.

Para que la aplicación se ejecute de la manera que se desea, creamos en el archivo principal App.js (que es el primer fichero que accede la aplicación), la pantalla principal que es la pantalla de Login y después creamos diferentes pantallas secundarias que vayan a las demás pantallas.

Para ello lo creamos mediante createNativeStackNavigator, que es una función que devuelve un objeto que contiene 2 propiedades: Screen y Navigator. Ambos son componentes de React que se utilizan para configurar el navegador. El Navigator debe contener elementos Screen como hijos para definir la configuración de rutas.

```
const Stack = createStackNavigator();
```

Figura 4-7. Creación de la pila

Para que todo funcione, NavigationContainer es un componente que administra nuestro árbol de navegación y contiene el estado de navegación. Este componente debe envolver la estructura de todos los navegadores.

```
function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName='Login'>
        <Stack.Screen
          name="Login"
          component={Login}
          options={{
            headerShown: false
          }} />
        <Stack.Screen
          name="Register"
          component={Register}      You, August 1
          options={{
            headerStyle: {
              backgroundColor: color.TOMATO_RED,
            },
            headerBackTitleStyle: {
              fontWeight: 'bold'
            },
            headerTintColor: color.WHITE,
            headerTitleAlign: 'center',
            title: 'Registro'
          }} />
          <Stack.Screen
            name='AppMenu'
            component={AppMenu}
            options={{
              headerShown: false
            }} />
      </Stack.Navigator>
    </NavigationContainer>
```

Figura 4-8. Estructura de la navegación de App.js

Una vez que iniciamos sesión, vamos al menú de la aplicación. Como este elemento es otro navegador diferente, para poder llegar desde App.js a AppMenu.js anidamos el navegador del menú dentro de la pila de App.js. Para ello simplemente añadimos en el atributo “component” del navegador principal al navegador secundario.

Al final, la estructura de navegadores que está siguiendo la aplicación es la siguiente:

- Stack.Navigator
    - Login (Screen Principal)
    - Register (Screen)
    - Menu (Screen)
        - FirstUser
        - CounterRegister
        - Profile
        - Configuration
        - ConfigurationInform
        - ConfigurationUser
        - Inform
        - Counter

Esta navegación permite que se pueda seguir la aplicación de forma fluida y ordenada, de modo que la pila pueda rellenarse por orden y permitir la navegación entre pantallas.

4.2.2 Componentes de React Native

Dentro de los ficheros de la aplicación utilizamos distintos componentes que aportan la estructura de la aplicación, es decir, los elementos que nos ayudan a que la aplicación se utilice correctamente. Cada componente realiza una acción característica sobre la aplicación general. [7]

Los distintos componentes que se añaden en esta aplicación son las siguientes:

4.2.2.1 BarChart

Este componente se utiliza para generar un gráfico de barras. Para ello tenemos que añadir a nuestro proyecto el módulo correspondiente a este componente, que sería el componente "react-native-svg-charts". [8] Para instalarlo ponemos dentro del cuadro de mando del proyecto:

Paquete de react-native-svg-charts:

```
expo install react-native-svg-charts
```

Una vez instalado el paquete tenemos que importar los elementos correspondientes al gráfico de barras que es el que vamos a utilizar en esta aplicación. Para ello utilizamos la siguiente línea:

```
import { BarChart, YAxis, Grid } from 'react-native-svg-charts';
```

Figura 4-9. Importar el componente BarChart

Una vez importado, podemos utilizarlo. Para ello incorporamos el elemento BarChart a nuestra aplicación.

```
<BarChart
    style={styles.barChart}
    animate={true}
    animationDuration={500}
    data= {barData}
    yMax={max}
    yAccessor={({ item }) => item.value}
    spacingInner={0.5}
    contentInset={contentInset}
>
</BarChart>
```

Figura 4-10. Componente BarChart

Dentro del componente hay un par de atributos que hay que aclarar. El atributo spacingInner nos permite manejar el espacio entre las barras mientras que el atributo contentInset nos permite manejar el posicionamiento del gráfico dentro del componente. El atributo contentInset indica la cantidad de inserción del contenido desde los bordes.

Además, podemos añadir al gráfico el número de datos distintos que queramos, que en nuestro caso nos permite aplicar una comparativa de barras del mismo contador en diferentes momentos.

Para poder decorar el gráfico hemos añadido a la izquierda de este el elemento YAxis, que es un componente auxiliar para diseñar las etiquetas del eje Y en las mismas coordenadas que su gráfico. Es muy importante que el componente tenga exactamente los mismos límites del gráfico si no el eje y estará descuadrado con los elementos del gráfico de barras.

```
<YAxis
    data={ydata}
    svg={{
        fill: 'black',
        fontSize: 10,
    }}
    numberOfTicks={8}
    formatLabel={(value) => `${value} L`}
    contentInset={yAxisInset}
/>
```

Figura 4-11. Componente YAxis

4.2.2.2 Image

Componente de React para mostrar diferentes tipos de imágenes, incluidas imágenes de red, recursos estáticos, imágenes locales temporales e imágenes del disco local, como el carrete de la cámara.

En nuestro caso, utilizamos este componente para manejar las diferentes imágenes que tendrá la aplicación, todas las imágenes se encuentran guardadas en la carpeta Images dentro del directorio assests, donde se guardan los diferentes activos dentro de la aplicación.

```
<Image
    style={styles.logo}
    source={require('../assets/Images/Logo.png')} />
```

Figura 4-12. Componente Image

Esto se logra mediante la propiedad del componente source, que puede buscar elementos tanto en local como en remoto, es decir, aplicando URL.

4.2.2.3 KeyboardAwareScrollView

Este componente se utiliza para controlar la apariencia del teclado y se desplaza automáticamente hasta enfocar la pantalla en el TextInput que estamos centrados. Para ello tenemos que añadir a nuestro proyecto el módulo correspondiente a este componente, que sería el componente "react-native-keyboard-aware-scroll-view". [9] Para instalarlo ponemos dentro del cuadro de mando del proyecto:

Paquete de react-native-keyboard-aware-scroll-view:

```
expo install react-native-keyboard-aware-scroll-view
```

Una vez instalado el paquete tenemos que importar el componente que vamos a utilizar en esta aplicación. Para ello utilizamos la siguiente línea:

```
import { KeyboardAwareScrollView } from 'react-native-keyboard-aware-scroll-view';
```

Figura 4-13. Importar el componente KeyboardAwareScrollView

Una vez importado, añadimos el elemento a la pantalla correspondiente. Nos aseguramos de que todos los elementos de la pantalla se encuentran recogidos dentro del componente para que funcione. Además, este componente funciona como un View y, por lo tanto, no puede tener un View inmediatamente antes del componente.

```
<KeyboardAwareScrollView
    style={styles.container} >

    <StatusBar barStyle='dark-content' backgroundColor='transparent' />

    <TouchableWithoutFeedback
        onPress={() => Keyboard.dismiss()}>
        <View>
            <Logo />

            <View style={styles.inputContainer}>···
            <View style={styles.buttonContainer}>
                <TouchableOpacity ···
                <Text
                    style={styles.registerText}
                    onPress={onRegisterPress}>
                    {strings.REGISTRAR}
                </Text>
            </View>
        </View>
    </TouchableWithoutFeedback>
</KeyboardAwareScrollView>
```

Figura 4-14. Componente KeyboardAwareScrollView

Normalmente, este componente se ocupará solo de manejar el contenido insertado. Si desea que KeyboardAwareScrollView se desplace automáticamente a un TextInput que obtiene el foco (por lo que se asegura que sea visible), puede agregar un atributo llamado getTextInputRefs: una devolución de llamada en la que puede devolver una matriz de referencias a los componentes TextInput que se desplazan automáticamente para ser manejado.

KeyboardAwareScrollView buscará el TextInput enfocado en la matriz y se asegurará de que sea visible desplazándose hasta su ubicación.

4.2.2.4 MaterialCommunityIcons

Este componente se utiliza para añadir iconos que se utilizan para mostrar elementos y es perfecto para botones, logotipos y barras de navegación/pestañas. Para ello tenemos que añadir a nuestro proyecto el módulo correspondiente a este componente, que sería el componente "@expo/vectors-icons" [10]. Para instalarlo ponemos dentro del cuadro de mando del proyecto:

Paquete de @expo/vectors-icons:

```
expo install @expo/vectors-icons
```

Una vez instalado el paquete tenemos que importar el componente que vamos a utilizar en esta aplicación. Para ello utilizamos la siguiente línea:

```
import {MaterialCommunityIcons as Icon} from '@expo/vector-icons';
```

Figura 4-15. Importar MaterialComunityIcons

Una vez importado, añadimos el elemento a la pantalla correspondiente. En nuestro caso, el elemento que vamos a añadir es el ojo que se añaden en las contraseñas. De modo que cuando pulsen, se cambie al icono de un ojo tachado permitiendo que la contraseña se visualice.

```
<Icon
    name={eye}
    size={30}
    style={styles.eye} />
```

Figura 4-16. Componente Icon

Para manejar el estado del ojo en cada momento, utilizamos la función useSate() de React Native, permitiendo de manera sencilla implementar los iconos a instalar.

```
const [eye, setEye] = useState('eye');
```

Figura 4-17. Variable del icono

Una vez realizado la aplicación, cuando pulsamos el icono, tenemos que cambiar la variable "eye" al valor de la variable con el ojo tachado, de modo que se cambia el icono.

```
onIconPress = () => {
    setEye(secureTextEntry ? 'eye-off' : 'eye' );
    setSecureTextEntry(!secureTextEntry);
}
```

Figura 4-18. Cambio de icono

4.2.2.5 RadioButtonRN

Este componente se utiliza para añadir un botón de radio animado. Para ello tenemos que añadir a nuestro proyecto el módulo correspondiente a este componente, que sería el componente "radio-buttons-react-native" [11]. Para instalarlo ponemos dentro del cuadro de mando del proyecto:

Paquete de radio-buttons-react-native:

```
expo install radio-buttons-react-native
```

Una vez instalado el paquete tenemos que importar el componente que vamos a utilizar en esta aplicación. Para ello utilizamos la siguiente línea:

```
import RadioButtonRN from 'radio-buttons-react-native';
```

Figura 4-19. Importar RadioButtonRN

Una vez importado, añadimos el elemento a la pantalla correspondiente.

```
<RadioButtonRN
    data={data}
    initial={nivel == 130 ? 1 : 2}      You, Augu
    box={false}
    style={styles.radioButton}
    boxStyle={styles.radioBox}
    textStyle={styles.radioText}
    selectedBtn={(e) => radioButtonSelected(e)}
/>
```

Figura 4-20. Componente RadioButtonRN

Este componente nos permitirá seleccionar el nivel de agua máximo que se consumirá sin que salte una alerta en la aplicación.

4.2.2.6 ScrollView

Componente que envuelve la plataforma al tiempo que proporciona integración con el sistema de "respuesta" de bloqueo táctil. Hay que tener en cuenta que ScrollView debe tener una altura limitada para funcionar.

El ScrollView es un contenedor de desplazamiento genérico que puede contener varios componentes y puntos de vista. Los elementos desplazables pueden ser heterogéneos y puede desplazarse tanto vertical como horizontalmente.

```
<ScrollView style={styles.dataWrapper}>
    <Table borderStyle={styles.borderStyle}>
        {
            tableData.map((rowData, index) => (
                <Row
                    key={index}
                    data={rowData}
                    widthArr={widthArr}
                    style={[styles.row, index%2 && {backgroundColor: color.MEDIU
                    textStyle={styles.tableText}
                />
            ))
        }

    </Table>
</ScrollView>
```

Figura 4-21. Componente ScrollView

En la aplicación nos sirve para mostrar la tabla de alertas, ya que la tabla es muy grande para la pantalla, este componente nos permite mostrar los datos sin que se tenga que empequeñecer la tabla para que quepa en la pantalla.

4.2.2.7 StatusBar

Componente para controlar la barra de estado de la aplicación. Nos permite guardar el espacio de la barra de notificaciones y que la aplicación no se solape con la misma.

```
<StatusBar barStyle='dark-content' backgroundColor='transparent' />
```

Figura 4-22. Componente StatusBar

En la aplicación usamos el atributo backgroundColor para que la barra de estado sea del dispositivo móvil poniéndolo transparente.

4.2.2.8 Svg

Este componente se utiliza para generar un gráfico de donut. Para ello tenemos que añadir a nuestro proyecto el módulo correspondiente a este componente, que sería el componente “react-native-svg” [12]. Para instalarlo ponemos dentro del cuadro de mando del proyecto:

Paquete de react-native-svg:

```
expo install react-native-svg
```

Este componente genera elementos gráficos básicos como círculos y elementos primitivos que en conjunto podemos crear elementos gráficos complejos (en nuestro caso la gráfica donut.)

Una vez instalado el paquete tenemos que importar los elementos correspondientes al gráfico que es el que

vamos a utilizar en esta aplicación. Para ello utilizamos la siguiente línea:

```
import Svg, {G, Circle} from 'react-native-svg';
```

Figura 4-23. Importar el componente Svg

Una vez importado, podemos utilizarlo. Para ello incorporamos el elemento BarChart a nuestra aplicación.

```
<Svg
    height={radius * 2}
    width={radius * 2}
    viewBox={`0 0 ${halfCircle * 2} ${halfCircle * 2}`}>
    <G
        rotation="-90"
        origin={`${halfCircle}, ${halfCircle}`}>
        <Circle
            ref={circleRef} ···
            strokeDasharray={circumference}
        />
        <Circle
            cx="50%" ···
        />
    </G>
</Svg>
```

Figura 4-24. Componente Svg

Para crear el donut, dentro de la gráfica tenemos que añadir los componentes Circle para que tenga la forma de círculos. Una vez creado el donut, lo animamos con el componente Animated para que le dé la forma de gráfica.

Además, podemos añadir al gráfico el número de datos distintos que queramos, que en nuestro caso nos permite darle la suma del contador para los diferentes días.

4.2.2.9 Switch

Muestra una entrada booleana.

Este es un componente controlado que requiere una devolución de llamada que actualice el accesorio value para que el componente refleje las acciones del usuario. Si el accesorio value no se actualiza, el componente seguirá representando el accesorio value proporcionado en lugar del resultado esperado de las acciones del usuario.

En la aplicación utilizamos este componente para que muestre los distintos botones de comparativa, si es que queremos comparar los datos. Si el Switch se encuentra apagado, la gráfica mostrada no tendrá la comparativa de los datos del contador. Si se encuentra encendido podemos ver como las gráficas realizan una comparativa.

```
<Switch
    style={styles.toogle}
    trackColor={{ false: color.GRAY, true: color.TOMATO_RED}}
    thumbColor={isEnabled ? color.GRAY : color.TOMATO_RED}
    onValueChange={toogleSwitch}
    value={isEnabled}
/>
```

Figura 4-25. Componente Switch

Para que se muestren los botones, hemos utilizados dos variables de estado que indican si el switch se encuentra activo y si los botones están escondidos. De este modo cuando activamos el componente, los

botones se muestran y si se desactiva, se vuelven a esconder.

```
const [isEnabled, setIsEnabled] = useState(false);
const [isHidden, setIsHidden] = useState(true);
const [button, setButton] = useState('Semana');
const [comparar, setComparar] = useState('Periodo');

const toogleSwitch = () => {
    if(button == 'Alarmas'){
        setIsEnabled(false);
        setIsHidden(true);
    }
    else{
        setIsEnabled(!isEnabled);
        setIsHidden(!isHidden);
    }
}
```

Figura 4-26. Activación del Switch

4.2.2.10 Table

Este componente se utiliza para generar una tabla. Para ello tenemos que añadir a nuestro proyecto el módulo correspondiente a este componente, que sería el componente "react-native-table-component". [13] Para instalarlo ponemos dentro del cuadro de mando del proyecto:

Paquete de react-native-table-component:

```
expo install react-native-table-component
```

Una vez instalado el paquete tenemos que importar los elementos correspondientes a la tabla que vamos a utilizar en esta aplicación. Para ello utilizamos la siguiente línea:

```
import {Table, Row} from 'react-native-table-component';
```

Figura 4-27. Importar el componente Table

Una vez importado, podemos utilizarlo. Para ello incorporamos el elemento Table a nuestra aplicación para generar la estructura de la tabla, para añadir elementos a la tabla, utilizamos el componente Row que nos permite crear filas de elementos.

```
<Table borderStyle={styles.borderStyle}>
    <Row
        data={tableHead}
        widthArr={widthArr}
        style={styles.header}
        textStyle={styles.tableHeaderText}
    />
</Table>
<ScrollView style={styles.dataWrapper}>
    <Table borderStyle={styles.borderStyle}>
        {
            tableData.map((rowData, index) => (
                <Row
                    key={index}
                    data={rowData}
                    widthArr={widthArr}
                    style={[styles.row, index%2 && {backgroundColor: color.MEDIUM_PURP}]}
                    textStyle={styles.tableText}
                />
            ))
        }

    </Table>
```

Figura 4-28. Componente Table

En nuestra aplicación, tenemos una primera tabla que contiene la cabecera de esta. A continuación, generamos otra tabla que contiene las alarmas que va generando la aplicación. Los elementos de esta segunda tabla son dinámicos y se añaden a medida que se van generando alarmas.

4.2.2.11 Text

Un componente de React para mostrar texto. En la aplicación lo usamos para manejar los diferentes strings que se desarrollan en el fichero "strings.js" del directorio "config".

```
<Text style={styles.text}>{strings.CONTRASEÑA}</Text>
```

Figura 4-29. Componente Text

4.2.2.12 TextInput

Un componente fundamental para ingresar texto en la aplicación a través de un teclado. Los accesorios brindan capacidad de configuración para varias funciones, como autocorrección, uso automático de mayúsculas, texto de marcador de posición y diferentes tipos de teclado, como un teclado numérico.

El caso de uso más básico es dejar caer un TextInput y suscribirse a los eventos onChangeText para leer la entrada del usuario. También hay otros eventos, como onSubmitEditing y a los onFocus que se puede suscribir.

```
<TextInput
    ref={ (input) => { emailInput = input; }}
    style={styles.textInput}
    placeholder={strings.EMAIL}
    placeholderTextColor={color.WHITE}
    clearTextOnFocus
    blurOnSubmit={false}
    returnKeyType='next'
    onSubmitEditing={ () => { passwordInput.focus(); }}
    onChangeText={(text) => changeHandler('Email', text)}
/>
```

Figura 4-30. Component TextInput

En la aplicación usamos este componente para realizar los formularios tanto de registro del usuario y del contador como para iniciar sesión de la aplicación. Uno de los atributos que se utilizan es el de secureTextEntry, que se utiliza para ocultar el texto en la entrada de las contraseñas.

4.2.2.13 TouchableOpacity

Un envoltorio para hacer que las vistas respondan correctamente a los toques. Al presionar hacia abajo, la opacidad de la vista envuelta se reduce, atenuándola. La opacidad se controla envolviendo a los hijos en un Animated.View, que se agrega a la jerarquía de vistas.

En la aplicación sirve para saber si hemos pulsado un botón. Una vez que se utiliza, todos los elementos que contienen se verán afectados al pulsarlo, por lo que sirve para crear botones con varios elementos (imágenes, texto…).

```
<TouchableOpacity
    style={styles.button}
    onPress={onConfigurationPress}
>
    <Text
        style={styles.text} >
        {strings.GUARDAR}
    </Text>
</TouchableOpacity>
```

Figura 4-31. Componente TouchableOpacity

4.2.2.14 View

El componente más fundamental para crear una interfaz de usuario. View está diseñado para anidarse dentro de otras vistas y puede tener otros componentes en su interior. En la aplicación, hemos usado este elemento para poder separar la aplicación en contenedores y que a cada uno le apliquemos un estilo diferente

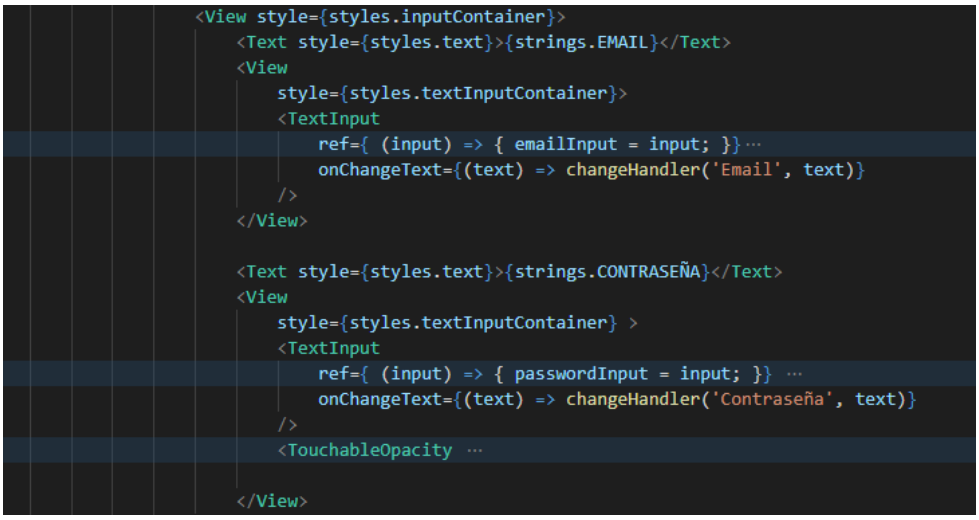

Figura 4-32. Componente View

4.2.3 APIs utilizadas en React Native

Una API es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. Las API son un medio simplificado de conectar los elementos de la aplicación, esto nos permite conectarnos también con elementos en la nube.

En la aplicación desarrollamos fundamentalmente dos APIs diferentes, la primera son aquellos elementos de React Native que nos sirven para que la aplicación reaccione de manera dinámica a los eventos que generamos en la aplicación y nos proporciona actividad y dinamismo. [7]

La segunda son aquellas APIs que nos aporta la funcionalidad con elementos externos de la aplicación y que nos sirven como un canal de comunicación para recoger y enviar información a estos servidores externos. En la aplicación utilizamos estas APIs para comunicarnos con ThingsBoard y nos proporcionan toda la información que requerimos para que la aplicación se ejecute sin problemas. [14]

4.2.3.1 APIs de React Native

Para poder ejecutar las API de React Native tenemos que importar a los ficheros correspondientes los elementos que se van a ejecutar para que se generen los efectos que se desean en la aplicación. Cada elemento tiene una serie de funciones y de propiedades que lo define de manera más específica.

A continuación, mostraremos con más detalle las APIs utilizadas en la aplicación, así como la utilidad que estos aportan a la ejecución de esta. De igual forma, describiremos las funciones y propiedades importantes que utilizaremos.

4.2.3.1.1 Alert

Inicia un cuadro de diálogo de alerta con el título y el mensaje especificados.

Opcionalmente, proporciona una lista de botones. Al tocar cualquier botón, se activará la devolución de llamada onPress respectiva y se descartará la alerta. De forma predeterminada, el único botón será un botón "Aceptar".

En nuestro caso utilizamos esta API para poder generar alerta en caso de que se genere algún error en la aplicación.

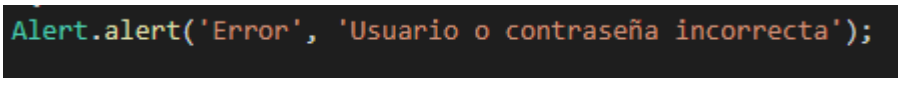

Figura 4-33. API Alert

La función alert nos permite que se genere en la aplicación la alerta y que este se muestre por pantalla. Una vez mostrada, la aplicación non podrá continuar hasta que el botón sea pulsado, de este modo nos aseguramos de

que el mensaje sea entregado.

4.2.3.1.2 Animated

La biblioteca Animated está diseñada para hacer que las animaciones sean fluidas, poderosas y fáciles de construir y mantener. Se centra en relaciones declarativas entre entradas y salidas, transformaciones configurables intermedias y métodos start/stop para controlar la ejecución de la animación basada en el tiempo.

El flujo de trabajo principal para crear una animación es crear una Animated.Value, conectarla a uno o más atributos de estilo de un componente animado y luego impulsar actualizaciones a través de animaciones usando Animated.timing().

En nuestro caso utilizamos la API Animated para generar animaciones de relleno tanto de la gráfica que genera las barras de contadores de agua como de las gráficas de dónuts que muestran las diferentes cantidades de agua acumulada.

```
const animation = (toValue) => {
    return Animated.timing(animated, {
      delay: 1000,
      toValue,
      duration,
      useNativeDriver: true,
      easing: Easing.out(Easing.ease),
    }).start();
};
```

Figura 4-34. API Animated

En este caso usamos el parámetro “delay” para determinar el retraso que tendrá la aplicación con respecto a la aparición de la pantalla en milisegundo (en nuestro caso 1 segundo) y que llegue hasta el valor marcado por la propiedad “toValue”.

Además, añadimos la API Easing. La API Easing implementa funciones de aceleración comunes. Este módulo es utilizado por Animated.timing () para transmitir movimiento físicamente creíble en animaciones.

4.2.3.1.3 Keyboard

Usamos el módulo Keyboard para controlar los eventos del teclado. Esto nos permite escuchar eventos nativos y reaccionar ante ellos, así como realizar cambios en el teclado, como descartarlo.

Esto nos permite descartar el teclado cuando salimos de escribir en un campo de texto, nos da la facilidad de ver la pantalla completa y así evitar que tengamos que pulsar la tecla de retroceso para ocultar el teclado, pudiendo provocar que retrocedamos en la pantalla.

```
<TouchableWithoutFeedback
    onPress={() => Keyboard.dismiss()}>
```

Figura 4-35. API KeyBoard

Utilizamos la función dismiss, esta función descarta el teclado activo y quita el foco del elemento donde se encontraba, en general un TextInput.

4.2.3.1.4 Platform

Usamos el módulo Platform para controlar el sistema donde corre la aplicación. Esto nos permite manejar diferentes estilos dependiendo de si el sistema donde se ejecuta la aplicación es Android o IoS.

Esta API se utiliza para que se generen unas alertas si el elemento es IoS y utilizar el componente ToastAndroid si el sistema operativo es Android.

```
if(Platform.OS === 'android'){
    ToastAndroid.show('Usuario Creado', ToastAndroid.SHORT);
}
else {
    Alert.alert('Info', 'Usuario Creado');
}       You, August 20th, 2021 5:57pm • Se puede logear y regis
```

Figura 4-36. API Platform

Dentro de este elemento utilizamos la función OS, esta función devuelve un valor de cadena que representa el sistema operativo actual. Puede ser una de las dos opciones siguientes: Android o IoS. Suele utilizarse para diferenciar acciones o elementos dependiendo del sistema operativo que se esté utilizando.

4.2.3.1.5 StyleSheet

API que realiza una abstracción similar a CSS del HTML. En este documento, utilizamos este componente para crear los estilos en los que se basa la pantalla correspondiente al fichero del estilo. Todos los ficheros de estilo se guardan en la carpeta de "styleSheet".

```
export default StyleSheet.create({
    barContainer: {
        flex: 1,
        flexDirection: 'row',
        backgroundColor: color.WHITE
    },
    barChartContainer: {
        flex: 1,
        marginLeft: 10
    },
    barChart: {
        flex: 1
    },
```

Figura 4-37. API StyleSheet

Para llamar al estilo correspondiente, solo hay que importarlo de la carpeta y ya se aplica el estilo a los elementos.

```
import styles from '../styleSheet/ConfigurationInformStyle.js';
```

Figura 4-38. Importar estilos

Utilizamos la función create, que crea una referencia de estilo StyleSheet a partir del objeto dado. Este objeto contiene todos los estilos que se aplicarán en la pantalla de la aplicación y que en nuestro caso llamaremos en el elemento al que le queramos aplicar el estilo.

4.2.3.1.6 ToastAndroid

Expone el módulo ToastAndroid de la plataforma Android como un módulo JS. Proporciona el método show(message, duration)que toma los siguientes parámetros:

- mensaje Una cadena con el texto para brindar
- duración La duración del brindis, ya sea ToastAndroid.SHORT o ToastAndroid.LONG.

En la aplicación se utiliza para generar notificaciones cortas sobre cambios específicos de la aplicación, como guardar una configuración o cortar/restaurar el agua.

```
ToastAndroid.show('Configuración Cambiada', ToastAndroid.SHORT);
```

Figura 4-39. Componente ToastAndroid

Utilizamos la función show, que muestra el mensaje del primer parámetro con la duración que le apliquemos en el segundo parámetro. Esto nos permite generar notificaciones que nos informan pero que, a diferencia de la API Alert, no son tan fundamentales para leer y por lo tanto no hace falta parar la aplicación para informarlo.

4.2.3.2 APIs de ThingsBoard

Estas API nos permite conectarnos con el servidor de ThingsBoard para sacar o enviar la información relativo a las diferentes entidades que se crean o que se encuentran creadas dentro del servidor. La API de ThingsBoard consiste en dos partes principales: las API del dispositivo y las API del lado del servidor. Como en la aplicación simplemente recogemos los datos del servidor, utilizamos las que se encuentran del lado del servidor.

ThingsBoard soporta la comunicación con las API a partir de múltiples protocolos (MQTT, CoAP, HTTP y LWM2M). Como uno de los requisitos es realizar peticiones mediante la API REST, esto se realizará mediante el protocolo HTTP.

Para que React Native realice las diferentes peticiones utilizamos apisauce [15], un elemento que nos permite hacer peticiones HTTP a las diferentes URL junto con las API que se vayan a utilizar. Para instalar este elemento tenemos que realizar:

Paquete de apisauce:

```
expo install apisauce
```

Esto nos permite utilizar el elemento. Para que se realice una petición, primero tenemos que crear el api a la que vamos a conectarnos y a realizar la petición. Mediante la función create del elemento nos conectamos a la base URL que vayamos a utilizar, en nuestro caso la IP 207.180.221.115:8080 que nos ha proporcionado el profesor, las distintas cabeceras que nos indican la autorización para usar la API y los distintos formatos en los que se espera que se reciban los datos.

```
const apiPost = create ({
    baseURL: constant.URL,
    headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
    }
});
```

Figura 4-40. Creación de la API

Una vez creada la API podemos cambiar las solicitudes HTTP llamando setHeader o setHeaders en la API. Estos permanecen con la instancia de api, por lo que puede configurarlos y olvidarlos.

```
await api.setHeaders({
    'X-Authorization': token,
    'Accept': 'application/json'
});
```

Figura 4-41. Cambio de cabecera

Antes de realizar las distintas solicitudes, hay que tener en cuenta que, al realizar la petición, esta se coloca al fondo de la pila, por lo que la aplicación sigue ejecutándose hasta que se termine la función donde se realice la llamada. Para evitar esto tendremos que aplicar el método async/await dentro de la función y a la llamada de la

función para que tengamos la respuesta antes de que se ejecute el resto de la función.

```
export async function getAlarms(api, token, deviceId){      You, August 25th, 2021 8:03pm •
    await api.setHeaders({
        'X-Authorization': token,
        'Accept': 'application/json'
    });

    const alarms = await api.get('/api/alarm/DEVICE/'+deviceId+'?limit=100&ascOrder=true')
    .then(function (response){
        let respuesta = response.data;
        return respuesta;
    });
    return alarms;
}
```

Figura 4-42. Método async/await

Las distintas llamadas que se van a realizar en esta aplicación serán las llamadas get y post, que manejan diferentes parámetros. El método get utiliza estos tres parámetros:

- url: la ruta relativa a la API (obligatorio)
- params - Objeto - variables de cadena de consulta (opcional)
- axiosConfig - Objeto - configuración pasada a la solicitud axios (opcional)

Mientras, el método post utiliza estos tres parámetros:

- url: la ruta relativa a la API (obligatorio)
- datos - Objeto - el objeto que contiene los datos que se utilizan (opcional)
- axiosConfig - Objeto - configuración pasada a la solicitud axios (opcional)

Las respuestas se basan en promesas, por lo que deberá manejar las cosas en una función .then(). Lo prometido siempre se resuelve con un objeto response.

Una vez aclarado el método para realizar las peticiones. A continuación, veremos las distintas APIs que se van a utilizar dependiendo del efecto que se tiene en la aplicación la llamada a los mismos.

4.2.3.2.1 Controlador de alarmas

En estas APIs vamos a realizar todas las peticiones relativas a las distintas alarmas que se generan en la aplicación como fruto del consumo excesivo de agua por parte del contador. Dentro de este controlador utilizamos la siguiente API:

- **/api/alarm/{entityType}/{entityId}{?searchStatus,status,limit,startTime,endTime,ascOrder,offset,fetchOriginator}:**

  Método GET que nos devuelve un objeto con los datos de las alarmas creadas por parte de la aplicación, cuando se han generado y el estado de estas. Los campos entityType, entitiyId y limit son obligatorios y sirven para recoger las alarmas generadas por una entidad concreta, en nuestro caso el contador de agua. El resto de los elementos son opcionales.

4.2.3.2.2 Controlador de autorización

En estas APIs vamos a realizar todas las peticiones relativas a las distintas autorizaciones que se requieren para activar las distintas cuentas. En nuestra aplicación las utilizamos para iniciar sesión dentro de nuestro usuario y salir del usuario. Dentro de este controlador utilizamos las siguientes APIs:

- **/api/auth/login:**

  Método POST al que le tenemos que pasar como datos el email del usuario y la contraseña de este y

que si es correcto nos devuelve un objeto con el token que nos permite autorizar las diferentes acciones de nuestro usuario y autenticarnos.

- **/api/auth/logout:**

  Método POST que si es correcto nos saca de la aplicación y que provoca que el token que utilizamos al iniciar sesión no sea válido.

- **api/noauth/activate:**

  Método POST al que le tenemos que pasar como datos el token de acceso del link de activación y la contraseña de la cuenta del usuario y que si es correcto la cuenta queda activada con la contraseña establecida.

4.2.3.2.3 Controlador del cliente

En estas APIs vamos a realizar todas las peticiones relativas a las distintas acciones que se va a realizar con respecto al cliente, en nuestro caso sería la creación del cliente, obtener los clientes que se han creado en el servidor y la obtención de los datos relativos a un cliente. Dentro de este controlador utilizamos las siguientes APIs:

- **/api/customer/{customerId}/title:**

  Método GET que devuelve un string donde viene dado el nombre del cliente que buscamos mediante su id. Una vez obtenido el nombre del cliente, lo utilizamos para realizar diferentes operaciones con él, como pueden ser obtener sus usuarios.

  El parámetro customerId sirve para identificar el cliente por su identificación única.

- **/api/customers{?textSearch,idOffset,textOffset,limit}**

  Método GET que devuelve una cadena de objetos donde vienen los clientes que tiene el servidor. Lo usamos para identificar un cliente por su nombre y ver si existe. Si no tiene ninguno, el API devuelve una cadena vacía.

  Esto lo logramos mediante el parámetro opcional textSearch que busca dentro de la cadena de objetos un objeto con el mismo nombre que el parámetro. El resto de los parámetros también son opcionales

  El parámetro limit es obligatorio e indica el número máximo de clientes que queremos buscar. Como solo puede haber un cliente con el mismo nombre, el límite es de uno.

  El resto de los parámetros son opcionales y sirven para buscar con mayor precisión el dispositivo a buscar.

- **/api/customer{?entityGroupId}:**

  Método POST donde le pasamos como parámetro customer un objeto JSON y te crea el cliente. Si la petición es correcta, te devuelve un objeto con los datos generados al crear el cliente. Si al pasar el objeto customer le pasamos un objeto con la misma id que un cliente creado, el cliente se actualiza al que se le ha pasado. Esto es útil a la hora de cambiar el nombre del cliente que queremos implementar sin que se borren los datos asociados al mismo.

  Si el elemento ya se encuentra creado da un mensaje de error 401 diciendo que el título ya se encuentra en uso. El resto de los parámetros son opcionales.

4.2.3.2.4 Controlador de API del dispositivo

En estas APIs vamos a realizar todas las peticiones relativas a los atributos y la telemetría de los distintos dispositivos asociados a los usuarios. En la aplicación las utilizamos para añadir los distintos atributos y obtener los datos de las telemetrías del contador. Dentro de este controlador utilizamos las siguientes APIs:

- **/api/v1/{deviceToken}/attributes{?clientKeys,sharedKeys}:**

  Método GET que devuelve un string con el valor de los atributos que se encuentran en el lado del cliente y en el compartido. En la aplicación sirve para encontrar el valor de los distintos atributos de los contadores en los distintos lados.

El parámetro deviceToken nos permite acceder al contador mediante su token de acceso y de ahí obtener los valores mediante los parámetros opcionales clientKeys y sharedKeys, que pasando el nombre del atributo podemos buscar su valor.

- **/api/plugins/telemetry/{entityType}/{entityId}/{scope}{?keys}**

  Método POST donde le pasamos como parámetro un objeto con los distintos atributos que queremos añadir al dispositivo. Los parámetros entityType y entityId nos permite identificar el contador al que le añadiremos los atributos y el parámetro scope identifica a qué lado del dispositivo le añadiremos los atributos.

  Esta API permite tanto cambiar atributos existentes identificándolo por el parámetro keys como añadir nuevos atributos en la parte que se desea del contador.

- **/api/plugins/telemetry/{entityType}/{entityId}/values/timeseries{?agg,keys,starTs,endTs}**

  Método GET que devuelve una cadena de objetos donde vienen los datos históricos de las telemetrías del dispositivo. En la aplicación se utiliza para obtener los datos del agua del contador hasta un máximo de un mes de retroceso.

  Los parámetros admitidos son los siguientes. El parámetro keys muestra la lista separada por comas de claves de telemetría para recuperar. StartTs marca de tiempo de Unix que identifica el inicio del intervalo en milisegundos. EndTs marca de tiempo de Unix que identifica el final del intervalo en milisegundos. El intervalo indica el intervalo de agregación, en milisegundos. Agg es la función de agregación. Puede ser no de MIN, MAX, AVG, SUM, COUNT, NONE. Por último, el límite indica la cantidad máxima de puntos de datos para devolver o intervalos para procesar.

4.2.3.2.5 Controlador del dispositivo

En estas APIs vamos a realizar todas las peticiones relativas a los dispositivos creados por el usuario. En la aplicación las utilizamos para crear o cambiar los dispositivos asociados a cada usuario. También obtenemos las credenciales asociadas al dispositivo creado debido para que se puedan realizar diversas acciones sobre el mismo. Dentro de este controlador utilizamos las siguientes APIs:

- **/api/device/{deviceId}/credentials:**

  Método GET que devuelve un objeto JSON donde viene las credenciales asociadas a un dispositivo. Lo usamos para obtener el token de acceso al contador del usuario para que después podamos utilizarlo para encontrar los atributos.

  El parámetro deviceId nos permite identificar un único contador por su id y así obtener sus credenciales.

- **/api/device{?accessToken,entityGroupId}:**

  Método POST donde le pasamos como parámetro device un objeto JSON y te crea el dispositivo asociado al usuario. Si la petición es correcta, te devuelve un objeto con los datos generados al crear el dispositivo. Si al pasar el objeto device le pasamos un objeto con la misma id que un dispositivo creado, el dispositivo se actualiza al que se le ha pasado. Esto es útil a la hora de cambiar el nombre del contador que queremos implementar sin que se borren los datos asociados al mismo.

  Si el elemento ya se encuentra creado da un mensaje de error 401 diciendo que el título ya se encuentra en uso. El resto de los parámetros son opcionales.

- **/api/user/devices{?type,textSearch,idOffset,textOffset,limit}:**

  Método GET que devuelve una cadena de objetos donde vienen los dispositivos que tiene el usuario. Lo usamos para obtener los datos del contador que el usuario tiene registrado. Si no tiene ninguno, el API devuelve una cadena vacía.

  El parámetro limit es obligatorio e indica el número máximo de dispositivos que queremos buscar. Como cada usuario tiene un dispositivo. El límite es de un contador.

  El resto de los parámetros son opcionales y sirven para buscar con mayor precisión el dispositivo a

buscar.

4.2.3.2.6 Controlador de grupos de entidades

En estas APIs vamos a realizar todas las peticiones relativas a los grupos de entidades creadas en el servidor. En la aplicación las utilizamos para obtener los distintos grupos de entidades y así coger las identificaciones para que posteriormente se puedan añadir usuarios en ellas. Dentro de este controlador utilizamos las siguientes APIs:

- **/api/entityGroup/{ownerType}/{ownerId}/{groupType}/{groupName}:**

  Método GET que devuelve una cadena de objetos donde vienen los distintos grupos que se encuentran en el dueño de esos grupos. En nuestro caso, lo utilizamos para obtener el id del grupo Customer-Administrator y así poder meter en él el usuario que se asocie con el cliente.

  Todos los parámetros son obligatorios. Los parámetros ownerType y ownerId sirven para identificar al cliente dueño del grupo. Los parámetros groupType y groupName sirven para identificar el grupo del cual queremos coger las credenciales.

4.2.3.2.7 Controlador de la cadena de reglas

En estas APIs vamos a realizar todas las peticiones relativas a las cadenas de reglas que se van a crear en el servidor de ThingsBoard. En la aplicación, manejamos una cadena de reglas que permite generar alarmas si el contador se mide más agua consumida que en el nivel marcado. También modificamos la regla raíz para que cuando termine de guardar la medida, acceda a la cadena de alarmas. Dentro de este controlador utilizamos las siguientes APIs:

- **/api/ruleChain:**

  Método POST donde le pasamos al parámetro ruleChain un objeto JSON y te crea en el servidor la cadena de reglas.

  También crea una cadena nueva con los datos establecidos para generar la alarma del contador. Si el elemento ya se encuentra creado da un mensaje de error 401 diciendo que el título ya se encuentra en uso.

- **/api/ruleChain/metadata:**

  Método POST donde le pasamos al parámetro ruleChainMetadata un objeto JSON donde le metemos las conexiones de la cadena de reglas, los nodos que se encuentran en ellas y las conexiones con las demás cadenas de reglas, es decir, llenamos la cadena de reglas con datos y nodos para que ejecute su función.

  Nos sirve tanto para actualizar la cadena de la alarma del contador como para cambiar el nivel de la alarma como en la cadena raíz para añadir el acceso a la cadena de la alarma.

  Si la petición es correcta te devuelve la nueva cadena de reglas que se ha implementado.

- **/api/ruleChains{?textSearch,idOffset,textOffset,limit}:**

  Método GET que devuelve una cadena de objetos donde vienen las cadenas de reglas que se encuentran en el servidor. En nuestro caso, lo utilizamos para obtener los id de las cadenas raíz y la alarma correspondiente a nuestro contador.

  Esto lo logramos mediante el parámetro opcional textSearch que busca dentro de la cadena de objetos un objeto con el mismo nombre que el parámetro. El resto de los parámetros también son opcionales.

- **/api/ruleChain/{ruleChainId}/metadata:**

  Método GET que devuelve una cadena de objetos donde vienen los datos de las cadenas de reglas que se encuentran en el servidor. Sirve para que la aplicación pueda actualizar las diferentes cadenas de reglas al cambiar los datos devueltos en esta petición.

  El parámetro ruleChainId nos permite escoger la cadena de reglas de la que queremos obtener los datos, para así poder añadir nodos o cambiar datos y descartar el resto de las cadenas de reglas.

4.2.3.2.8 Controlador de usuario

En estas APIs vamos a realizar todas las peticiones relativas a las distintas acciones que se va a realizar con respecto al usuario, en nuestro caso sería la creación del usuario, obtener los usuarios que se han creado en el servidor y la obtención de los datos relativos a un usuario. Dentro de este controlador utilizamos las siguientes APIs:

- **/api/customer/{customerId}/users{?textSearch,idOffset,textOffset,limit}:**

  Método GET que devuelve una cadena de objetos donde vienen los usuarios que tiene el cliente. Lo usamos para obtener el id del usuario asociado a ese cliente mediante su email. Si no tiene ninguno, el API devuelve una cadena vacía.

  El parámetro limit es obligatorio e indica el número máximo de usuarios que queremos buscar. Como cada cliente tiene un usuario. El límite es de uno.

  El resto de los parámetros son opcionales y sirven para buscar con mayor precisión el usuario a buscar

- **/api/user/users{?textSearch,idOffset,textOffset,limit}:**

  Método GET que devuelve una cadena de objetos donde vienen los usuarios que tiene el cliente. Lo usamos para obtener los datos del usuario. Si no tiene ninguno, el API devuelve una cadena vacía.

  Esto lo logramos mediante el parámetro opcional textSearch que busca dentro de la cadena de objetos un objeto con el mismo email que el parámetro. El resto de los parámetros también son opcionales.

  El parámetro limit es obligatorio e indica el número máximo de usuarios que queremos buscar. Como cada cliente tiene un usuario. El límite es de uno. El parámetro customerId sirve para identificar el cliente por su identificación única.

  El resto de los parámetros son opcionales y sirven para buscar con mayor precisión el usuario a buscar.

- **/api/user/{userId}/activationLink:**

  Método GET que devuelve un string que contiene el link para activar la cuenta del usuario. El parámetro userId nos sirve para identificar el usuario que queremos activar.

  De este link sacamos el token de activación que luego usaremos para activar la cuenta.

- **/api/user{?sendActivationMail,entityGroupId}:**

  Método POST donde le pasamos como parámetro user un objeto JSON y te crea el usuario. Si la petición es correcta, te devuelve un objeto con los datos generados al crear el usuario. Si al pasar el objeto user le pasamos un objeto con la misma id que un usuario creado, el usuario se actualiza al que se le ha pasado. Esto es útil a la hora de cambiar los datos del usuario que queremos implementar sin que se borren los datos asociados al mismo.

  El parámetro semActivationMail lo ponemos como falso ya que queremos que la aplicación active la cuenta sin que tenga el usuario de activarlo por correo.

  El parámetro entityGroupId sirve para identificar el grupo donde queremos meter el usuario.

  Además, para que el usuario se asocie con el cliente, tenemos que darle la autoridad al usuario como "CUSTOMER_USER" y luego asociarlo con el cliente que se ha creado anteriormente.

  Si el elemento ya se encuentra creado da un mensaje de error 401 diciendo que el título ya se encuentra en uso. El resto de los parámetros son opcionales.

4.3 ThingsBoard

Hasta ahora hemos visto que elementos en el lado de la aplicación se han utilizado hasta ahora. Sin embargo, como ya hemos en el apartado 3. La aplicación se conecta con la plataforma ThingsBoard, pero no utiliza todos los elementos de esta ya que es una plataforma que sirve para múltiples dispositivos inteligentes.

Para acceder a la plataforma, la aplicación realiza una petición a la URL http://207.180.221.115:8080. Si accedemos a esta URL mediante el navegador llegaremos a la plataforma de ThingsBoard donde, tras meter el usuario y la contraseña, accederemos a nuestro panel de control de esta.

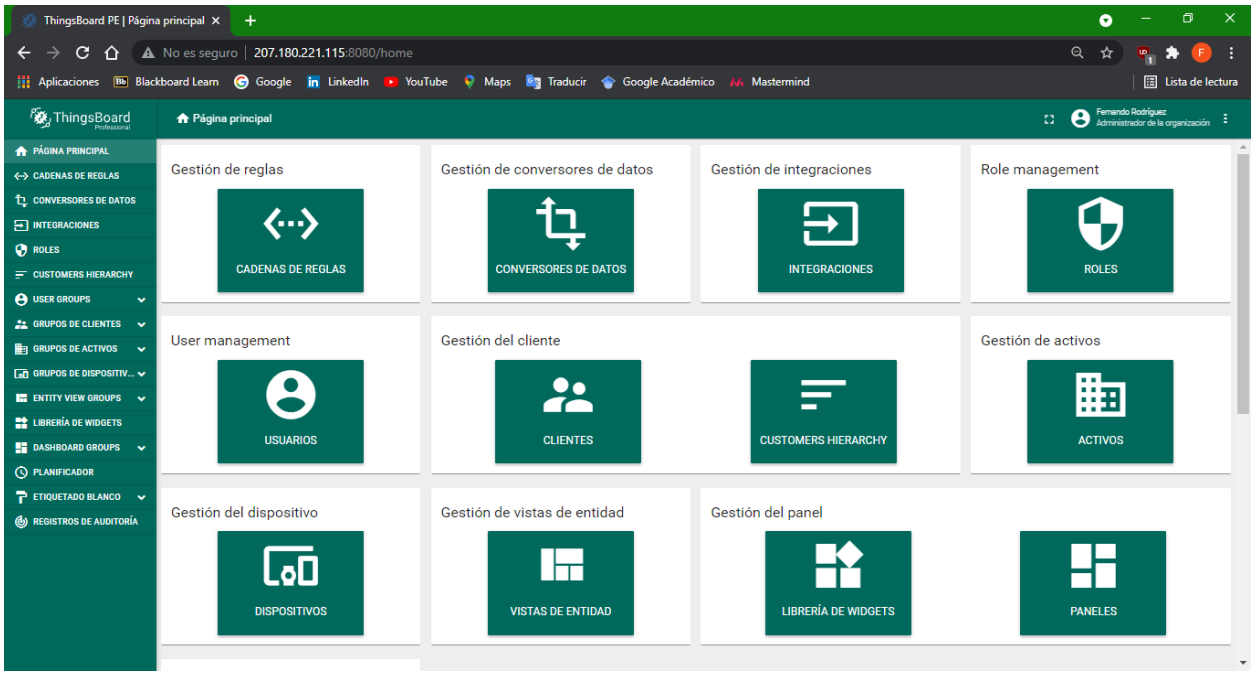

Figura 4-43. Panel de control de ThingsBoard

Por lo tanto, en este apartado veremos qué aspectos de la plataforma son las que la aplicación maneja, de forma que miramos como lo trabaja y cuál es el resultado final de esta. Para ello, hemos dividido este apartado en los diferentes aspectos que tiene la plataforma a la que accede la aplicación.

4.3.1 Cadenas de reglas

Una vez que se ha creado el dispositivo, la aplicación crea una cadena de reglas que genera una alarma cada vez que se sobrepasa cierto nivel de agua. Una vez creado la alarma, esta aparece en el apartado de cadenas de reglas de la plataforma.

Para que la identificación de la alarma sea más sencilla, a la cadena se la nombra de “Alarm” seguida del nombre del contador de modo que queda una lista de alarmas con todos los contadores creados.

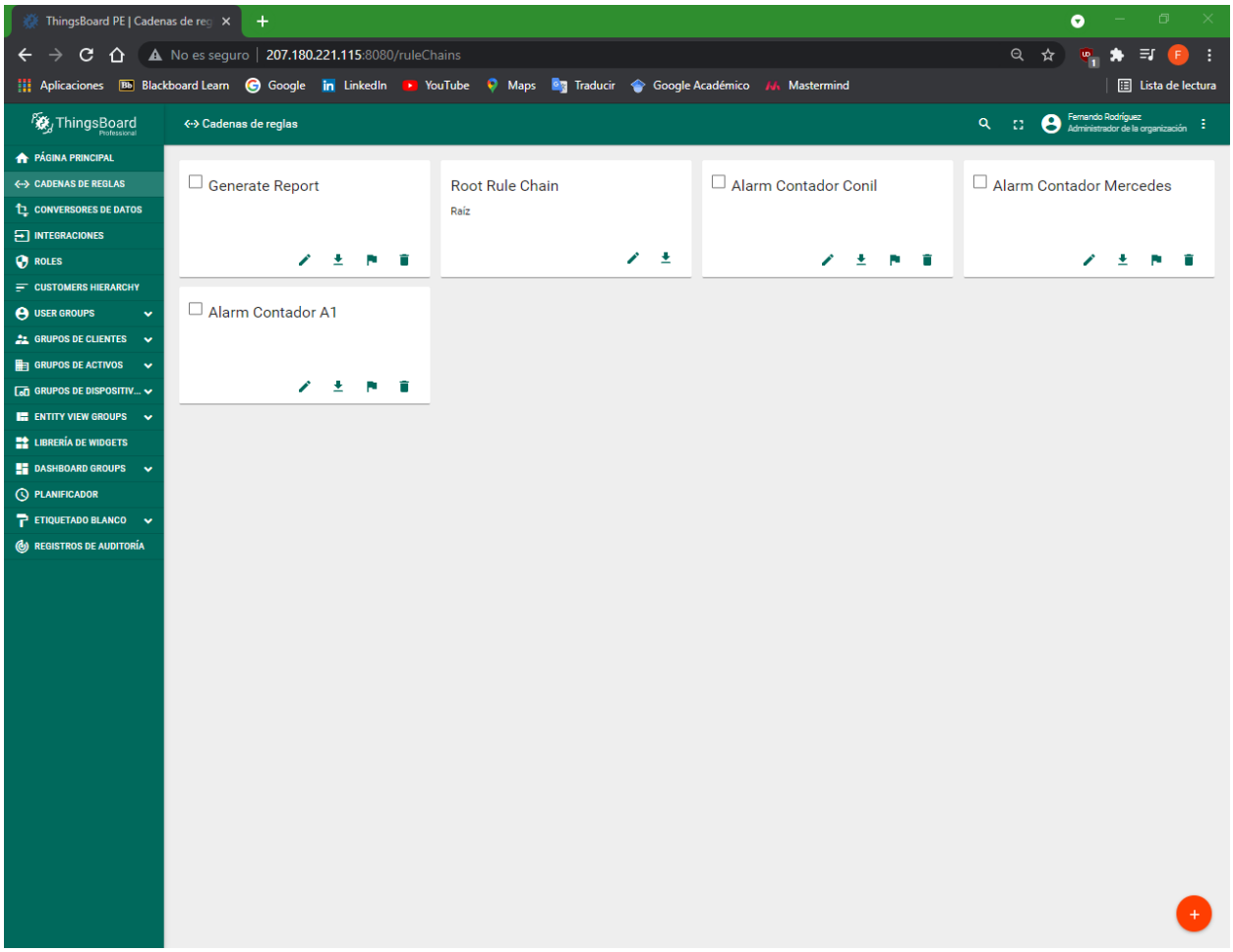

Figura 4-44. Grupo de cadenas de reglas creadas

Si accedemos a la cadena de reglas de una alarma veremos el flujo que sigue la cadena de esta. Una vez que el mensaje entra, vemos que el primer nodo comprueba si es el contador correcto, esto sirve para identificar si es el contador correcto el que envía la telemetría.

Figura 4-45. Comprobación del dispositivo

Si es el contador, comprueba si la cantidad de agua es superior a la indicada. Si no lo supera, se limpia la alarma y se acaba la cadena.

Figura 4-46. Comprobación del nivel

Sin embargo, si la cantidad supera el nivel, se crea un script con los datos y se crea la alarma. Una vez creada la alarma creamos el email a enviar al usuario y lo enviamos.

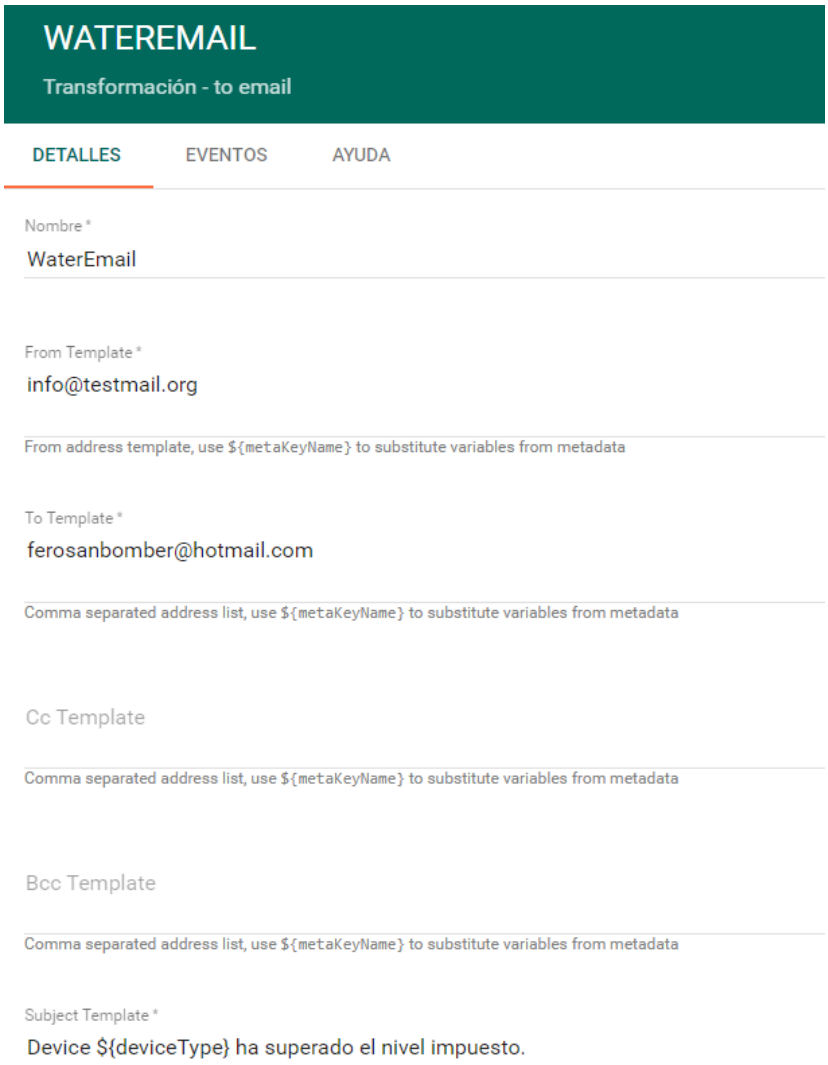

Figura 4-47. Creación del email

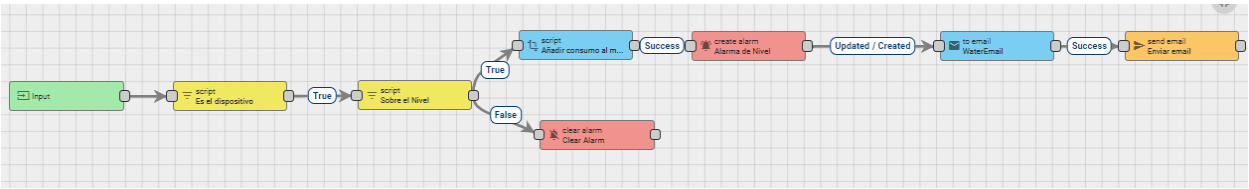

Figura 4-48. Cadena de reglas de la alarma.

Sin embargo, se accede a esta cadena de reglas mediante la cadena de reglas raíz, que es el que maneja la entrada y desde esta cadena se accede a la cadena de reglas correspondiente. Por lo tanto, tenemos que añadir a la cadena raíz el acceso a la cadena alarma para que le pueda pasar el mensaje. Esto la aplicación lo realiza añadiendo un nodo de cadena de reglas a la cadena raíz.

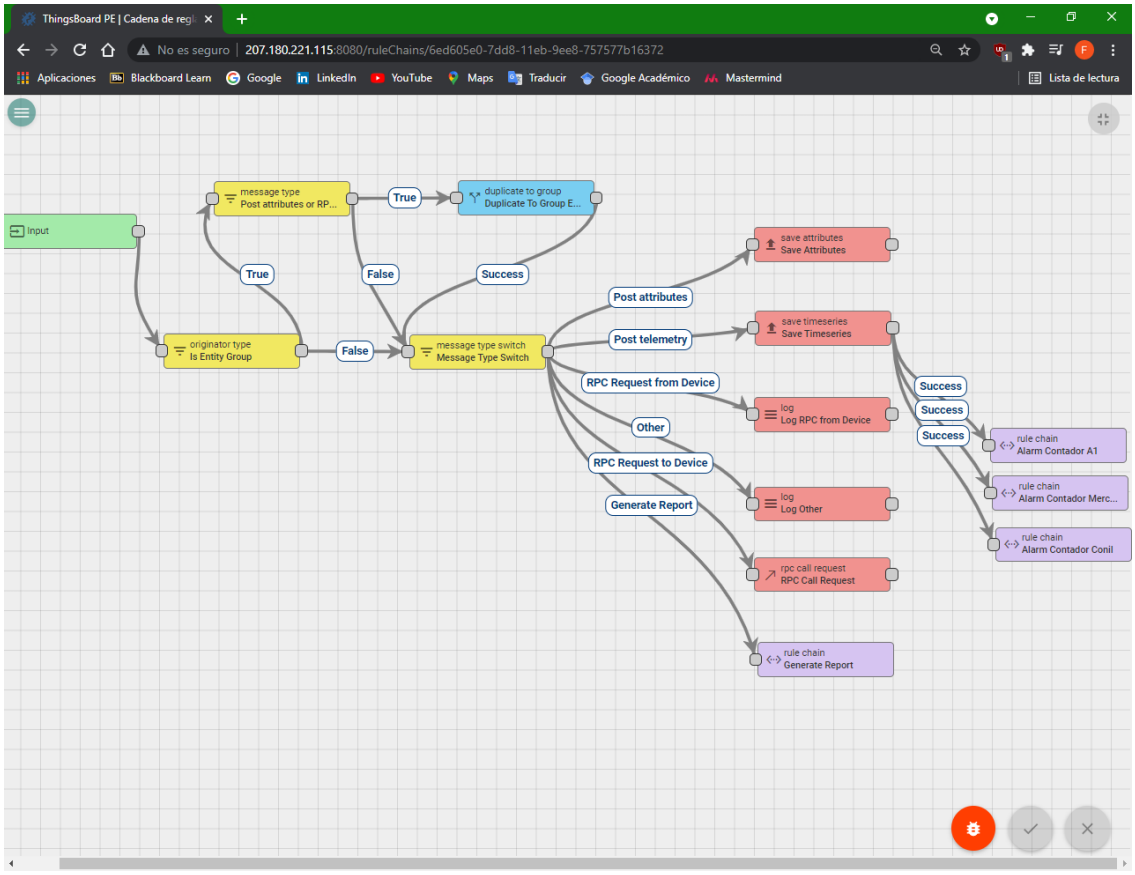

Figura 4-49. Cadena de reglas raíz.

4.3.2 Jerarquía de clientes

Para que la aplicación funcione, crea un cliente por cada persona que se registra. Cuando se ha creado el cliente, la aplicación asigna un usuario al cliente y le autoriza. Luego asigna a este usuario un dispositivo que es de tipo “wáter-meter” y al que le añade una serie de atributos.

Cuando accedemos al apartado de jerarquía de clientes, este despliega todos los clientes que se han creado en el servidor de modo que tenemos un panel al que acceder a todos los apartados de los distintos clientes.

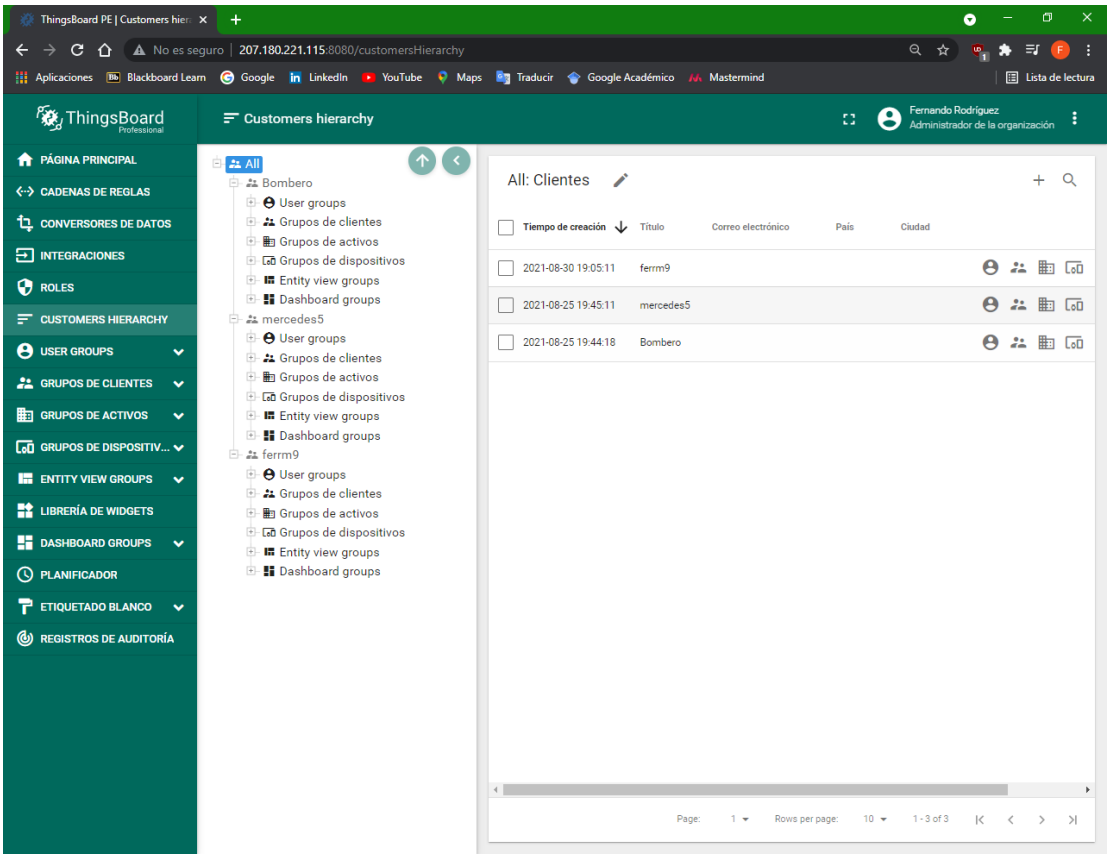

Figura 4-50. Jerarquía de clientes

Una vez que accedemos a los grupos de usuarios asignados a cada cliente podemos ver los grupos de los usuarios, para que el usuario pueda manejar los datos de los dispositivos, la aplicación lo asignó al grupo de Customers Administratos, que tiene la mayoría de los permisos. En este grupo se encuentran todos los usuarios asignados al cliente que la aplicación ha creado. Estos usuarios se identifican por el email, el nombre y los apellidos.

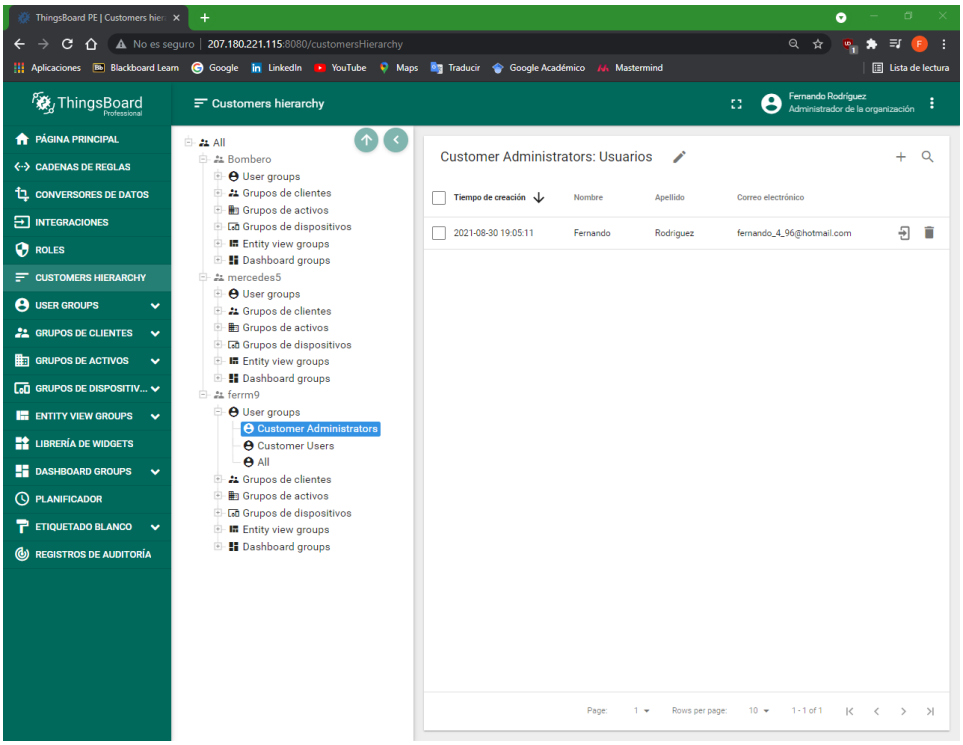

Figura 4-51. Usuario del cliente

Una vez que se ha creado el usuario, la aplicación es la que gestiona la activación de la cuenta, por lo que no necesita que el usuario active la cuenta en el servidor de ThingsBoard. Si accedemos a la cuenta del usuario, vemos que tiene menos elementos que el administrador, ya que goza de menos privilegios y permisos.

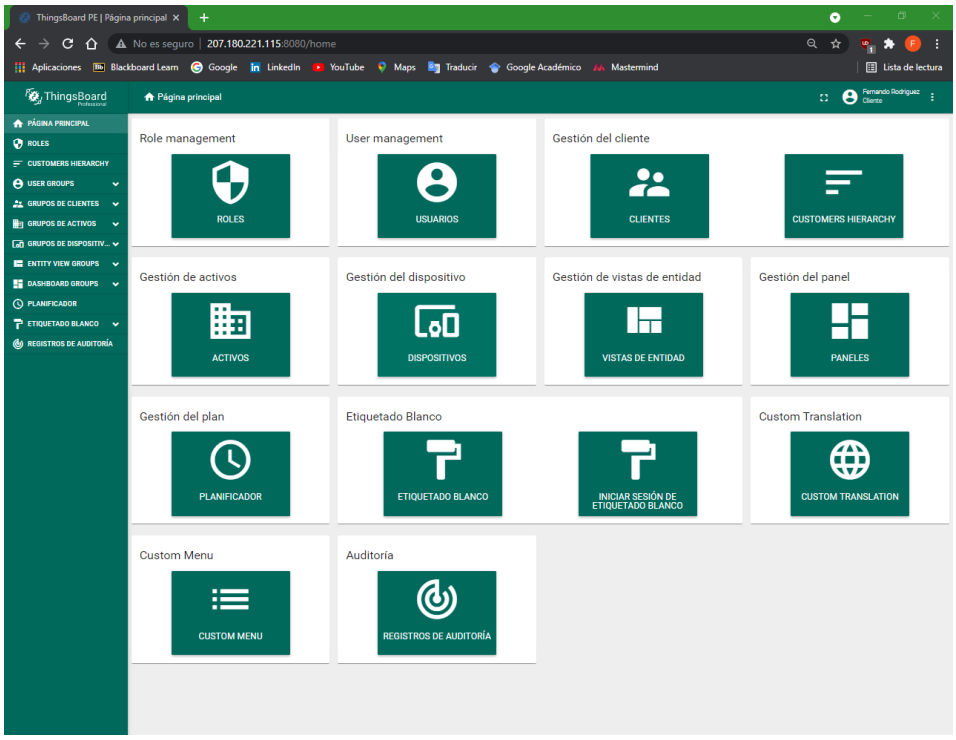

Figura 4-52. Cuenta de usuario activado

4.3.3 Dispositivos

Una vez que se ha creado el usuario, la aplicación crea el contador en la cuenta de este. Para comprobarlo accedemos al apartado de grupo de dispositivos donde en el grupo de todos podemos observar el dispositivo creado.

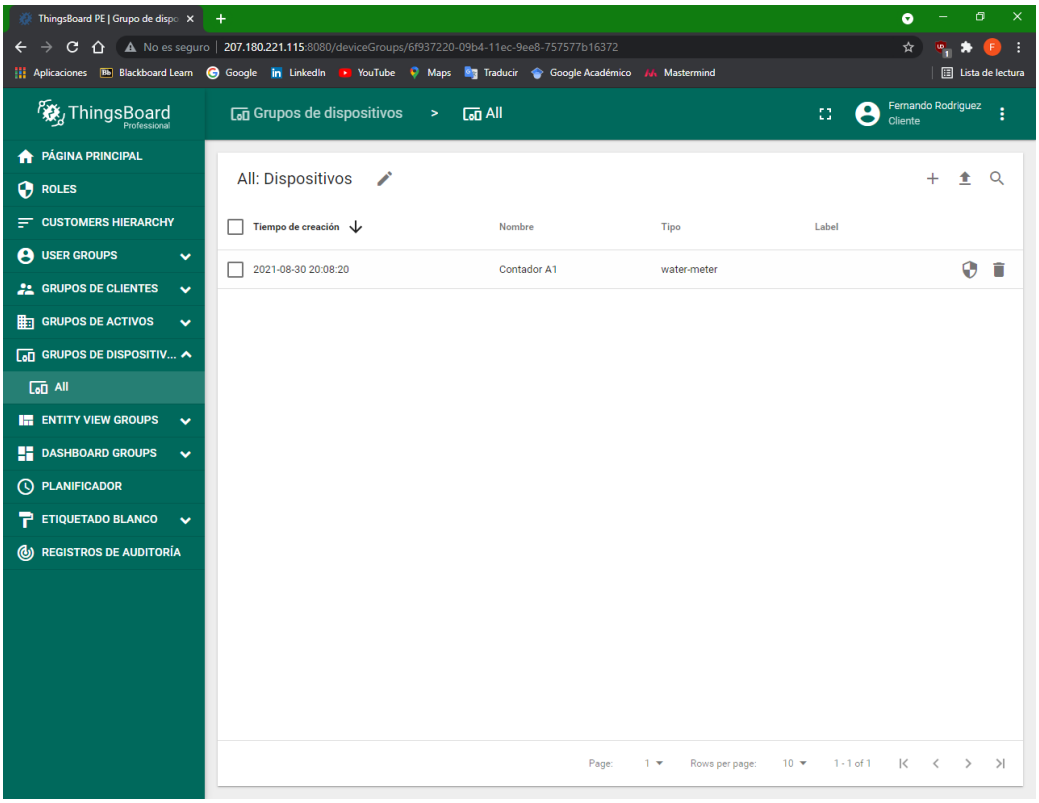

Figura 4-53. Contador asignado al usuario

El contador, al estar asignado al usuario y este asignado a un cliente, puede ser visto desde la cuenta del administrador en la jerarquía de clientes. Para ello, nos vamos al cliente del cual tenemos el dispositivo y nos vamos al grupo de dispositivos que maneja, donde podemos observar el dispositivo.

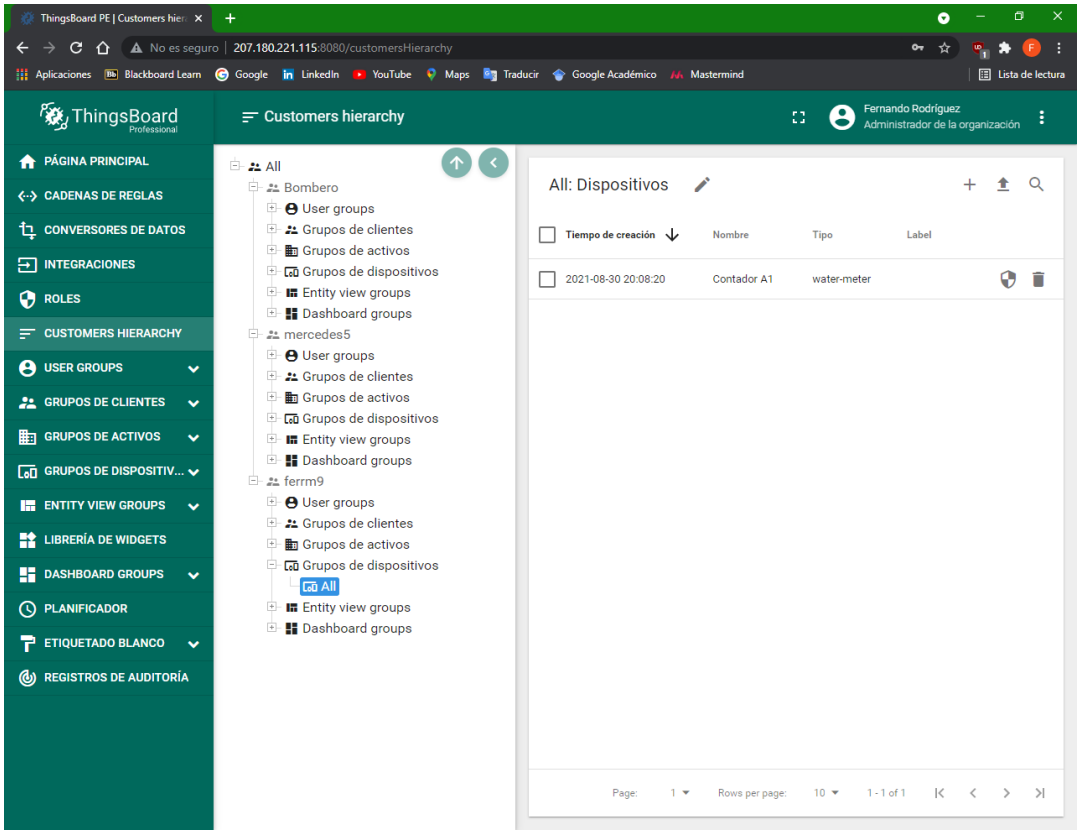

Figura 4-54. Grupo de dispositivos del cliente.

Una vez que se ha creado el contador, la aplicación asigna los atributos que se van a utilizar para comprobar si se encuentra activo o los distintos datos asignados al contador. Los atributos utilizados se dividen tanto en atributos del servidor como atributos compartidos.

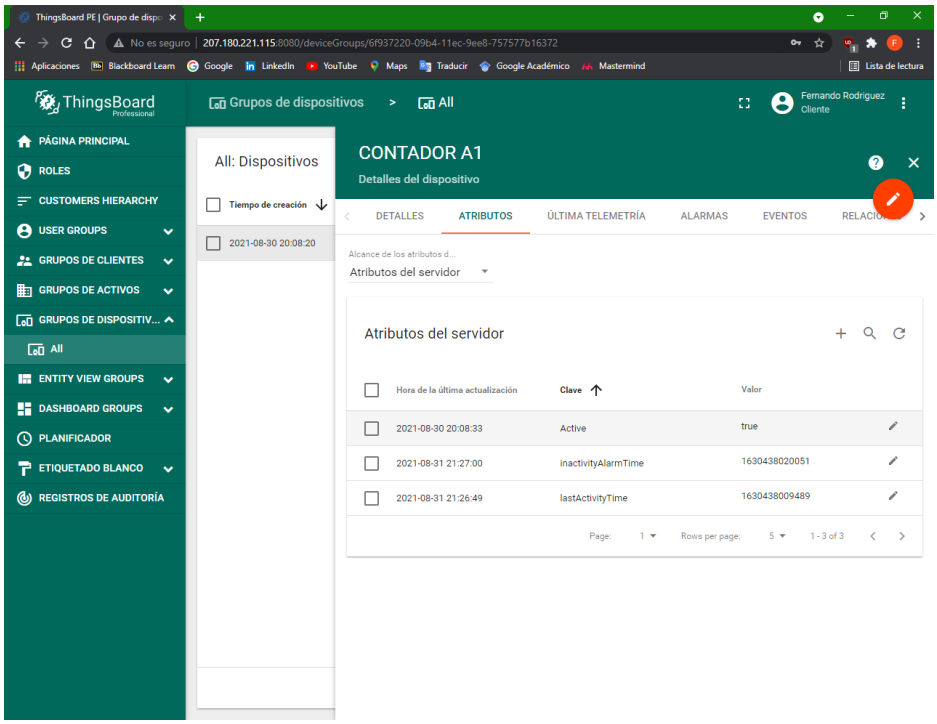

Figura 4-55. Atributos del servidor

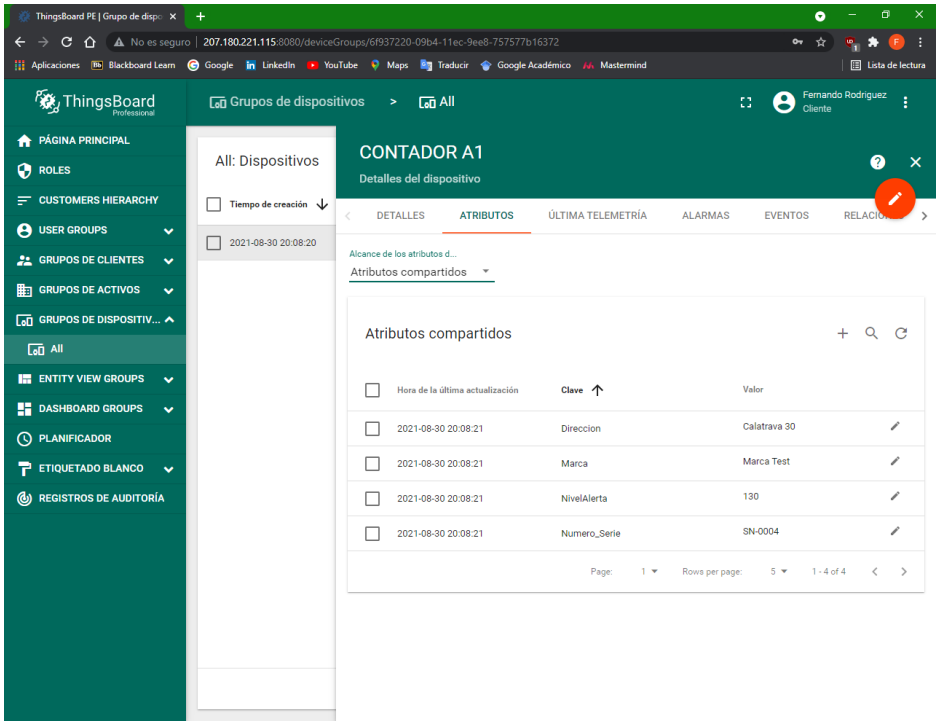

Figura 4-56. Atributos compartidos

Una vez rellenado todos los atributos y haciendo que el contador se encuentre activo. El servidor se encuentra en disposición de recoger todos los datos disponibles del contador. Se guardan en el apartado de la última telemetría donde se recoge la última lectura del contador en una variable llamada watercounter.

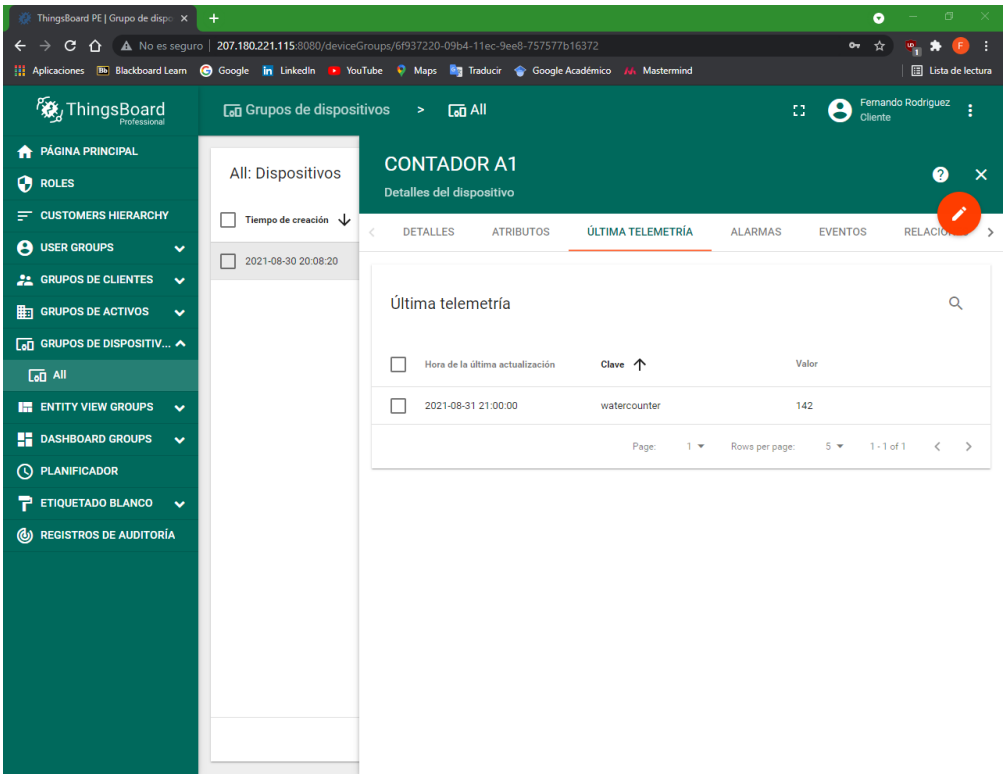

Figura 4-57. Última telemetría registrada

Una vez que se guardan las telemetrías, la cadena de alarma comprueba si esta supera el nivel máximo permitido. En el caso de que lo supere, se genera una alarma que se guarda en el apartado de alarmas del contador, mostrando una tabla de alarmas que se han creado a lo largo de un periodo que es personalizable.

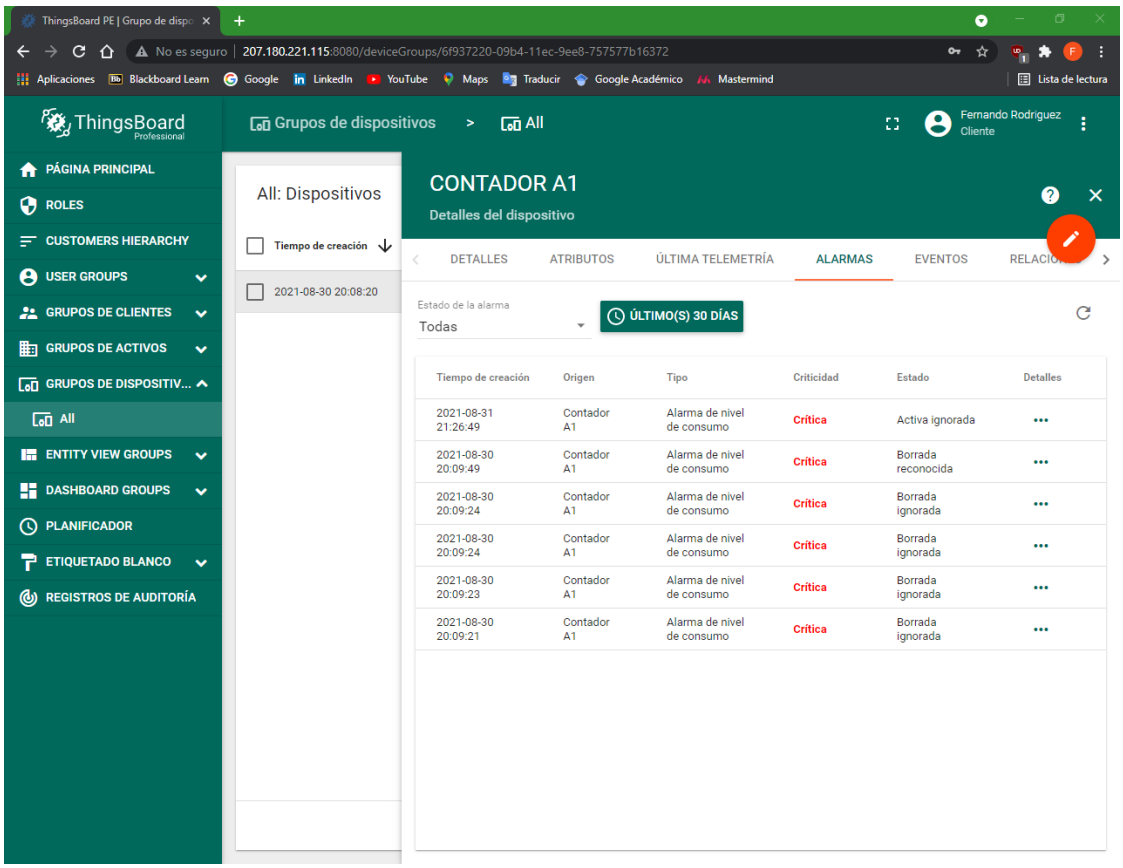

Figura 4-58. Alarmas generadas por el contador

4.4 Visión global

Ya hemos aclarado en los anteriores apartados tanto la utilización de la aplicación de React Native para realizar el proyecto como el uso del servidor de ThingsBoard en este. Por lo tanto, solo queda dar una visión global de cómo funciona la aplicación en el dispositivo móvil.

En este apartado vamos a dar una visión conjunta del funcionamiento de la aplicación, mostrando pantalla por pantalla como se va realizando. De este modo se va a ver como se implementan conjuntamente los aspectos mencionados en los anteriores apartados.

Una vez iniciado la aplicación sale la pantalla de inicio de sesión. En esta pantalla tenemos que acceder introduciendo el email del usuario y la contraseña.

Figura 4-59. Inicio de Sesión

Si el usuario o la contraseña no son introducidos salta un mensaje diciendo que falta por introducir datos. Si algunos de los datos son erróneos, entonces el mensaje de error dice que no existe el usuario o la contraseña.

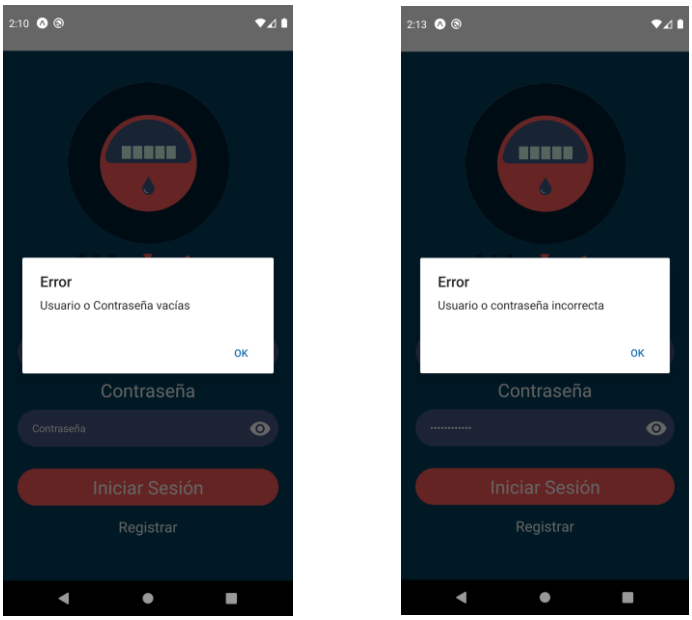

Figura 4-60. Mensajes de error

Para poder registrar un nuevo usuario, se pulsa en la opción de Registrar, donde se accede a la pantalla de registro del usuario, en esta pantalla se ven los campos de registro de los usuarios como son el Alias, Nombre, Apellidos, Email y Contraseña. Como medida de seguridad, se ha pedido que se repita la contraseña para que el usuario se encuentre seguro que ha introducido la contraseña que desea.

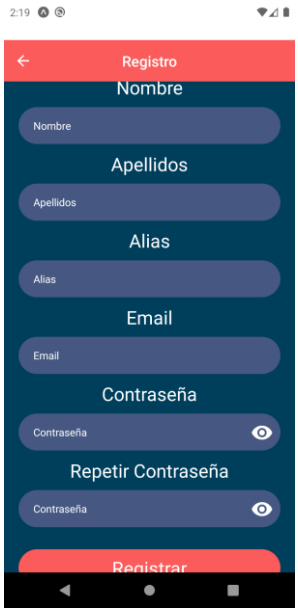

Figura 4-61. Registro del usuario

Los mensajes de error se producen como en el caso anterior, si no se introduce algún elemento, si el usuario a crear ya existe o, en este caso, si la contraseña y la contraseña de seguridad no coinciden. De este modo se genera una alerta con el mensaje de error correspondiente.

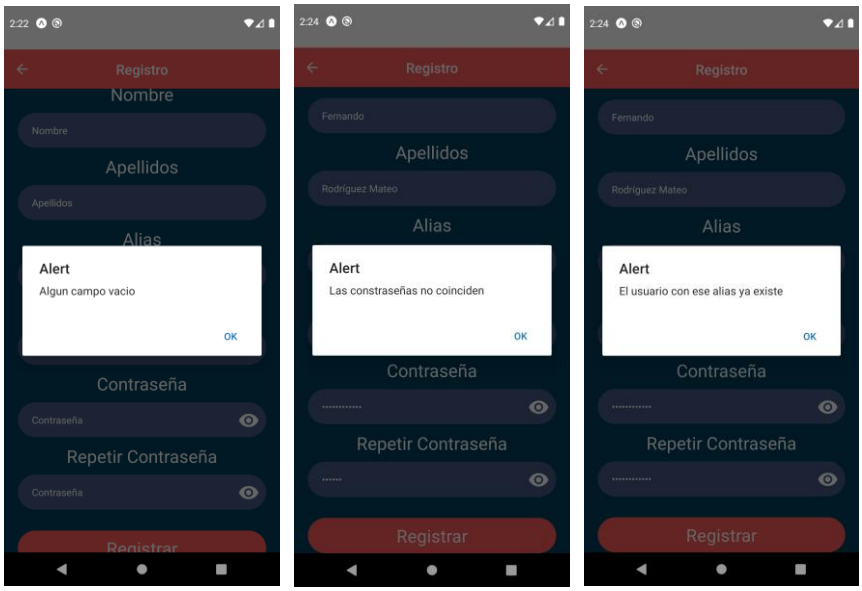

Figura 4-62. Mensajes de error

Si el usuario es creado correctamente, se vuelve a la pantalla de inicio de sesión con la notificación de que el usuario ha sido creado. Además, le llegará un correo a su cuenta diciendo que su usuario ha sido activado.

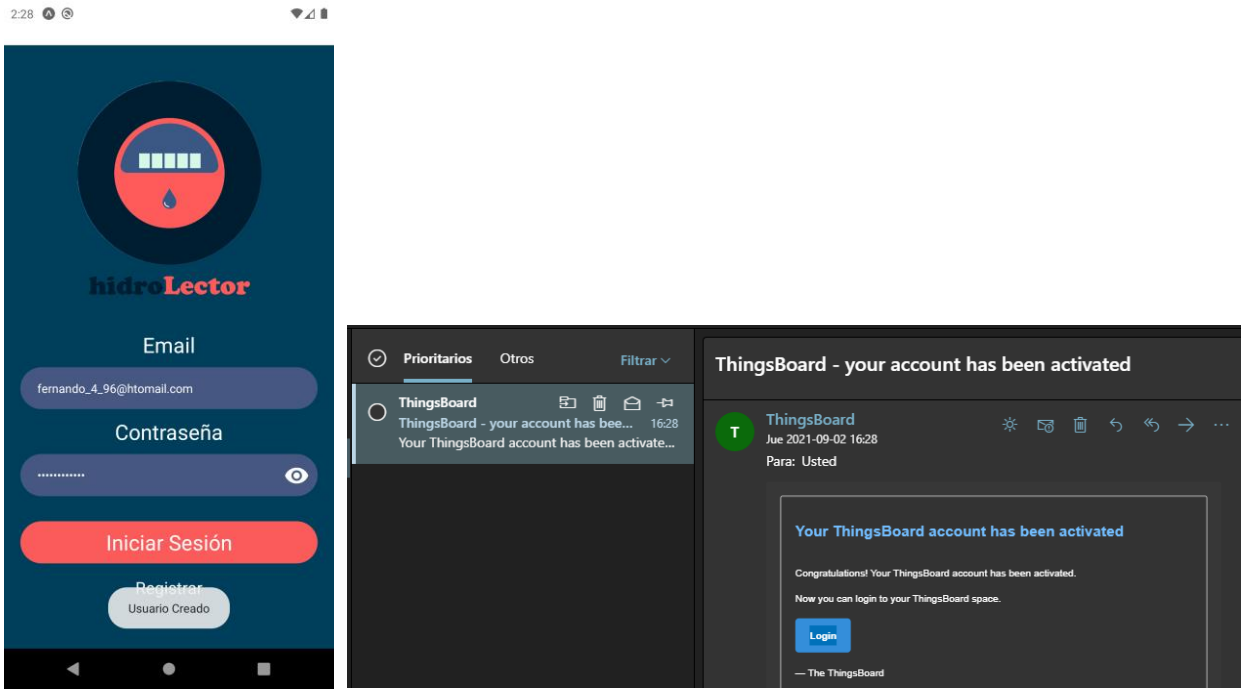

Figura 4-63. Usuario creado y cuenta activada

Una vez iniciado sesión con una cuenta activada, la cuenta te lleva al menú de la aplicación. Este menú está compuesto por cuatro botones donde se acceden al contador, a los informes del contador, a la configuración y al perfil respectivamente.

Figura 4-64. Menú de la aplicación

Si intentamos acceder a cualquier botón que no sea el de perfil cuando aún no hemos creado nuestro contador nos aparecerá un mensaje de error indicándonos que accedamos al perfil para registrar el contador.

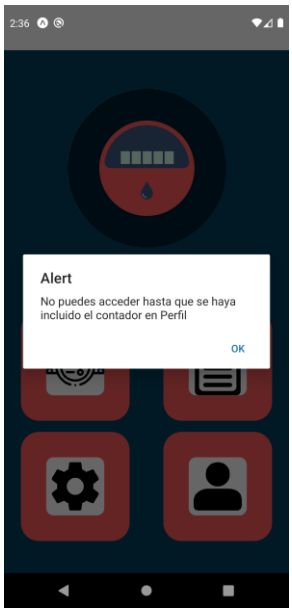

Figura 4-65. Mensaje de error

Una vez que se ha accedido al perfil para registrar el contador, aparecerá una pantalla donde se muestran los datos del usuario un botón para acceder al formulario de registro del contador. Cuando lo pulsamos aparece el formulario donde hay que introducir el nombre que le queremos asignar al dispositivo, la marca de este, el número de serie asignado y la dirección donde se encuentra instalada.

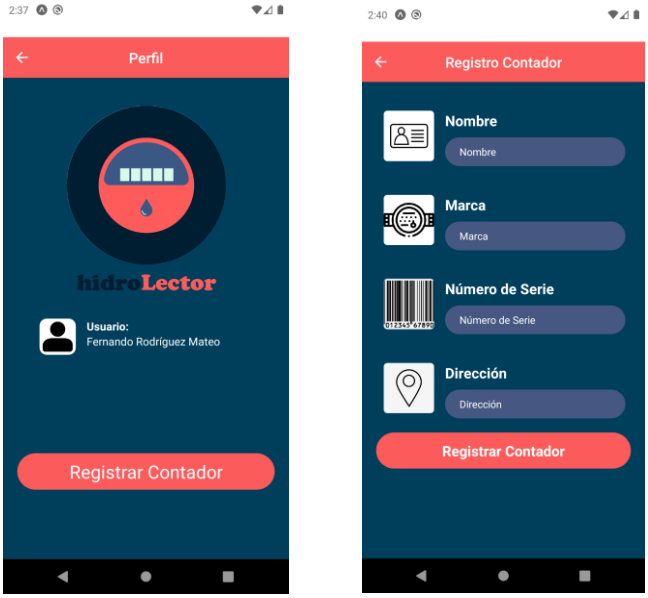

Figura 4-66. Registrar Contador

Los mensajes de error se producen si se ha dejado algún campo vacío o si ha habido algún fallo en la creación del dispositivo. En cuyo caso se muestra una alerta con el mensaje de error correspondiente al fallo.

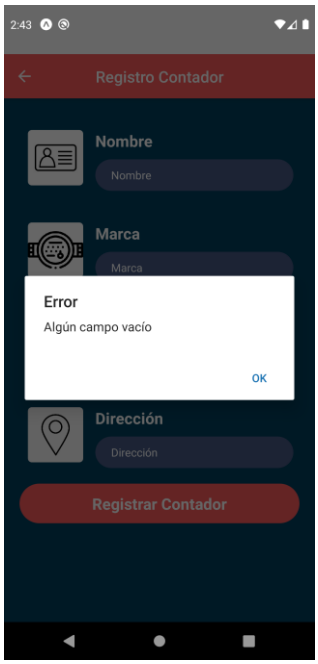

Figura 4-67. Mensaje de error

Si se ha realizado la creación de un contador correctamente, la aplicación vuelve a la pantalla de menú con una notificación indicando que el dispositivo se ha creado correctamente.

Figura 4-68. Dispositivo creado correctamente

Una vez que el dispositivo ha sido creado, en el apartado de perfil podemos ver una pantalla con los datos de tanto del usuario como del contador. Además, se observa un botón para salir de la aplicación que nos permite volver a la pantalla de inicio de sesión saliendo de nuestra cuenta.

Figura 4-69. Pantalla de perfil

En el apartado de Contador podemos observar tres gráficas. En la primera aparece la suma de la cantidad de agua que se ha consumido por litros en durante una semana. En la segunda, aparece la cantidad de agua que se ha consumido en un mes. Por último, en la tercera se observa la cantidad de agua que se ha consumido en el día hasta el momento.

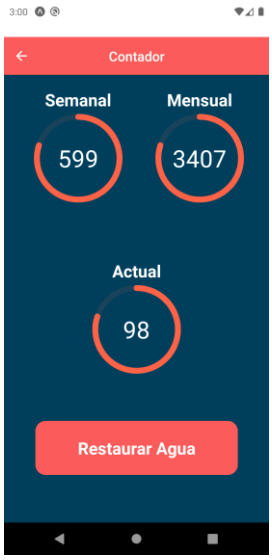

Figura 4-70. Pantalla del contador

Debajo de estas gráficas, aparece un botón que sirve para cortar o activar el agua. Esto se realiza cambiando el atributo Active en el lado servidor del dispositivo en el servidor ThingsBoard. Este cambio se produce a la vez de una notificación indicando la acción realizada.

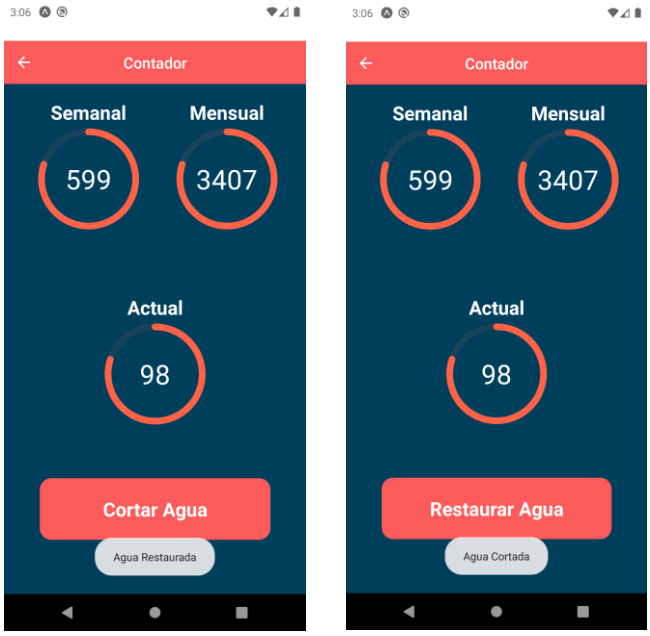

Figura 4-71. Corte o Activación del agua

En el menú, cuando accedemos a la pantalla de informes mediante el botón correspondiente, aparece la pantalla que muestra una gráfica de barras indicando el consumo de los últimos 7 días. Debajo de la gráfica nos muestran dos botones que se utilizan para cambiar entre la gráfica de barras a la tabla de alarmas y un switch que nos permite aplicar comparativas, que si se activa nos muestran las opciones de aplicar una comparativa visual de los datos de esta semana con los datos de la semana anterior o con respecto hace una quincena.

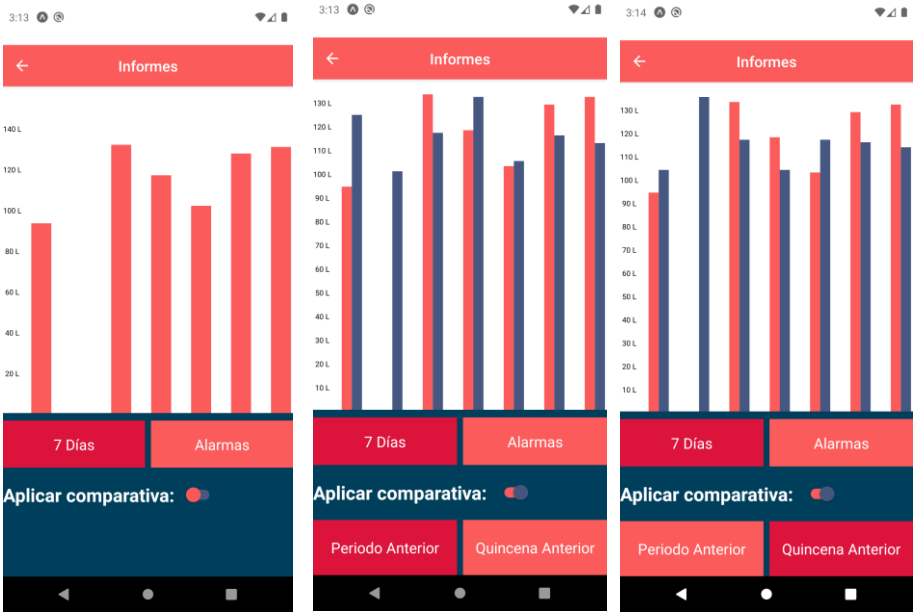

Figura 4-72. Distintas gráficas de barras

Si activamos el botón de alarmas, aparecerá una tabla con las alarmas que se han activado al consumir el agua. Como la tabla es muy grande para la pantalla del dispositivo móvil, se puede realizar un scroll horizontal para ver todos los datos. La alarma también se notifica por correo.

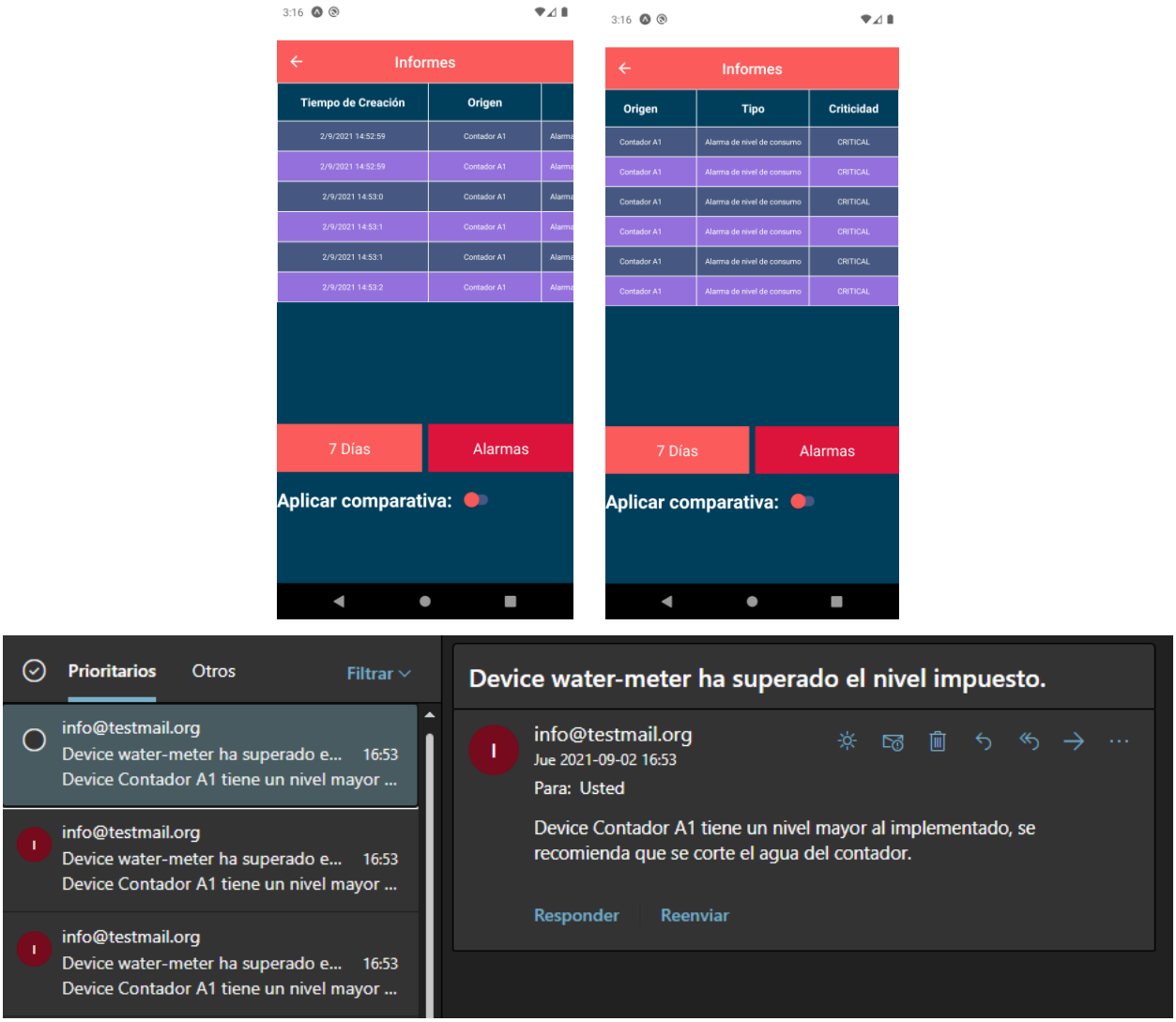

Figura 4-73. Tabla de alarmas y el correo

En el apartado de configuración tenemos dos botones para poder configurar dos elementos fundamentales. El primero accede a la configuración de los informes de consumo, es decir, sirve para configurar el nivel en el que el contador genera una alarma. El segundo caso tiene la configuración tanto del usuario como del contador asociado a él.

Figura 4-74. Pantallas de configuración

En la pantalla de configuración del informe tenemos dos opciones: o aplicamos el nivel por defecto o aplicamos el nivel personalizado, permitiendo poner la cantidad de litros máximo por el que la aplicación genera una alerta. Al pulsar el botón de configuración, la aplicación cambia el atributo de NivelAlerta por el que establecemos en la aplicación. Cuando cambiamos el nivel, se vuelve a la pantalla del menú con una notificación de que la configuración ha sido modificada.

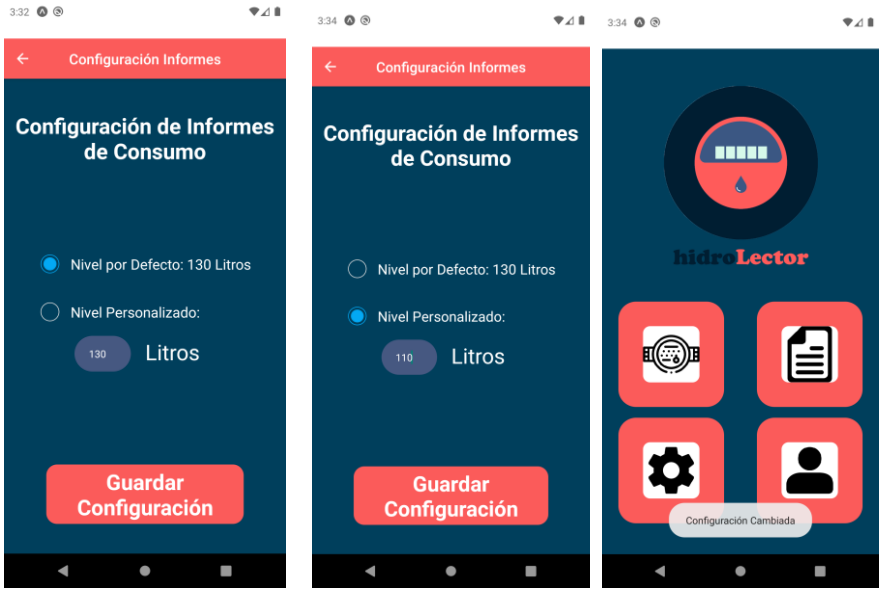

Figura 4-75. Cambio de configuración

En el apartado de la configuración del usuario aparece un formulario con los distintos elementos que se pueden modificar tanto del usuario como del contador. Del usuario tenemos el Nombre, los apellidos, el alias y el email. Del contador podemos cambiar la marca el Nombre, la Marca y la Dirección de instalación.

En este caso, no hace falta llenar toda la configuración para poder cambiarla, basta con llenar los campos que se quieren modificar para poder realizar el cambio. Solo se produce un mensaje de error si todos los campos se encuentran vacíos, provocando un mensaje de error indicándolo.

Figura 4-76. Mensaje de error

Si el cambio se ha producido correctamente, se han cambiado los datos puestos en el servidor de ThingsBoard. Luego, se ha vuelto a la pantalla del menú con una notificación que indica que la configuración ha sido modificada. Estos cambios lo podemos ver de nuevo en la pantalla de perfil.

Figura 4-77. Cambios del usuario y contador

5 GENERACIÓN DE DATOS

Ya hemos hablado de los elementos que componen la aplicación, tanto de la parte de React Native como de la parte de ThingsBoard. También hemos hablado del servidor ThingsBoard en sí mismo y como la aplicación es capaz de comunicarse con ella. Por último, hemos visto globalmente la aplicación. Sin embargo, no hemos hablado de como los contadores generan los datos y se los comunican al servidor.

Como este proyecto no tiene contadores reales, hemos tenido que simular los efectos que tendría uno sobre la aplicación. Para ello hemos creado códigos que generen datos que simulen una lectura de agua que luego hemos enviado a ThingsBoard para que la aplicación pueda recogerlos.

En este apartado vamos a desarrollar los códigos producidos y como estos se han conectado a ThingsBoard.

5.1 Conexión con ThingsBoard

Para poder enviar los datos hemos realizados peticiones API a la URL del servidor de ThingsBoard. Estas peticiones se realizan mediante el protocolo HTTP y con ellos hemos enviado los datos necesarios para que la aplicación sea funcional.

Cómo el código ha sido realizado en JavaScript, hemos realizado las peticiones mediante fetch. Un elemento que nos permite hacer peticiones HTTP a las diferentes URL junto con las API que se vayan a utilizar. Para instalar este elemento tenemos que realizar:

Paquete de fetch:

```
npm install -g node-fetch
```

Una vez instalado, en la aplicación se utiliza aplicando el siguiente procedimiento:

```
const fetch = require('node-fetch');
```

Figura 5-1. Incorporar fetch al código

Una vez aclarado el método para realizar las peticiones. A continuación, veremos las distintas APIs que se van a utilizar dependiendo del efecto que se tiene en la aplicación la llamada a los mismos.

5.1.1 /api/auth/login

Método POST al que le tenemos que pasar como datos el email del usuario y la contraseña de este y que si es correcto nos devuelve un objeto con el token que nos permite autorizar las diferentes acciones de nuestro usuario y autenticarnos.

```
async function  login() {
    const req = await fetch('http://207.180.221.115:8080/api/auth/login',{
        method: 'post',
        body: JSON.stringify({
            username: 'fernandor.1996@gmail.com',
            password: 'Fernando34;'
        }),
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
    }});
    let token = await req.json();
    return token.token;
}
```

Figura 5-2. Petición a login

5.1.2 /api/plugins/telemetry/DEVICE/{deviceId}/values/attributes/SERVER_SCOPE

Este método GET nos permite obtener los atributos del Contador del servidor ThingsBoard para así comprobar si se encuentra activo o no. Para ello, le pasamos como parámetros el id del dispositivo que queremos que genere datos y el token obtenido al logearnos por el método anterior.

```
async function  getAttribute(token ,deviceId) {
    const req = await fetch('http://207.180.221.115:8080/api/plugins/telemetry/DEVICE/'+deviceId+'/v
        method: 'get',
        headers: {
            'Content-Type': 'application/json',
            'x-authorization': 'Bearer ' + token
    });
    let attrib = await req.json();
    return attrib;
}
```

Figura 5-3. Petición al dispositivo

Si todo sale correcto, tendrás una cadena con los atributos el cual tendrás que sacar el que corresponda a Active.

5.1.3 /api/v1/{accessToken}/telemetry

Es un método POST que genera los datos que queremos entregar. Como parámetros le pasas el token de acceso del dispositivo que genera los datos y de ahí le entregamos un JSON donde le pasamos la variable que se va a generar con el dato producido y opcionalmente, el timeseries con el tiempo en UNIX del momento donde se genera el dato.

```
async function  postTelemetry(accessToken, timeStamp, data) {
    const req = await fetch('http://207.180.221.115:8080/api/v1/'+accessToken+'/telemetry',{
        method: 'post',
        body: JSON.stringify({
            ts: timeStamp,
            values: {
                watercounter: data
            }
        }),
        headers: {
            'Content-Type': 'application/json'
    }});
    let telemetry = req.status;
    return telemetry;
}
```

Figura 5-4. Entrega del dato

Si todo es correcto, en el servidor ThingsBoard se habrá actualizado los elementos que se han pasado como variables, en este caso la variable watercounter.

5.2 Códigos del generador de datos

Una vez que ya hemos analizado las peticiones que se han producido entre el contador simulado y el servidor ThingsBoard, vamos a proceder con analizar los códigos que se han creado para simular los datos del dispositivo.

Hemos creado dos códigos que, aunque son ciertamente parecidos, producen dos efectos diferentes. El primero es un generador de datos del día, este código genera datos desde las 00:00 hasta la hora actual en periodos de una hora. El segundo es un generador de datos mensual, es decir, genera datos diariamente durante 30 días desde hace un mes atrás de la fecha actual.

A continuación, desarrollamos ambos códigos.

5.2.1 Generador de datos del día

Como se ha dicho anteriormente, este código genera un dato cada hora hasta la hora actual. Para ello, le pasamos al código el token de acceso al dispositivo y el id de este. Una vez que se ejecuta, el código comprueba mediante peticiones API si el dispositivo se encuentra activo.

Si el dispositivo no se encuentra activo, el código devuelve por pantalla un mensaje de error. Si el dispositivo se encuentra activo, lo primero que realiza es la obtención del momento actual en tiempo UNIX. Esto se realiza mediante el constructor Date. [16]

El constructor Date permite trabajar con fechas y horas. Para hacerlo creamos un objeto fecha y con el método now obtenemos el tiempo actual UNIX en milisegundos.

A continuación, calculamos el tiempo de inicio del código obteniendo la fecha actual, cogiendo la fecha y poniéndole las 00:00:00. Mediante el método getTime que devuelve ese tiempo en milisegundos.

Por último, enviamos datos aleatorios del dispositivo que se van acumulando a medida que avanza el bucle, que dura desde que empieza hasta la fecha actual añadiendo una hora en milisegundos al tiempo en UNIX

```
const fetch = require('node-fetch');

const deviceId = process.argv[2];
const accessToken = process.argv[3];

main(deviceId, accessToken);

//Funciones
async function main(deviceId, accessToken){
    const loginToken = await login();
    const attrib = await getAttribute(loginToken, deviceId);
    let active = false;
    let data = 0;

    for (var i = 0; i<attrib.length; i++){
        if (attrib[i].key == 'Active'){
            active = attrib[i].value;
        }
    }
    const timestampDay = obtenerDia();
    const timestampnow = Date.now();
    if(active){
        for (var timeStart = timestampDay; timeStart < timestampnow; timeStart = (timeStart + 60*60*1000)){
            const fecha = new Date (timeStart);
            console.log(fecha);
            data += random();
            console.log(data);
            const telemetry = await postTelemetry(accessToken, timeStart, data);
        }
    }
    else{
        console.log('El dispositivo no se encuentra activo');
    }
}

async function  login() {
    const req = await fetch('http://207.180.221.115:8080/api/auth/login',{
        method: 'post',
        body: JSON.stringify({
            username: 'fernandor.1996@gmail.com',
            password: 'Fernando34;'
        }),
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
    }});
```

```
    let token = await req.json();
    return token.token;
}

async function  getAttribute(token ,deviceId) {
    const req = await fetch('http://207.180.221.115:8080/api/plugins/telemetry/DEVICE/'+deviceId+'/values/attributes/SERVER_SCOPE',{
        method: 'get',
        headers: {
            'Content-Type': 'application/json',
            'x-authorization': 'Bearer ' + token
    }});
    let attrib = await req.json();
    return attrib;
}

async function  postTelemetry(accessToken, timeStamp, data) {
    const req = await fetch('http://207.180.221.115:8080/api/v1/'+accessToken+'/telemetry',{
        method: 'post',
        body: JSON.stringify({
            ts: timeStamp,
            values: {
                watercounter: data
            }
        }),
        headers: {
            'Content-Type': 'application/json'
    }});
    let telemetry = req.status;
    return telemetry;
}


function random() {
    return Math.floor(Math.random() * 15);
}

function obtenerDia () {
    const fecha = new Date();
    const day = fecha.getDate();
    const month = fecha.getMonth();
    const year = fecha.getFullYear();
    const hora = 0;
    const min = 0;
    const seg = 0;
    const newTime = new Date(year, month, day, hora, min, seg);
    return newTime.getTime();
```

Figura 5-5. Código del generatorDaily.js

Al ejecutar el código en la pantalla de la consola irán apareciendo los momentos que van generando los datos y el valor de estos en la línea de abajo.

```
C:\Users\Propietario\Desktop\Fernando\TFG\Trabajo\datos>node generatorDaily.js aa519460-0bfc-11ec-9ee8-757577b16372 bNrqWipSjEz2x05hvITJ
2021-09-01T22:00:00.000Z
7
2021-09-01T23:00:00.000Z
19
2021-09-02T00:00:00.000Z
28
2021-09-02T01:00:00.000Z
38
2021-09-02T02:00:00.000Z
49
2021-09-02T03:00:00.000Z
60
2021-09-02T04:00:00.000Z
68
2021-09-02T05:00:00.000Z
71
2021-09-02T06:00:00.000Z
82
2021-09-02T07:00:00.000Z
82
2021-09-02T08:00:00.000Z
88
2021-09-02T09:00:00.000Z
95
2021-09-02T10:00:00.000Z
107
2021-09-02T11:00:00.000Z
108
2021-09-02T12:00:00.000Z
114
2021-09-02T13:00:00.000Z
114
2021-09-02T14:00:00.000Z
125
2021-09-02T15:00:00.000Z
131
2021-09-02T16:00:00.000Z
132
2021-09-02T17:00:00.000Z
138
```

Figura 5-6. Ejecución del código

5.2.2 Generador de datos del mes

Este código es muy similar al anterior. Al ejecutarlo genera un dato cada día desde hace 30 días hasta la fecha actual. Para ello, le pasamos al código el token de acceso al dispositivo y el id de este. Una vez que se ejecuta, el código comprueba mediante peticiones API si el dispositivo se encuentra activo.

Si el dispositivo no se encuentra activo, el código devuelve por pantalla un mensaje de error. Si el dispositivo se encuentra activo, lo primero que realiza es la obtención del momento actual en tiempo UNIX. Esto se realiza mediante el constructor Date. [16]

El constructor Date permite trabajar con fechas y horas. Para hacerlo creamos un objeto fecha y con el método now obtenemos el tiempo actual UNIX en milisegundos.

A continuación, calculamos el tiempo de inicio del código obteniendo la fecha actual en UNIX y restándole 30 días en milisegundos.

Por último, enviamos datos aleatorios del dispositivo entre los valores 180 y 100 litros a medida que avanza el bucle, que dura desde que empieza hasta la fecha actual añadiendo una hora en milisegundos al tiempo en UNIX.

El código es el siguiente:

```
const fetch = require('node-fetch');

const deviceId = process.argv[2];
const accessToken = process.argv[3];

main(deviceId, accessToken);

//Funciones
async function main(deviceId, accessToken){
    const loginToken = await login();
    const attrib = await getAttribute(loginToken, deviceId);
    let active = false;

    for (var i = 0; i<attrib.length; i++){
        if (attrib[i].key == 'Active'){
            active = attrib[i].value;
        }
    }
    const timestampMonth = obtenerMes();
    const timestampnow = Date.now();
    console.log(timestampnow);
    console.log(timestampMonth);
    if(active){
        for (var timeStart = timestampMonth; timeStart < (timestampnow); timeStart = (timeStart + 24*60*60*1000)){
            const fecha = new Date (timeStart);
            console.log(fecha);
            const telemetry = await postTelemetry(accessToken, timeStart);
        }
    }
    else{
        console.log('El dispositivo no se encuentra activo');
    }
}

async function  login() {
    const req = await fetch('http://207.180.221.115:8080/api/auth/login',{
        method: 'post',
        body: JSON.stringify({
            username: 'fernandor.1996@gmail.com',
            password: 'Fernando34;'
        }),
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
    }});
```

```
    let token = await req.json();
    return token.token;
}

async function  getAttribute(token ,deviceId) {
    const req = await fetch('http://207.180.221.115:8080/api/plugins/telemetry/DEVICE/'+deviceId+'/values/attributes/SERVER_SCOPE',{
        method: 'get',
        headers: {
            'Content-Type': 'application/json',
            'x-authorization': 'Bearer ' + token
    }});
    let attrib = await req.json();
    return attrib;
}

async function  postTelemetry(accessToken, timeStamp) {
    const data = random();
    console.log(data);
    const req = await fetch('http://207.180.221.115:8080/api/v1/'+accessToken+'/telemetry',{
        method: 'post',
        body: JSON.stringify({
            ts: timeStamp,
            values: {
                watercounter: data
            }
        }),
        headers: {
            'Content-Type': 'application/json'
    }});
    let telemetry = req.status;
    return telemetry;
}


function random() {
    return Math.floor(Math.random() * (140 - 100) + 100);
}

function obtenerMes () {
    const timestamp = Date.now();
    const timestampMonth = timestamp - ((24*60*60*1000)*30);
    return timestampMonth;
}
```

Figura 5-7. Código de generatorMonth.js

Al ejecutar el código en la pantalla de la consola irán apareciendo las fechas que van generando los datos y el valor de estos en la línea de abajo junto con el tiempo inicial y final en UNIX.

```
Administrador: Símbolo del sistema
C:\Users\Propietario\Desktop\Fernando\TFG\Trabajo\datos>node generatorMonth.js aa519460-0bfc-11ec-9ee8-757577b16372 bNrqWipSjEz2x05hvITJ
1630594377612
1628002377612
2021-08-03T14:52:57.612Z
121
2021-08-04T14:52:57.612Z
109
2021-08-05T14:52:57.612Z
100
2021-08-06T14:52:57.612Z
134
2021-08-07T14:52:57.612Z
103
2021-08-08T14:52:57.612Z
106
2021-08-09T14:52:57.612Z
136
2021-08-10T14:52:57.612Z
118
2021-08-11T14:52:57.612Z
111
2021-08-12T14:52:57.612Z
116
2021-08-13T14:52:57.612Z
118
```

Figura 5-8. Ejecución del código

6 CONCLUSIONES

El diseño y desarrollo de una aplicación móvil es todo un reto técnico en las que suele emplearse equipos de proyectos completos para realizarlo. Poner en marcha una aplicación es un trabajo arduo ya que todo tiene que funcionar sin ningún error.

Que desde el inicio tus ideas y perspectivas casen con el funcionamiento es un trabajo complicado que suele llevar mucho tiempo. Por ello se suele realizar una aplicación sencilla que se pueda manejar y que si surge algún inconveniente se pueda solucionar sin mayor problema.

A nivel técnico ha sido todo un reto desarrollar esta aplicación desde cero utilizando una aplicación nueva y que hasta el momento era totalmente desconocida para mí como es React Native y ThingsBoard. No obstante, estoy muy satisfecho con el resultado obtenido y con haber alcanzado todos los objetivos que me propuse.

He realizado un trabajo arduo en aprender los conocimientos necesarios para, no solo manejar ambos, sino que se comuniquen entre sí y se pueda realizar el fin de este proyecto.

Estoy muy contento con la aplicación desarrollada. Creo que es una aplicación que pretende sentar las bases de cómo deben ser las futuras aplicaciones del mismo ámbito y del ámbito de las tecnologías inteligentes en general.

En cuanto a React Native, ha demostrado ser un entorno de desarrollo no demasiado complejo que contiene mucha documentación para realizar el proyecto. Permite desarrollar en alto nivel de manera eficaz y que sea adaptable a varios sistemas. La versatilidad para construir tus propios widgets es inmensa. Además, los distintos desarrolladores generan nuevas librerías que sirven para ampliar el catálogo de opciones de la aplicación.

Por otro lado, ThingsBoard ha sido un servidor muy eficaz y sencillo. El producto es muy fácil de utilizar y aporta una gran cantidad de opciones para manejar diferentes situaciones, por lo que demuestra ser un elemento muy completo que puede abarcar casi cualquier ámbito de los dispositivos inteligentes.

Como líneas de actuación futuras se podría añadir a la aplicación elementos como la disposición de más de un contador de agua, la capacidad de añadir a familias con distintos usuarios pero que posean el mismo dispositivo. Además, al ser ThingsBoard tan versátil, podemos incorporar a la aplicación nuevas funcionalidades.

REFERENCIAS

[1] J. Segarra, «Segarra Ingenieros SL,» [En línea]. Available: https://www.segarraingenieros.com/que-es-iot-y-el-futuro-tecnologico-que-nos-espera/.

[2] Iot Analytics, «State of IoT Q4/2020 & Outlook 2021,» 2020.

[3] Ubidots, «Ubidots,» [En línea]. Available: https://ubidots.com/blog/2020-the-year-of-iot/.

[4] «El aumento de medidores digitales de agua impulsa una nueva forma de gestión de datos,» *itTrends,* 1 Agosto 2019.

[5] R. Martinez Jacobson, «Comparativa y estudio de plataformas IoT,» 2017.

[6] React Native, «React Navigation,» [En línea]. Available: https://reactnavigation.org/docs/getting-started/.

[7] React Native , «Core Components and APIs,» [En línea]. Available: https://reactnative.dev/docs/components-and-apis.

[8] npm, «react-native-svg-charts,» [En línea]. Available: https://www.npmjs.com/package/react-native-svg-charts?activeTab=readme.

[9] npm, «react-native-keyboard-aware-scrollview,» [En línea]. Available: https://www.npmjs.com/package/react-native-keyboard-aware-scrollview.

[10] npm, «@expo/vectors-icons,» [En línea]. Available: https://www.npmjs.com/package/@expo/vector-icons.

[11] npm, «radio-buttons-react-native,» [En línea]. Available: https://www.npmjs.com/package/radio-buttons-react-native.

[12] npm, «react-native-svg,» [En línea]. Available: https://www.npmjs.com/package/react-native-svg.

[13] npm, «react-native-table-component,» [En línea]. Available: https://www.npmjs.com/package/react-native-table-component.

[14] ThingsBoard, «ThingsBoard REST API,» [En línea]. Available: http://207.180.221.115:8080/swagger-ui.html#/.

[15] npm, «apisauce,» [En línea]. Available: https://www.npmjs.com/package/apisauce/v/1.0.2.

[16] «MDN Web Docs,» [En línea]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Date.