



TRABAJO FIN DE GRADO  
INGENIERÍA EN INFORMÁTICA

# Implementación de un Dashboard configurable para IoT

---

AppIoT

**Autor**

Francisco Javier Merchán Martín (alumno)

**Director**

José Luis Garrido Bullejos (tutor)  
Francisco Manuel García Moreno (tutor)



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

—  
Granada, a 16 de Noviembre de 2022

# AppIot

---

Implementación de un Dashboard configurable para IoT

## **Autor**

Francisco Javier Merchán Martín (alumno)

## **Directores**

José Luis Garrido Bullejos (tutor)  
Francisco Manuel García Moreno (tutor)

—  
Granada, a 16 de Noviembre de 2022

# Implementación de un Dashboard configurable para IoT

Francisco Javier Merchán Martín (alumno)

**Palabras clave:** IoT, dashboard, dispositivos, software libre,

## Resumen

Este proyecto tiene como objetivo realizar una propuesta de una plataforma IoT (Internet of Things) para la recopilación, el procesamiento, la visualización de datos y la gestión de dispositivos, que puede ser utilizada por usuarios de forma simple y sencilla, y con capacidad de ser escalable y extenderse en un futuro.

Para la realización de este proyecto se han utilizado una serie de tecnologías emergentes y cada vez más populares, las cuales no han explotado aún todo su potencial.

La principal motivación es contribuir, mediante dicha plataforma a facilitar el uso de software bajo el nuevo paradigma IoT y poder visualizar/consultar/-procesar/controlar la información asociada a los dispositivos (cosas) que forman parte de dichas aplicaciones mediante un tablón (dashboard).

La plataforma está basada en una arquitectura dirigida por eventos con todo lo que significa e implica en cuanto a poder dotar a las aplicaciones de ciertas propiedades en términos de escalabilidad, la independencia de las partes que componen la arquitectura, portabilidad a otros sistemas y extensión con nuevas funcionalidades.

# Implementation of a configurable Dashboard for IoT

Francisco Javier Merchan Martin (student)

**Keywords:** IoT, dashboard, devices, OpenSource,

## Abstract

This project aims to make a proposal for an IoT (Internet of Things) platform for the collection, processing, visualisation of data and device management, which can be used by users in a simple and easy way, and with the capacity to be scalable and extendable in the future.

A number of emerging and increasingly popular technologies, which have not yet exploited their full potential, have been used in the realisation of this project.

The main motivation is to contribute, through this platform, to facilitate the use of software under the new IoT paradigm and to be able to visualise/consult/process/control the information associated with the devices (things) that form part of these applications through a dashboard.

The platform is based on an event-driven architecture with all that it means and implies in terms of being able to provide applications with certain properties in terms of scalability, the independence of the parts that make up the architecture, portability to other systems and extension with new functionalities.

---

Yo, **Francisco Javier Merchán Martín**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 31032280J, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

A handwritten signature in black ink, enclosed within a hand-drawn oval. The signature appears to read 'Fco Javier'.

Fdo: Francisco Javier Merchán Martín

Granada, a 16 de Noviembre de 2022.

---

D. **José Luis Garrido Bullejos (tutor)** y D. **Francisco Manuel García Moreno (tutor)**, Profesores del Área de Lenguajes y Sistemas Informáticos del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

**Informan:**

Que el presente trabajo, titulado , ha sido realizado bajo su supervisión por **Francisco Javier Merchán Martín(alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada, a 16 de Noviembre de 2022 .

**El Director:**

**El Director:**

**José Luis Garrido Bullejos**

**Francisco Manuel García Moreno**

”Un Anillo para gobernarlos a todos. Un Anillo para encontrarlos, un Anillo  
para atraerlos a todos y atarlos en las tinieblas”  
Sauron en *El señor de los anillos*

# Agradecimientos

*Este trabajo esta dedicado a todas esa personas que confiaron en mí durante tanto tiempo y es gracias a ellas porque hoy estoy aquí.*

*A mi grupo de amigos y compañeros de universidad, en especial a Vero, Paula, Bella, Marcin y Vicky.*

*A mi familia por haberme acompañado y apoyado en todas las etapas.*

*Por ultimo quiero dar las gracias a mis abuelos por haberme enseñado a perseverar y continuar para adelante con todo.*



# Índice general

<b>1. Introducción</b>	<b>16</b>
1.1. Dominio del problema y motivación . . . . .	16
1.2. Objetivos . . . . .	17
1.3. Metodología de desarrollo . . . . .	18
1.4. Presupuesto . . . . .	20
1.4.1. Recursos hardware . . . . .	20
1.4.2. Recursos software . . . . .	20
1.4.3. Recursos humanos . . . . .	21
1.4.4. Presupuesto final . . . . .	21
1.5. Estructura de la memoria . . . . .	22
<b>2. Estado del arte</b>	<b>23</b>
2.1. Definiciones . . . . .	23
2.2. Plataformas . . . . .	26
2.2.1. thingsboard.io . . . . .	26
2.2.2. DataDog . . . . .	27
2.2.3. Kaaiot . . . . .	28
2.2.4. Cayenne myDevices . . . . .	29
2.3. Conclusión . . . . .	30
<b>3. Herramientas y Tecnologías</b>	<b>32</b>
3.1. Contenedores . . . . .	32
3.2. MQTT . . . . .	33
3.3. API . . . . .	34
3.4. BBDDS y NoSQL . . . . .	35
3.4.1. Bases de Datos NoSQL . . . . .	35
3.5. Frameworks de desarrollo de software . . . . .	35
3.6. Vue y Nuxt . . . . .	36
3.6.1. Nuxt.js . . . . .	36
3.7. Docker . . . . .	37
3.8. Mongo . . . . .	37
3.9. Node . . . . .	38
3.10. EMQX . . . . .	39
3.11. Postman . . . . .	39

<b>4. Análisis y diseño de propuesta</b>	<b>41</b>
4.1. Requisitos	41
4.1.1. Requisitos funcionales	41
4.1.2. Requisitos no funcionales	45
4.1.3. Restricciones Semánticas	46
4.2. Casos de Uso	50
4.2.1. Tablas de Casos de Uso	50
4.2.2. Esquemas de casos de uso	60
4.3. Diseño arquitectónico	64
4.4. Modelo de datos	68
4.5. Front-end	72
4.5.1. Widgets	75
4.5.1.1. Botón	75
4.5.1.2. Switch	76
4.5.1.3. Indicador	76
4.5.1.4. Gráfica numérica	76
4.5.2. Páginas	77
4.5.2.1. Marco	77
4.5.2.2. Dashboard	82
4.5.2.3. Dispositivos	82
4.5.2.4. Plantillas	83
4.5.2.5. Alarmas	85
4.5.2.6. Login	86
4.5.2.7. Registro	87
4.6. Estructura del Back-end	87
4.6.1. Mongo	87
4.6.1.1. Conexión	88
4.6.1.2. Modelo	89
4.6.2. Broker EMQX	91
4.6.2.1. Motor de Reglas	94
4.6.2.2. Autenticación	98
4.6.2.3. Cluster	101
4.6.2.4. Topics	103
4.6.3. JWT	104
4.6.4. API	105
4.6.4.1. Usuarios	108
4.6.4.2. Plantillas	110
4.6.4.3. Dispositivos	111
4.6.4.4. Reglas de Guardado	115
4.6.4.5. Tipos de Dispositivos	115
4.6.4.6. Alarmas	118
4.6.4.7. Notificaciones	120
4.6.4.8. Web Hooks	121

<b>5. Sprints</b>	<b>123</b>
5.1. Diagrama de Grant . . . . .	123
5.2. Sprints . . . . .	124
5.2.1. Sprint 0 . . . . .	124
5.2.2. Sprint 1 . . . . .	124
5.2.2.1. Objetivos . . . . .	124
5.2.2.2. Tareas realizadas . . . . .	124
5.2.2.3. PMV v.1 . . . . .	125
5.2.2.4. Pruebas . . . . .	125
5.2.2.5. Comentarios . . . . .	126
5.2.2.6. Capturas de pantalla . . . . .	127
5.2.3. Sprint 2 . . . . .	129
5.2.3.1. Objetivos . . . . .	129
5.2.3.2. Tareas realizadas . . . . .	129
5.2.3.3. PMV v.2 . . . . .	129
5.2.3.4. Pruebas . . . . .	130
5.2.3.5. Comentarios . . . . .	133
5.2.3.6. Capturas de pantalla . . . . .	134
5.2.4. Sprint 3 . . . . .	137
5.2.4.1. Objetivos . . . . .	137
5.2.4.2. Tareas realizadas . . . . .	137
5.2.4.3. PMV v.3 . . . . .	137
5.2.4.4. Pruebas . . . . .	137
5.2.4.5. Comentarios . . . . .	137
5.2.4.6. Capturas de pantalla . . . . .	138
5.3. Despliegue . . . . .	140
<b>6. Conclusiones y trabajo futuro</b>	<b>142</b>

# Índice de figuras

2.1. Diferentes visiones de IoT . . . . .	24
2.2. The internet of things: a survey [1] . . . . .	25
2.3. Productos ofrecidos por ThingsBoard.io[2] . . . . .	26
2.4. Datadog[3] . . . . .	27
2.5. KaaIoT[4] . . . . .	28
2.6. Cayenne myDevices[5] . . . . .	30
3.1. Vue.js [6] . . . . .	36
3.2. Nuxt [7] . . . . .	36
3.3. Docker [8] . . . . .	37
3.4. MongoDB [9] . . . . .	38
3.5. Node.js [10] . . . . .	38
3.6. EMQX[11] . . . . .	39
3.7. Postman[12] . . . . .	40
4.1. Esquema Casos de uso Usuarios . . . . .	61
4.2. Esquema Casos de uso Dispositivos . . . . .	62
4.3. Esquema Casos de uso Alarmas y Notificaciones . . . . .	63
4.4. Esquema Casos de uso Plantillas . . . . .	64
4.5. Diseño general de la propuesta . . . . .	65
4.6. Diseño específico . . . . .	65
4.7. Diagrama con un un ejemplo de paso de mensaje [13] . . . . .	66
4.8. Esquema de relación de los datos . . . . .	69
4.9. Diagrama de las base de datos . . . . .	71
4.10. Widget Botón . . . . .	75
4.11. Widget Switch . . . . .	76
4.12. Widget Indicador . . . . .	76
4.13. Widget Gráfica . . . . .	76
4.14. Página Dispositivos señalando la parte superior . . . . .	77
4.15. Pagina Dispositivos señalando la parte central . . . . .	78
4.16. Pagina Dispositivos señalando la parte inferior . . . . .	78
4.17. Sección de Dashboard . . . . .	82
4.18. Sección de Dispositivos . . . . .	83
4.19. Sección de Plantillas . . . . .	84
4.20. Sección de Plantilla con Widgets . . . . .	84

4.21. Sección de Plantilla con Previsualización y tabla . . . . .	85
4.22. Sección de Alarmas . . . . .	85
4.23. Sección de Alarmas con tabla . . . . .	86
4.24. Login . . . . .	86
4.25. Registro . . . . .	87
4.26. Pantalla principal del dashboard de EMQX . . . . .	92
4.27. Pestaña de Clientes conectados . . . . .	92
4.28. Pestaña de módulos activos . . . . .	93
4.29. Pestaña de EMQX API HTML . . . . .	93
4.30. Pestaña del cliente Websocket . . . . .	94
4.31. Pestaña Resources EMQX . . . . .	95
4.32. Ventana emergente de creación de un recurso . . . . .	95
4.33. Pestaña de Reglas EMQX . . . . .	97
4.34. Pestaña de Plugins EMQX . . . . .	99
4.35. Datos de usuarios en la colección emqxauthrules en MongoDB . . . . .	100
4.36. Esquema de un cluster EMQX [14] . . . . .	102
5.1. Diagrama de Gantt . . . . .	123
5.2. Login . . . . .	127
5.3. Registro . . . . .	127
5.4. Página de Dispositivos . . . . .	128
5.5. Página de Plantillas con la creación de 1 widget . . . . .	128
5.6. Página de Plantillas . . . . .	134
5.7. Página de Alarmas . . . . .	134
5.8. Página de Dispositivos . . . . .	135
5.9. Página de Dashboard . . . . .	135
5.10. Página de Dashboard con Notificaciones abiertas . . . . .	136
5.11. Página de Dashboard con el botón de logout . . . . .	136
5.12. Página de Dispositivos . . . . .	138
5.13. Página de Plantillas . . . . .	139
5.14. Página de Alarmas . . . . .	139
5.15. Página de Dashboard con un ejemplo . . . . .	140

# Índice de tablas

1.1. Presupuesto recursos hardware . . . . .	20
1.2. Presupuesto recursos software . . . . .	20
1.3. Presupuesto recursos humanos . . . . .	21
1.4. Presupuesto final . . . . .	21
2.1. Características thingsboard.io . . . . .	27
2.2. Características Datadog . . . . .	28
2.3. Características Kaaiot . . . . .	29
2.4. Características Cayenne[15] . . . . .	30
4.1. Requisito F - 1 . . . . .	41
4.2. Requisito F - 2 . . . . .	41
4.3. Requisito F - 3 . . . . .	42
4.4. Requisito F - 4 . . . . .	42
4.5. Requisito F - 5 . . . . .	42
4.6. Requisito F - 6 . . . . .	42
4.7. Requisito F - 7 . . . . .	42
4.8. Requisito F - 8 . . . . .	43
4.9. Requisito F - 9 . . . . .	43
4.10. Requisito F - 10 . . . . .	43
4.11. Requisito F - 11 . . . . .	43
4.12. Requisito F - 12 . . . . .	43
4.13. Requisito F - 13 . . . . .	44
4.14. Requisito F - 14 . . . . .	44
4.15. Requisito F - 15 . . . . .	44
4.16. Requisito F - 16 . . . . .	44
4.17. Requisito F - 17 . . . . .	44
4.18. Requisito F - 18 . . . . .	45
4.19. Requisito F - 19 . . . . .	45
4.20. Requisito F - 20 . . . . .	45
4.21. Requisito F - 21 . . . . .	45
4.22. Requisito NF - 1 . . . . .	45
4.23. Requisito NF - 2 . . . . .	46
4.24. Requisito NF - 3 . . . . .	46
4.25. Requisito NF - 4 . . . . .	46

4.26. Restricción Semántica - 1 . . . . .	46
4.27. Restricción Semántica - 2 . . . . .	46
4.28. Restricción Semántica - 3 . . . . .	47
4.29. Restricción Semántica - 4 . . . . .	47
4.30. Restricción Semántica - 5 . . . . .	47
4.31. Restricción Semántica - 6 . . . . .	47
4.32. Restricción Semántica - 7 . . . . .	47
4.33. Restricción Semántica - 8 . . . . .	48
4.34. Restricción Semántica - 9 . . . . .	48
4.35. Restricción Semántica - 10 . . . . .	48
4.36. Restricción Semántica - 11 . . . . .	48
4.37. Restricción Semántica - 12 . . . . .	48
4.38. Restricción Semántica - 13 . . . . .	49
4.39. Restricción Semántica - 14 . . . . .	49
4.40. Restricción Semántica - 15 . . . . .	49
4.41. Restricción Semántica - 16 . . . . .	49
4.42. Restricción Semántica - 17 . . . . .	49
4.43. Restricción Semántica - 18 . . . . .	50
4.44. Caso de uso CU-01 . . . . .	50
4.45. Caso de uso CU-02 . . . . .	51
4.46. Caso de uso CU-03 . . . . .	51
4.47. Caso de uso CU-04 . . . . .	52
4.48. Caso de uso CU-05 . . . . .	52
4.49. Caso de uso CU-06 . . . . .	53
4.50. Caso de uso CU-07 . . . . .	53
4.51. Caso de uso CU-08 . . . . .	54
4.52. Caso de uso CU-09 . . . . .	54
4.53. Caso de uso CU-10 . . . . .	55
4.54. Caso de uso CU-11 . . . . .	55
4.55. Caso de uso CU-12 . . . . .	56
4.56. Caso de uso CU-13 . . . . .	56
4.57. Caso de uso CU-14 . . . . .	57
4.58. Caso de uso CU-15 . . . . .	57
4.59. Caso de uso CU-16 . . . . .	58
4.60. Caso de uso CU-17 . . . . .	58
4.61. Caso de uso CU-18 . . . . .	59
4.62. Caso de uso CU-19 . . . . .	59
4.63. Caso de uso CU-20 . . . . .	60
4.64. Caso de uso CU-21 . . . . .	60

# Índice de Códigos Fuente

4.1. Main panel de un template . . . . .	74
4.2. Ejemplo Store de Nuxt . . . . .	74
4.3. Middleware de autenticación . . . . .	75
4.4. Parte html del archivo default.js de la carpeta layouts . . . . .	79
4.5. Extracto de la parte script del archivo default.js de la carpeta layouts . . . . .	80
4.6. Ejemplo de conexión con mongoose . . . . .	88
4.7. Modelo de datos de Plantilla . . . . .	89
4.8. Import del modelo de datos Plantilla . . . . .	90
4.9. Ejemplo de uso del modelo con la función <i>create</i> . . . . .	91
4.10. Ejemplo de uso del modelo con la función <i>find</i> . . . . .	91
4.11. Ejemplo de Objeto Auth . . . . .	96
4.12. Ejemplo de datos Recurso . . . . .	96
4.13. Ejemplo de creación de una regla . . . . .	97
4.14. Ejemplo de datos de un usuario para MQTT . . . . .	101
4.15. Requiere del middleware . . . . .	104
4.16. Ejemplo de utilizacion del middleware de autenticación . . . . .	104
4.17. Ejemplo de crear una instancia de Express . . . . .	106
4.18. Configuración Express . . . . .	106
4.19. Exportación de la ruta del archivo plantillas.js . . . . .	106
4.20. Exportación del modulo . . . . .	107
4.21. Ejemplo de escucha por el puerto 3001 . . . . .	107
4.22. Requiere para el uso en archivos de rutas . . . . .	107
4.23. Ejemplo de un end-point . . . . .	107
4.24. Objeto JSON enviado por el Body . . . . .	108
4.25. Respuesta correcta . . . . .	108
4.26. Objeto JSON enviado por el Body . . . . .	108
4.27. Ejemplo de Respuesta . . . . .	109
4.28. Ejemplo de Respuesta . . . . .	109
4.29. Ejemplo de Respuesta . . . . .	109
4.30. Objeto JSON enviado por el Body . . . . .	110
4.31. Respuesta correcta . . . . .	110
4.32. Ejemplo de Respuesta . . . . .	111
4.33. Ejemplo de Respuesta . . . . .	111
4.34. Objeto JSON enviado por el Body . . . . .	112



4.35. Respuesta correcta . . . . .	112
4.36. Ejemplo de Respuesta . . . . .	112
4.37. Ejemplo de Respuesta . . . . .	114
4.38. Objeto JSON enviado por el Body . . . . .	114
4.39. Respuesta correcta . . . . .	114
4.40. Objeto JSON enviado por el Body . . . . .	115
4.41. Respuesta correcta . . . . .	115
4.42. Objeto JSON enviado por el Body . . . . .	115
4.43. Respuesta correcta . . . . .	116
4.44. Ejemplo de Respuesta . . . . .	116
4.45. Ejemplo de Respuesta . . . . .	118
4.46. Objeto JSON enviado por el Body . . . . .	118
4.47. Respuesta correcta . . . . .	118
4.48. Objeto JSON enviado por el Body . . . . .	119
4.49. Respuesta correcta . . . . .	119
4.50. Respuesta correcta . . . . .	119
4.51. Ejemplo de Respuesta . . . . .	120
4.52. Objeto JSON enviado por el Body . . . . .	120
4.53. Respuesta correcta . . . . .	121
4.54. Objeto JSON enviado por el Body . . . . .	121
4.55. Objeto JSON enviado por el Body . . . . .	122

# Capítulo 1

## Introducción

En este capítulo se detalla el dominio del problema que se aborda, cuál es la motivación que ha llevado a realizar este proyecto, los objetivos del mismo, metodología de desarrollo y presupuesto.

### 1.1. Dominio del problema y motivación

El mundo vive cada vez un intenso cambio en el cual cada vez se crean más y más dispositivos electrónicos para dar una solución a un problema o simplemente para hacernos la vida más fácil.

En particular, cada vez son más los dispositivos que llevamos encima, relojes inteligentes, móviles, llaveros, etc. Todos ellos recogen datos y están conectados y se comunican entre sí. Estos dispositivos al estar conectados entre sí crean una red muy rica en recursos y cada vez más explotada por empresas para poder usar estos datos.

Ahí entra el Internet de las Cosas o como se le conoce comúnmente por su término en inglés IoT o Internet of Things. Este término engloba a todos esos pequeños aparatos que están conectados y muchos de ellos nos hacen la vida más fácil, muchas veces sin que seamos conscientes. Por ejemplo, las persianas eléctricas, bombillas inteligentes o un simple termostato. Todos ellos nos ofrecen funciones o datos que nos son útiles.

De ahí nace la idea de poder tener todos esos dispositivos juntos en un dashboard o tablón con los que poder interactuar de forma visual, fácil y sencilla. Desde un tablón personalizable se puede consultar la temperatura de una casa, encender la bombilla de baño, enviar una notificación cuando un sensor alcance cierto valor simplemente monitorizar el pulso de tu perro.

La principal motivación es contribuir, mediante una plataforma, a facilitar

el uso de software bajo el nuevo paradigma IoT y poder visualizar/consultar/-procesar/controlar la información asociada a los dispositivos (cosas) que forman parte de dichas aplicaciones mediante un tablón. El uso de la plataforma se ilustra gracias al desarrollo llevado a cabo de un prototipo que intenta mostrar las posibilidades que ofrece y que pueden incluirse como trabajo futuro.

## 1.2. Objetivos

El objetivo principal de este proyecto es realizar una propuesta de una plataforma de IoT para la recopilación, el procesamiento, la visualización de datos así como la gestión de dispositivos, que sea fácil de utilizar para usuarios y con capacidad de ser escalable y extenderse en un futuro. La plataforma debe permitir visualizar la información de diferentes tipos de sensores y datos mediante un tablón o dashboard. Para lograr dicho objetivo general se plantean los siguientes objetivos específicos:

- Recopilar, procesar y explotar los datos de los distintos sensores que formen parte de la aplicación IoT. Por ejemplo, almacenar y consultar la temperatura y, con ello, bajar o subir la calefacción, o conocer si una persona tiene el pulso acelerado.
- Ofrecer la capacidad de crear una personalización y organización amplia del dashboard, donde cada usuario podrá elegir qué sensores quiere visualizar, qué tipo de gráficas se usarán para ello y los distintos controles de los dispositivos que estarán disponibles.
- Prestar atención a la experiencia de usuario para que la herramienta resulte sencilla de usar y no sea costosa de mantener, así se contribuirá que los usuarios muestren más interés en el uso de la herramienta.

Al final del presente proyecto se espera obtener un prototipo de la plataforma deseada, a partir de un diseño que permita poder ampliarla en un futuro.

Adicionalmente, como objetivos formativos específicos se plantean:

- El estudio de IoT como tecnología y su futuro potencial en distintas aplicaciones.
- El aprendizaje de las distintas tecnologías relacionadas con el IoT y con el desarrollo de plataformas.
- Poner en práctica los conocimientos adquiridos en relación con la gestión de un proyecto de estas características a desarrollar en un periodo de tiempo.
- El afrontar el reto de desarrollar una solución como la que se deriva de este proyecto.

### 1.3. Metodología de desarrollo

Para este proyecto se ha decido utilizar la metodología ágil SCRUM[16]. Esta metodología de desarrollo de software pertenece al conjunto de metodología ágiles que se ha vuelto muy popular en los últimos años gracias a su tolerancia a los fallos y al aceleramiento en el desarrollo.

Estas metodologías ágiles entre las que se encuentra SCRUM, también se encuentra Kaban, diseñada en Toyota y aunque no solo es específica para el desarrollo de software está ganando cada vez más adeptos. Todas ellas siguen el manifiesto ágil el cual tiene 4 principios[17]:

1. Individuos e interacciones sobre procesos y herramientas
2. Software funcionando sobre documentación extensiva
3. Colaboración con el cliente sobre negociación contractual
4. Respuesta ante el cambio sobre seguir un plan

SCRUM además de seguir los principios del manifiesto ágil tiene otras particularidades, como la definición de Sprint, el cual es el pilar de esta metodología. En pocas palabras un Sprint es un periodo de tiempo en cual se desarrolla SCRUM y tiene una duración de entre 2 y 4 semanas. Un proyecto tiene varios Sprints.

Empieza con una reunión entre todas las personas involucradas en el proyecto, entre ellas el SCRUM Master (vela por que se siga la metodología y que todo esté a tiempo), el cliente y el equipo de desarrollo. Y que termina con una reunión de objetivos cumplidos y no cumplidos y con un prototipo funcional o PMV (Producto Mínimamente Viable) de la aplicación a desarrollar.

En esa reunión inicial al principio de cada Sprint se determinan los Jobs o tareas a realizar en ese Sprint. En este caso los desarrolladores se comprometen a llevarlo a cabo en ese tiempo. Además de esa reunión se tienen una todos los días una reunión interna para comentar los avances y problemas surgidos en los Jobs. Así como una reunión semanal para el mismo propósito. Esto hace que el SCRUM Master pueda ir dirigiendo y coordinando a todas las partes para así poder llevar a buen puerto el Sprint.

Es una metodología cercana con el cliente ya que está involucrado en el proyecto y puede ver cómo va construyéndose cada parte a partir de prototipos y ayudando así si no se tienen unos objetivos/requisitos desde el principio poder ir adaptándose a los cambios.

Ya que el proyecto a desarrollar es de carácter cambiante y los requisitos no estaban del todo definidos desde el principio, así como cercanía con el cliente, en

este caso los tutores, se ha decidido seguir este método. Con alguna peculiaridad, ya que este método está pensado para un grupo de programadores y no una sola persona he decidido añadir algunas adiciones. Por ejemplo, el periodo de Sprint no se suele cambiar a lo largo del proyecto, este caso porque ello ha ayudado de forma específica a avanzar más rápido e ir adaptándose a las distintas partes del proyecto; también debido a que el trabajo paralelo de varias personas no se realizará y no hay que coordinar las distintas partes. En un principio los Sprint tendrán una duración de tres semanas. Pero nunca alejándose de la base de SCRUM que son los Sprint, el seguimiento del manifiesto ágil, así como el seguimiento a través de reuniones y autorreflexiones.

Otro punto importante de SCRUM es asegurar una calidad mínima del producto cuando se entrega cada MVP (Minimum Viable Product). Estos MVP serán nombrados con el término release en mi caso y con cada release se impondrá un grupo de pruebas que se realizará para que el prototipo pueda asegurar su calidad. La gran mayoría de ellas están apuntadas junto a las características de cada versión entregada al cliente. Estas pruebas se basarán en programación de sentencias condicionales para comprobar datos y la integración de try-catch que recojan los errores para poder tratarlos correctamente consiguiendo una aplicación altamente tolerante a fallos. Junto con estas dos características se realizan comprobaciones de uso en forma de beta testing en la que personalmente probaré el uso de la aplicación como un usuario objetivo.

Para poder tener más visibles los trabajos se ha dividido el proyecto en varios módulos:

- Front end
- Back end
- Docker
- API y servidor
- Conexión con dispositivos

En paralelo se ha ido montando esta documentación de forma continua a lo largo de todo el proyecto, así como pequeños errores y correcciones que se han ido solucionando conforme han ido pasando las etapas.

Para finalizar, con respecto al despliegue se ha incluido una sección en el capítulo de Sprints para ello, además e ha subido a la plataforma GitHub para que sea utilizado por todo el que lo necesita ya que se ha desarrollado como OpenSource. Y se encuentra en el siguiente [link](#) .

## 1.4. Presupuesto

### 1.4.1. Recursos hardware

Para el proyecto no se ha necesitado adquirir ningún dispositivo nuevo, pero hay que tener en cuenta el gasto previo realizado de los dispositivos utilizados para su realización. Naturalmente, para la amortización, se hace una estimación de la vida útil de los dispositivos utilizados, la cual va a ser de 5 años y para calcular el coste de amortización se va a usar la siguiente fórmula:

$$\text{Coste de amortización} = \text{Precio} * \frac{\text{Meses de uso en el proyecto}}{\text{Vida útil}} \quad (1.1)$$

Recurso	Precio	Amortización
Ordenador de mesa (Intel 10700K, 16 GB de RAM)	1240 €	124 €
Teclado, ratón	60,00 €	6 €
Pantallas	260,00 €	26,00 €
Total	1560,00 €	

Tabla 1.1: Presupuesto recursos hardware

### 1.4.2. Recursos software

En este proyecto, todos los recursos utilizados son gratuitos, gratuitos para estudiantes, o licencia Apache, aunque existen versiones de pago de estos recursos, pero para el desarrollo no ha sido necesario recurrir a las versiones de pago.

Recurso	Precio
Microsoft Windows 10 Education	0,00 €
EMQX	0,00 €
Node.js	0,00 €
MongoDB	0,00 €
Postman	0,00 €
Visual Studio Code	0,00 €
Plantilla TIM Creative	0,00 €
Microsoft Office	0,00 €(Gratis para estudiantes)
Google Drive	0,00 €
Total	0,00 €

Tabla 1.2: Presupuesto recursos software

### 1.4.3. Recursos humanos

Para calcular el coste de recursos humanos se ha tenido en cuenta el precio de un Ingeniero Junior recién graduado, estando establecido en 14 €/h [18] aproximadamente y para el supervisor de 25 €/h [19]. Teniendo en cuenta ese coste se ha calculado el coste total del presupuesto dedicado a recursos humanos.

<b>Recurso</b>	<b>Horas</b>	<b>Coste/Hora</b>	<b>Coste total</b>
Desarrollador Junior	780 h	14,00 €/h	10920,00 €
Desarrollado Senior (Supervisor)	50 h	25,00 €/h	1250,00 €
		Total	12170,00 €

Tabla 1.3: Presupuesto recursos humanos

### 1.4.4. Presupuesto final

Una vez obtenidos los costes referentes a los recursos materiales, de software y humanos, se ha calculado el presupuesto final, al cual se le ha añadido el IVA del 21 %.

<b>Recurso</b>	<b>Precio</b>
Recursos materiales	1560,00 €
Recursos de software	0,00 €
Recursos humanos	12170,00 €
Total	13730,00 €
Total (IVA incluido)	16613.3 €

Tabla 1.4: Presupuesto final

## 1.5. Estructura de la memoria

La presente memoria se ha estructurado en los capítulos que se enumeran a continuación:

**Capítulo 1: Introducción.** El capítulo introduce el dominio del problema que se aborda, cuál es la motivación que ha llevado a realizar este proyecto y los objetivos del mismo.

**Capítulo 2: Estado del arte.** En este capítulo se describe en detalle el paradigma IoT de forma abstracta y conceptual, así como las distintas soluciones existentes en el mercado relacionadas con el presente trabajo.

**Capítulo 3: Herramientas y Tecnologías.** Se ha dedicado este capítulo a estudiar y revisar las tecnologías y herramientas a ser usadas en el desarrollo del proyecto, es decir, las más aptas para la solución propuesta.

**Capítulo 4: Análisis y diseño de propuesta.** En este capítulo se describe el desarrollo de los distintos aspectos de la propuesta que se han llevado a cabo, así como una serie de guías para los posibles usuarios desarrolladores.

**Capítulo 5: Sprints.** Aquí se detallan los Sprints realizados en el proyecto, así como el diagrama de Gran resultante

**Capítulo 6: Conclusiones y trabajo futuro.** Se resumen las principales conclusiones y se incluye una valoración personal del proyecto, así como una serie de posibles ampliaciones o mejoras para la plataforma.



## Capítulo 2

# Estado del arte

En este capítulo se describe en detalle el paradigma IoT de forma abstracta y conceptual, así como las distintas soluciones existentes en el mercado relacionadas con el presente trabajo.

### 2.1. Definiciones

Aunque en el anterior punto hemos definido IoT, a continuación, se describe con más detalle. IoT es un término confuso es hablar de un paradigma con muchas visiones dentro de ese mismo. Es un término muy confuso ya que es una mezcla de muchas áreas que componen estas tecnologías. A alguien ajeno al mundillo relacionado con IoT le puede resultar difícil comprender de qué se trata realmente. Pero esa también es su gran virtud ya que le otorga muchísima versatilidad.

En la Figura 1 se muestran las diferentes visiones de este término, la convergencia de las 3 visiones es lo que da lugar a la definición de IoT. La definición de la Comisión Europea dice que son esas entidades inteligentes virtuales que operan espacios inteligentes a través de interfaces inteligentes para comunicarse y conectarse dentro de contextos sociales, ambientales y de usuarios[20], lo que recuerda mucho a la definición de Smart things, que son esos dispositivos “inteligentes” que nos ayudan en nuestra vida diaria.

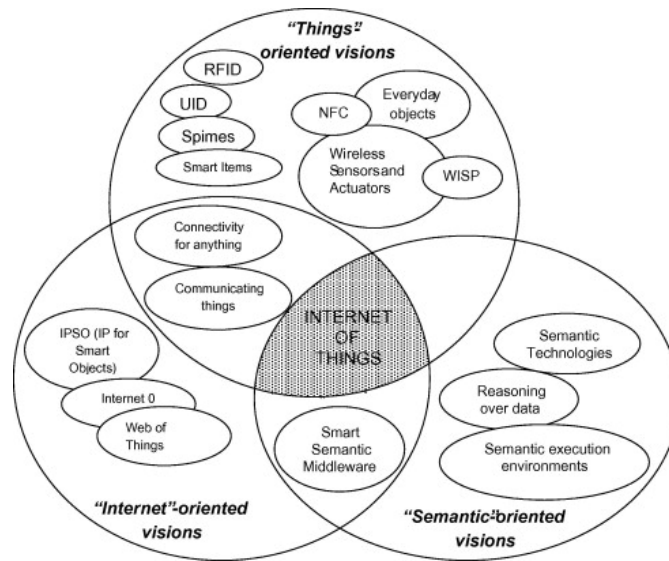


Figura 2.1: Diferentes visiones de IoT

Al final podemos decir que IoT[21] trata de una red de dispositivos con sensores que se conectan y comunican entre ellos a través de distintos protocolos o interfaces sin intervención humana.

Esta idea no es nueva y se ido intentado dar forma de distintas maneras, como por ejemplo Kevin Ashton propuso un Concepto de IoT en 1999 y que se relaciona intrínsecamente con la tecnología RFID o *Radio Frequency Identification*. Esto es solo una visión inicial la cual se fue desarrollando poco a poco de forma exponencial hasta lo que conocemos hoy, que guarda estrechamente relación con el mundo de las TIC y las Smart things, por lo menos con el concepto comercial de este último término.

Antes de la explosión comercial del llamado Internet de las cosas, el IoT se desarrolló a alrededor de empresas, estas utilizaban una red de sensores junto con otros dispositivos conectados entre sí para por ejemplo vigilar la producción y si hay algún problema hacer saltar las alarmas. Al final todos estos dispositivos formaban una red que al estar conectados entre sí envían y reciben datos. La mayoría de estas redes son cerradas, por lo que el verdadero auge fue cuando estas redes se pudieron conectar a Internet.

Como cada red era independiente, cada fabricante o empresa tenía sus propios protocolos de conexión entre ellos, así como sus propios estándares, haciendo esto que su corto recorrido comercial se viera entorpecido. Abajo podemos ver una imagen con unos pocos de los muchos protocolos, tecnologías y estándares basados o compatibles con este paradigma.

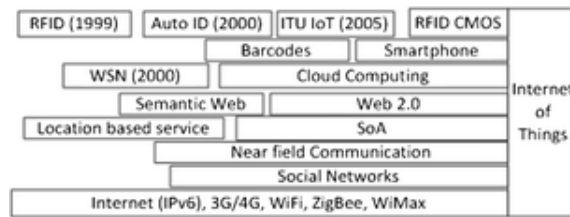


Figura 2.2: The internet of things: a survey [1]

Esta tendencia ha ido a la baja, aunque al principio muchas de las empresas crearon sus propios protocolos y formas de conexión de dispositivos, cada vez son más las que trabajan en formas para poder operar bajo los mismos estándares.

En la actualidad es mucho lo que se espera de este paradigma que hace «inteligentes» los aparatos de la vida mundana y como tal tendrá un gran futuro y eso significa aunar todos bajo un mismo paraguas, de ahí nació el protocolo MQTT[22], que es un protocolo desarrollado por IBM pero que es abierto y totalmente gratuito y que cada vez lo implementan más empresas como forma de conectar sus dispositivos. Esto hace que en un futuro no muy lejano tengamos esas «Smart Homes» que se anuncian desde hace años, en las cuales los dispositivos conectados entre sí puedan darte información y a su vez se comuniquen entre ellos y ayudarte a hacerte la vida más fácil.

Aun con unas posibilidades infinitas y los muchos productos del mercado, solo unos pocos se han atrevido a adentrarse en el mundo de los dashboard a nivel de usuario que te dé esa oportunidad de ver la información que recolecta esa red de sensores e interactuar con ella. La mayoría de estos productos, están pensados para su uso empresarial y no para usuarios. A continuación, vamos a ver cuáles son algunas de las plataformas IoT disponibles.

Pero antes cabe mencionar que este desarrollo ha ido también estrechamente relacionado con lo que comúnmente conocemos como Big Data. Esta red de sensores que definimos antes ha servido a las empresas para recabar datos, para luego usarlos para el desarrollo de productos o para entrenar Inteligencias Artificiales. Al principio no se daba importancia, pero en la actualidad es común tener conocimiento que esos datos son importantes a la vez que valiosos, por lo que el buen uso de estos se ha hecho cada vez más importante.

Muchas empresas han puesto a disposición de los usuarios estos datos, pero estos usuarios no son conscientes de estos o no son capaces de utilizarlos o de tratarlos. Muchas soluciones son complicadas de manejar o no son viables para un usuario promedio.

Al final la solución para el usuario para manejar estos datos es lo que lla-

mos Dashboard, que no es más que un tablón con el que poder interactuar o que nos permita mostrar estos datos. Estos Dashboards pueden incluir otros componentes o subsistemas más allá de una simple interfaz gráfica, por ejemplo, un back-end conectado con una base de datos. Podemos decir que es una plataforma, a la cual conectamos los dispositivos y que esta recibe los datos de estos y los procesa de cierta manera.

## 2.2. Plataformas

Con la anterior concepción de plataforma con Dashboard o tablón, a continuación se describen las siguientes plataformas en el mercado.

### 2.2.1. thingsboard.io

Esta plataforma pertenece a ThingsBoard Inc[23] fundada en 2016 en Estados Unidos. Ofrece distintos productos (Figura 2.3), pero casi todos para profesionales o grandes empresas.

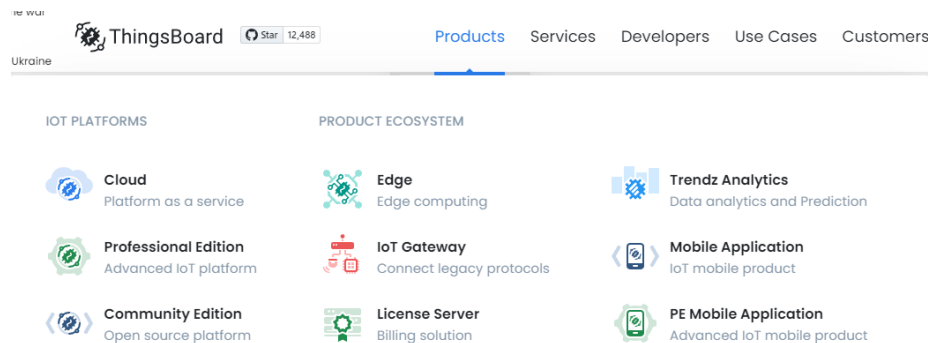


Figura 2.3: Productos ofrecidos por ThingsBoard.io[2]

Tienen una versión OpenSource en la puedes hacer muchas cosas, pero tienes que encargarte de programar todo, así como desplegarlo. Son compatibles con distintos tipos de dispositivos populares como son Arduino, ESP32, etc. Su solución más completa incluye el desarrollo y el lanzamiento en la nube de la empresa.

Sus mejores bazas son la cantidad de dispositivos y de productos que ofrece como una aplicación para móviles, así como un sistema de reglas y alarmas y la personalización.

Son compatibles con los protocolos HTTP, CoAP y MQTT.

Características	
<b>Vers. OpenSource</b>	Sí
<b>Precio Licencia</b>	10 \$/month a 749 \$/month
<b>Servicio en la Nube</b>	Sí, de pago
<b>Compatible con distintos protocolos</b>	Sí (MQTT, HTTP, CoAP, LwM2M)
<b>Documentación Clara</b>	Sí. solo en inglés
<b>Aplicaciones externas</b>	Sí, de pago para móviles
<b>Soporte</b>	Soporte de la comunidad, otras formas de soportes de pago
<b>Fácil de instalar</b>	Moderado
<b>Fácil de manejar</b>	Moderado
<b>Sistema de Reglas</b>	Sí
<b>Distintas formas de visualización</b>	Sí
<b>API</b>	Sí

Tabla 2.1: Características thingsboard.io

### 2.2.2. DataDog

DataDog [3] es una empresa estadounidense especializada en monitoreo de datos y de seguridad en aplicaciones cloud. Por lo tanto, tiene muchísimas opciones de productos, desde soluciones machine learning, manejo de datos sensibles, etc. Proporciona un “servicio de monitorización para aplicaciones en la nube, que proporciona monitorización de servidores, bases de datos, herramientas y servicios, a través de una plataforma de análisis de datos basada en SaaS (Software como Servicio)[24].

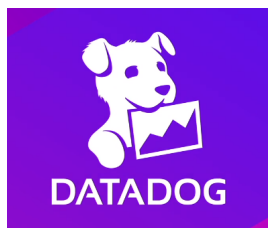


Figura 2.4: Datadog[3]

La solución que ofrece es totalmente personalizada con un gran equipo detrás, pero pensada para empresas, con dashboard personalizados, etc.

Al ser una solución tan personalizada no dan muchos detalles más allá del énfasis en la personalización del producto y en que puedes monitorear millones de datos.

Características	
<b>Vers. OpenSource</b>	No
<b>Precio Licencia</b>	Por módulos de áreas (Ej, infraestructura, Databse Monitoring, etc)
<b>Servicio en la Nube</b>	No especificado
<b>Compatible con distintos protocolos</b>	No especificado
<b>Documentación Clara</b>	Sí, en varios idiomas y muy escueta
<b>Aplicaciones externas</b>	Sí, de pago para móviles
<b>Soporte</b>	Soporte de pago
<b>Fácil de instalar</b>	No especificado por falta de información
<b>Fácil de manejar</b>	Moderado
<b>Sistema de Reglas</b>	No especificado
<b>Distintas formas de visualización</b>	Sí
<b>API</b>	Sí

Tabla 2.2: Características Datadog

### 2.2.3. Kaaiot

Al igual que las dos anteriores es una empresa estadounidense, que ofrece una gran personalización en las plantillas. Además, como la primera reseñada tiene su propia nube e incluye una prueba gratuita.



Figura 2.5: Kaaiot[4]

Su principal público objetivo vuelve a ser las empresas o autónomos, tiene varios planes entre ellos varios en su nube, en la cual cobra dependiendo del número de dispositivos, o para montarlo en tu propio servidor.

Características	
<b>Vers. OpenSource</b>	No
<b>Precio Licencia</b>	Por dispositivos conectados, desde 499 \$/month. La no vinculada al cloud no disponible el precio.
<b>Servicio en la Nube</b>	Sí
<b>Compatible con distintos protocolos</b>	Sí 1/KP, MQTT y próximamente con CoAP
<b>Documentación Clara</b>	Sí. aunque demasiado escueta y más organizada
<b>Aplicaciones externas</b>	No
<b>Soporte</b>	Soporte de pago
<b>Fácil de instalar</b>	No especificado por falta de información
<b>Fácil de manejar</b>	No especificado por falta de información
<b>Sistema de Reglas</b>	Sí
<b>Distintas formas de visualización</b>	Sí
<b>API</b>	Sí

Tabla 2.3: Características Kaaiot

#### 2.2.4. Cayenne myDevices

Plataforma de origen estadounidense perteneciente a la empresa myDevices[15], cabe mencionar a favor de esta plataforma que es muy visual y tiene varias funciones que se habían pensado para este proyecto que se propone, además permite desarrollar código en términos personalización de widgets y contenidos de los mensajes en el lado del front-end. También nos ofrece una aplicación para smartphones con el controlar los dispositivos conectados.

Pero ocurre con este tipo de plataformas y es que la adición de nuevos dispositivos pasa por tener que programar los propios dispositivos, adaptándonos a los que nos ofrecen.

Por decir otro, pero es la poca información de la plataforma de los planes que ofrecen, entre las limitaciones tenemos que cada cliente solo envía 60 mensajes por minuto y por IP, por decir alguna.

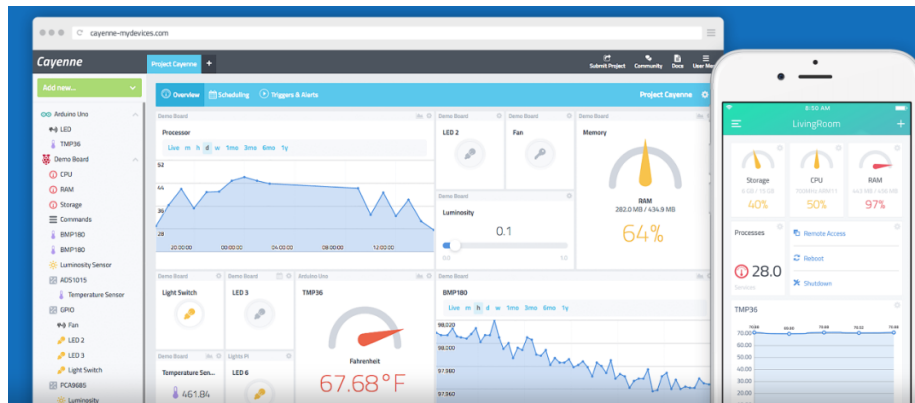


Figura 2.6: Cayenne myDevices[5]

Por otra parte, ofrece un gran Marketplace con multitud de soluciones terminadas que al parecer es a lo que está dirigido su modelo de negocio. Además de estar dirigido principalmente para empresas o autónomos.

Características	
<b>Vers. OpenSource</b>	No especificado
<b>Precio Licencia</b>	No especificado
<b>Servicio en la Nube</b>	Sí, de la empresa matriz
<b>Compatible con distintos protocolos</b>	Sí, LoRa, y MQTT
<b>Documentación Clara</b>	No
<b>Aplicaciones externas</b>	Sí
<b>Soporte</b>	Soporte de la comunidad
<b>Fácil de instalar</b>	Moderado debido a la falta de información oficial
<b>Fácil de manejar</b>	Moderado
<b>Sistema de Reglas</b>	Sí
<b>Distintas formas de visualización</b>	Sí
<b>API</b>	Sí

Tabla 2.4: Características Cayenne[15]

## 2.3. Conclusión

Después de esta búsqueda y estudio de propuestas que van en la línea de la nuestra, podemos concluir que la mayoría de las soluciones son solo viables para usos comerciales, solo 1 de las opciones incluía versión para la comunidad y varias de ellas ni siquiera disponían de información sobre la licencia. Podemos



decir que es posible contribuir proponiendo plataformas que no se dediquen al uso empresarial y que sean accesibles para el usuario medio.

La mayoría de ellas requieren un gran esfuerzo no solo por parte del usuario experto para montarlo si no para el usuario final para el mantenimiento, además del desembolso grande que hay que realizar para la instalación y compra de licencias.

Para encontrarnos con algo parecido a lo que tenemos en mente hay que considerar aplicaciones exclusivas para dispositivos móviles, como es Mi Home, que agrega a la mayoría de sus dispositivos de manera fácil y sencilla y que ofrece muchas funcionalidades.

Si queremos verlo como un todo, la que más se le acerca es la solución de IoTthings.io. Al final el objetivo es crear una plataforma que sea fácil de usar para el usuario medio y adoptar principios y decisiones de diseño que permitan su extensión mediante el desarrollo futuro. Así más adelante será posible incluir más dispositivos, no solo compatibles con MQTT, que por el tiempo que dura un proyecto de este tipo ha sido imposible.

Así pues, podemos considerar que este proyecto busca proponer una plataforma para la recolección, procesamiento y visualización de datos, así como la gestión y control de dispositivos, facilitando el uso de dicha plataforma y extensión.

## Capítulo 3

# Herramientas y Tecnologías

Este capítulo está dedicado a estudiar y revisar las tecnologías y herramientas a ser usadas en el desarrollo del proyecto, es decir, las que pueden resultar más aptas para la solución propuesta.

Para el presente proyecto se han valorado varias tecnologías y herramientas para resolver distintos tipos de problemas. Las soluciones más óptimas con el contexto presente como para el proyecto son las que veremos más abajo.

Como se ha pensado en una plataforma modular, en la que se pudiese cambiar aspectos de ella sin mucho esfuerzo o con el mínimo número de cambios. Estas tecnologías salvo algunas claves se pueden cambiar, por ejemplo, la utilización de Docker como motor de contenedores o la utilización de un broker para MQTT respecto a otro.

### 3.1. Contenedores

Los contenedores por hacer un símil con un contenedor físico en la vida real es un recipiente hermético y que su contenido es invisible en ambas direcciones tanto para dentro como para fuera. Podemos decir que los contenedores[25] por software son realmente es un conjunto de software metido en una caja y que se comunica a través del motor de aplicaciones (por ejemplo, Docker) con el sistema operativo. Esto hace posible que la estandarización de los procesos como ocurrió con los contenedores de verdad, facilitando[26] así el transporte de programas software de un sitio a otro sin importar el entorno donde esté. Con el tiempo MQTT se ha ido actualizando y añadiendo más características al protocolo como son la caducidad de los mensajes, alias de tema, etc. La versión Actual es la 5.0.

A diferencia de las máquinas virtuales[27] que emulan tanto el software como el hardware, los contenedores se emulan sobre el sistema operativo, es decir, que

comparten el mismo núcleo que sistema anfitrión, con todo lo que eso conlleva como por ejemplo reducir la carga sobre el sistema. Es útil verlo como la arquitectura por microservicios, es decir, cada contenedor solo debería contener lo necesario para ejecutar el software objetivo.

Algunos ejemplos de motores de contenedores son:

- Docker
- Kubernetes

## 3.2. MQTT

MQTT (MQ Telemetry Transport)[28] es un protocolo extremadamente liviano (lightweight) de mensajería para el IoT o Internet of things y se basa en una la idea de publicación y suscripción. En el cual los dispositivos se suscriben a un topic o tema que deseen obtener información. Fue diseñado en su primera versión por Andy Stanford-Clark (IBM) y Arlen Nipper(Eurotech, Inc) y fue privado hasta que en 2013, entonces IBM, la propietaria, paso los derechos a OASIS(Organization for the Advancement of Structured Information Standards). OASIS es un consorcio que vela por la estandarización, el desarrollo y la convergencia de varias áreas tecnológicas como el IoT[29] y lo convirtió en público. MQTT al basarse sobre el protocolo TCP/IP puede usarse sobre cualquier red, por lo que cualquier dispositivo compatible con este protocolo de comunicación puede albergar un cliente MQTT.

Como hemos dicho antes se basa en una arquitectura de suscripción. Hay dos tipos de dispositivos en MQTT, los clientes y los brokers. Los primeros envían o reciben información de los segundos, que actúan como intermediarios. Los clientes nunca se contactan entre ellos.

Se usa normalmente para redes no fiables o con recursos limitados ya que los paquetes de información no son muy grandes. En esta línea, MQTT tiene tres tipos de QoS o Quality of Services para velar por la información en los distintos escenarios de uso:

- QoS 0 → En este tipo de calidad de servicios, los mensajes se entregan una sola vez y no se guardan para el cliente que se ha desconectado ya que se asume que todos los dispositivos lo han recibido correctamente.
- QoS 1 → Para este tipo de calidad, se pide que el cliente le mande un recibo de que ha recibido la información, si no el broker volverá a mandar la información. Podemos decir que se manda al menos una vez y los mensajes son guardados a la espera de los clientes que están desconectados.
- QoS 2 → Por último, en este caso, se hace una confirmación en cuatro pasos, para que el mensaje llegue solo una vez. También se guardan los mensajes para los dispositivos desconectados.

La seguridad no fue una prioridad cuando se creó este protocolo por lo que se puede considerar inseguro, y esto provocó que se hiciera un pequeño intento de solventar esa falta de seguridad, creando una conexión con usuario y contraseña, pero este texto también va sin cifrar ya que se impuso el que fuera liviano y rápido. Como se ejecuta sobre el protocolo TCP/IP podemos añadirle SSL/TLS para crear conexiones más seguras, pero pagando el precio de que las conexiones no sean tan livianas. Por lo que queda a cargo de los implementadores la adición de mecanismos adicionales de seguridad. En nuestro caso, hemos configurado el broker se puede configurar para tener restricciones a los tópicos que se puede suscribir y publicar el usuario, y como parte de la plataforma se ha incluido también el cambio de credenciales cada vez que se conecte el usuario.

Por último, decir que la elección del protocolo MQTT como el protocolo principal, es principiante por su gran versatilidad y compatibilidad con muchísimos dispositivos. Trabajar con él resulta muy cómodo por que es muy potente pero a la vez es muy fácil de comprender y de usar, con hay mucha documentación disponible. Además, es de lo pocos protocolo Open Source de MQTT y de los que no esta asociado a ninguna tecnología o empresa que tenga que restrinja los dispositivos compatibles, como Zigbee que necesita que los dispositivos tengan un modulo certificado por ellos, por lo que restringe mucho la cantidad de dispositivos compatibles. Otras opciones como CoAP[30], no esta estandarizadas por completo por lo que puede ser un problema a largo plazo ya que no tiene una versión estable y estandarizada como MQTT.

### 3.3. API

API[31] son las siglas de Application Programming Interface, que en español significa interfaz de programación de aplicaciones. Las API sirven para comunicar dos productos software a través de un conjunto de protocolos y definiciones. No es la parte más visible de los servicios, pero es una parte fundamental de muchos de ellos ya que permite obtener información estructurada de forma fácil y estandarizada.

Una API[32] permiten ofrecer un servicio o una funcionalidad sin tener que proporcionar nada más que su acceso, ofreciendo así privacidad, ya que no se sabe cómo funciona por dentro. Esto también ayuda a la hora de actualizar una API, ya que mientras no cambiamos la entrada o la salida podemos hacer las actualizaciones que queramos totalmente transparentes a fuera.

Hay muchísimos tipos de APIS[33], pueden ser públicas o privadas a un grupo de personas o incluso locales.

### 3.4. BBDDS y NoSQL

Según Wikipedia, una base de datos[34] en términos generales es un conjunto de datos estructurado que pertenece a un mismo contexto. Un buen ejemplo puede ser una biblioteca, la cual es una base de datos compuesta por documentos y textos, indexados para poder consultarlos.

Para poder manejar esta información hay programas preparados para ellos, y se definen como sistemas gestores de bases de datos[35] o SGBD, los cuales permiten acceder y almacenar los datos de una forma estructurada y rápida.

Hay multitud de tipos de bases de datos, desde multidimensionales, transnacionales, documentales o deductivas.

Las más utilizadas actualmente son las bases de datos relacionales basadas en SQL, aunque poco a poco las NoSQL van ganando cuota.

#### 3.4.1. Bases de Datos NoSQL

Las bases de datos NoSQL[36] o *Not only SQL*, son bases de datos que aparecen para solución a los problemas de estabilidad para gestionar grandes cantidades de información de las bases de datos relaciones que utilizaban las grandes empresas.

Bases de datos NoSQL suelen tener menos consistencia en los datos ya que son bases de datos especialmente pensadas para la lectura y escritura rápida de gran cantidad de datos.

Aunque no existe gran cantidad de aplicaciones o de soporte, cada vez son más las empresas que piensan e implantan este tipo de bases de datos. Algunos gestores de base de datos NoSQL son Mongo o Cassandra.

### 3.5. Frameworks de desarrollo de software

Cómo define Arimetic, un framework es un marco o esquema de trabajo generalmente utilizado por programadores para realizar el desarrollo de software[37]. Gracias a esto se pueden realizar los procesos ya que permite repetir reutilizar código asegurando la consistencia del código y la buena praxis.

También facilita el trabajo colaborativo ya define estándares, así como mayor seguridad ya que muchos frameworks tiene una gran comunidad o una gran empresa que los prueba y mantiene.

Ejemplos de frameworks[38] son .NET, Nuxt.js , Django, etc.

Ahora que hemos visto las tecnologías vamos a ver las herramientas o soluciones que hemos usado en el proyecto.

### 3.6. Vue y Nuxt

Vue[39] es un framework de JavaScript, es open source, es decir que es libre y gratuito. Su principal utilidad es la de construir interfaces de usuario de forma más sencilla.



Figura 3.1: Vue.js [6]

Fue creado por un extrabajador de Google[40] Evan You en 2014. Él además fue quien desarrolló el actualmente famoso framework Angular. Aunque en principio no fue pensado más que para ser un proyecto personal y pequeño, la comunidad realizó un arduo trabajo para hacer que creciera y convertirlo en el framework que conocemos hoy en día.

Vue se basa en el uso de componentes, los cuales encapsulan código y extiende elementos básicos de HTML. Este framework se centra en SPA o aplicaciones de una sola página, que hace que al principio una web cargue un poco más lento, pero después vaya mucho más fluida mientras navegas con ella.

Otro punto fuerte de Vue es que es reactivo, es decir, reacciona al cambio de eventos que van sucediendo con la interacción del usuario.

#### 3.6.1. Nuxt.js

Nuxt.js[41] es un framework de un framework, ya que se basa en Vue.js. Al igual que Vue, Nuxt es open source y tiene una gran comunidad detrás que lo ha hecho crecer hasta donde está ahora mismo. Nuxt[42] resuelve algunas configuraciones tediosas de Vue haciéndolas más sencillas como el enrutamiento. Es un framework totalmente modular, así que podemos ir añadiendo a nuestro proyecto solo las partes que nos interesen e ir cambiando entre Vue y Nuxt.



Figura 3.2: Nuxt [7]

Nuxt[43] podemos configurarlo para página SPA, SSR o estáticas. Tiene una estructura de carpetas ya definida lo que hace que el código esté mucho más ordenado

### 3.7. Docker

Antes hemos hablado de los contenedores Software. Pues Docker[44] es un proyecto que los gestiona. Utiliza distintas características de Linux para poder hacer uso del kernel para poder aislar procesos y ejecutarlos en entornos separados. En Windows es posible gracias al WSL que utiliza el núcleo de Linux de forma nativa en Windows.



Figura 3.3: Docker [8]

Docker[8] permite crear, parar y ejecutar de manera rápida y sencilla contenedores en cualquier entorno.

Docker[45] es una buena opción si tienes pocos contenedores y no necesitas relacionarlos entre ellos ya que no tienen ninguna herramienta para administrarlos de manera automática y fácil. Por ello se creó Docker-compose, con esto podemos definir y poner en funcionamiento múltiples contenedores a la vez en una sola máquina. Aunque si necesitas mayor escalabilidad y poner en funcionamiento varias máquinas Kubernetes es la mejor opción para ello.

Actualmente Docker[46] es acogido por los principales proveedores de servicios en la nube por lo que es una manera fácil y sencilla de desplegar nuestras aplicaciones. Aparte de que es muy conocido, también muy polivalente y sirve tanto en el ámbito empresarial como en el de desarrollo y no requiere de grandes requisitos, además de que la curva de aprendizaje es pequeña ya que se sabe usar.

### 3.8. Mongo

Es un SGBD para base de datos NoSQL[9] creado en 2009 por 10gn Inc. Está amparado por GNU por lo que se considera Software libre. Está escrito en C++, aunque hay multitud de componentes y de drivers para multitud de lenguajes.



Figura 3.4: MongoDB [9]

La base de datos que usa es una base de datos documental, por lo que a diferencia de las bases de datos relaciones los datos se guardan en colecciones. Estas colecciones guardan datos en forma de documentos que están en formato BSON. No es necesario seguir esquemas ya que los documentos de una misma colección pueden tener esquemas diferentes.

MongoDB[47] es muy bueno escalando horizontalmente ya que puede dividir las colecciones para poder repartirlas por distintos servidores.

Actualmente Mongo[9] es acogido por los principales proveedores de servicios en la nube. Además, es un SGBD muy liviano, es open source, rico en documentación y muchos clientes realizados por la comunidad para poder conectarse desde diferentes lenguajes de programación. Otra opción es DocumentDB de Amazon que es de pago o Couchbase Server sin tanta respuesta por la comunidad además de pago.

### 3.9. Node

Node[48] es un entorno de ejecución multiplataforma para con el foco en la capa del servidor basada en el lenguaje JavaScript y orientado a eventos asíncronos. Fue lanzada inicialmente en 2009, es de código abierto bajo licencia MIT.



Figura 3.5: Node.js [10]

Node.js[49] está basado en el motor V8 de Google (para compilar JavaScript a código máquina) y enfocado principalmente para la crear programas de red



fácilmente escalables. No se ejecuta en el navegador si no es el servidor y desde su página web se puede descargar el entorno.

Node.js[50] dispone de multitud de módulos que se pueden descargar e instalar a través del instalar npm que viene con el entorno. Hay muchos módulos o como por ejemplo para conectarse a mongo o Express para desarrollar APIS.

Esto lo hace idóneo para hacer la ejecución de distritos procesos en la parte del servidor o para ejecutar framework como por ejemplo Nuxt.js además de ser altamente escalable.

### 3.10. EMQX

Es un broker de MQTT[51] desarrollado por EMQ Teams en 2013. Es open source y licenciado bajo Apache 2.0. Está desarrollado en el lenguaje de programación Erlang de Ericsson.



Figura 3.6: EMQX[11]

Empezó soportando la versión 3.0 del protocolo, pero actualmente soporta la última versión de este la 5.0. Es fácilmente escalable tanto vertical como horizontalmente y su configuración[52] es muy sencilla. Se puede decir que es el más escalable de los brokers estudiados ya que tiene una configuración tipo cluster, la cual alberga nodos que se comunican entre sí.

Otra cosa que diferencia a EMQX frente a todos:

EMQX[53] es multiplataforma y tiene extensiones para casi todos los lenguajes de programación y posee soporte en múltiples servicios cloud.

### 3.11. Postman

Postman[54] es un software que se usa para probar y documentar APIs, también tiene un gran repositorio de APIS para poder consultar y probar. Se eligió este programa para hacer la documentación y el testeo de la API por su sencillez y lo rápido que se aprende, también por su interfaz amigable.



Figura 3.7: Postman[12]

Otros productos usados para este fin son Apache Jmeter o SoapUI, este último desarrollado por la empresa Smartbear, inicialmente pensado para el protocolo SOAP, poco después se abrió a RESTful. Ambas tienen su parte de pago, pero en este caso con las características que ofrece la versión gratuita es suficiente.

En Postman[55] he podido agrupar las peticiones en colecciones y dentro de estas en subcarpetas para poder ordenarlas mejor. También ayuda mucho cuando se tiene que desarrollar en equipo ya que incluye muchísimas funciones para ello. Lo único malo que para exportar la documentación es un poco difícil ya que la sube públicamente.

## Capítulo 4

# Análisis y diseño de propuesta

En este capítulo se describe el desarrollo de los distintos aspectos de la propuesta que se han llevado a cabo.

### 4.1. Requisitos

Estos son los requisitos que debería tener la plataforma que se tiene la idea de desarrollar.

#### 4.1.1. Requisitos funcionales

<b>RF</b>	1
<b>NOMBRE</b>	Registrarse
<b>DESCRIPCIÓN</b>	El usuario que quiera usar el sistema podrá registrarse.

Tabla 4.1: Requisito F - 1

<b>RF</b>	2
<b>NOMBRE</b>	Iniciar sesión
<b>DESCRIPCIÓN</b>	El usuario que ya este registrado en el sistema podrá iniciar sesión.

Tabla 4.2: Requisito F - 2

<b>RF</b>	3
<b>NOMBRE</b>	Cerrar sesión
<b>DESCRIPCIÓN</b>	El usuario con una sesión iniciada puede acabar desconectarse del sistema.

Tabla 4.3: Requisito F - 3

<b>RF</b>	4
<b>NOMBRE</b>	Crear Dispositivos
<b>DESCRIPCIÓN</b>	El usuario podrá registrar un dispositivo.

Tabla 4.4: Requisito F - 4

<b>RF</b>	5
<b>NOMBRE</b>	Borrar Dispositivos
<b>DESCRIPCIÓN</b>	El usuario podrá borrar un dispositivo creado por él.

Tabla 4.5: Requisito F - 5

<b>RF</b>	6
<b>NOMBRE</b>	Crear Alarma
<b>DESCRIPCIÓN</b>	El usuario podrá crear una Alarma para un dispositivo que él haya registrado.

Tabla 4.6: Requisito F - 6

<b>RF</b>	7
<b>NOMBRE</b>	Borrar Alarma
<b>DESCRIPCIÓN</b>	El usuario podrá borrar una Alarma para un dispositivo que él haya registrado.

Tabla 4.7: Requisito F - 7

<b>RF</b>	8
<b>NOMBRE</b>	Encender Alarma
<b>DESCRIPCIÓN</b>	El usuario podrá encender una Alarma que él haya creado.

Tabla 4.8: Requisito F - 8

<b>RF</b>	9
<b>NOMBRE</b>	Apagar Alarma
<b>DESCRIPCIÓN</b>	El usuario podrá apagar una Alarma que él haya creado.

Tabla 4.9: Requisito F - 9

<b>RF</b>	10
<b>NOMBRE</b>	Enviar Notificación
<b>DESCRIPCIÓN</b>	El sistema enviará una notificación al usuario cuando una regla de una Alarma definida por él se rompa.

Tabla 4.10: Requisito F - 10

<b>RF</b>	11
<b>NOMBRE</b>	Crear Plantilla
<b>DESCRIPCIÓN</b>	El usuario podrá crear una plantilla añadiendo los widgets que necesite.

Tabla 4.11: Requisito F - 11

<b>RF</b>	12
<b>NOMBRE</b>	Borrar Plantilla
<b>DESCRIPCIÓN</b>	El usuario podrá borrar una plantilla que él haya creado.

Tabla 4.12: Requisito F - 12

<b>RF</b>	13
<b>NOMBRE</b>	Añadir Widget
<b>DESCRIPCIÓN</b>	El usuario podrá añadir un widget de una plantilla.

Tabla 4.13: Requisito F - 13

<b>RF</b>	14
<b>NOMBRE</b>	Borra Widget
<b>DESCRIPCIÓN</b>	El usuario podrá borrar un widget de una plantilla

Tabla 4.14: Requisito F - 14

<b>RF</b>	15
<b>NOMBRE</b>	Guardar datos recibidos
<b>DESCRIPCIÓN</b>	El usuario podrá decidir si los datos recibidos de un dispositivo se guardan o no.

Tabla 4.15: Requisito F - 15

<b>RF</b>	16
<b>NOMBRE</b>	Procesar datos recibidos
<b>DESCRIPCIÓN</b>	El sistema procesará los datos de los dispositivos para poder ser usados por el usuario o por el propio sistema.

Tabla 4.16: Requisito F - 16

<b>RF</b>	17
<b>NOMBRE</b>	Enviar Notificación al usuario
<b>DESCRIPCIÓN</b>	El usuario recibirá una notificación del sistema cada vez que una alarma encendida sea avisada.

Tabla 4.17: Requisito F - 17

<b>RF</b>	18
<b>NOMBRE</b>	Visualizar datos de forma gráfica
<b>DESCRIPCIÓN</b>	El sistema podrá mostrar al usuario los datos de los dispositivos de forma gráfica.

Tabla 4.18: Requisito F - 18

<b>RF</b>	19
<b>NOMBRE</b>	Enviar datos al dispositivo
<b>DESCRIPCIÓN</b>	El usuario podrá enviar datos al dispositivo.

Tabla 4.19: Requisito F - 19

<b>RF</b>	20
<b>NOMBRE</b>	Cambiar de plantilla
<b>DESCRIPCIÓN</b>	El usuario podrá cambiar de plantilla entre las creadas por él.

Tabla 4.20: Requisito F - 20

<b>RF</b>	21
<b>NOMBRE</b>	Mostrar Plantilla
<b>DESCRIPCIÓN</b>	El sistema mostrará una plantilla entre las creadas por el usuario que este escoja.

Tabla 4.21: Requisito F - 21

#### 4.1.2. Requisitos no funcionales

<b>RNF</b>	1
<b>NOMBRE</b>	Confidencialidad
<b>DESCRIPCIÓN</b>	Se protegerá la información del sistema de accesos no autorizados.

Tabla 4.22: Requisito NF - 1

<b>RNF</b>	2
<b>NOMBRE</b>	Fiabilidad
<b>DESCRIPCIÓN</b>	Tolerancia a fallos del sistema.

Tabla 4.23: Requisito NF - 2

<b>RNF</b>	3
<b>NOMBRE</b>	Escalabilidad
<b>DESCRIPCIÓN</b>	El sistema tiene que facilitar la construcción de soluciones escalables.

Tabla 4.24: Requisito NF - 3

<b>RNF</b>	4
<b>NOMBRE</b>	Adaptativo
<b>DESCRIPCIÓN</b>	El sistema tiene que ser capaz de adaptarse a nuevas tecnologías.

Tabla 4.25: Requisito NF - 4

#### 4.1.3. Restricciones Semánticas

<b>RS</b>	1
<b>DESCRIPCIÓN</b>	Cualquier persona puede registrarse, los datos obligatorios son email y contraseña.
<b>RF</b>	RF-1

Tabla 4.26: Restricción Semántica - 1

<b>RS</b>	2
<b>DESCRIPCIÓN</b>	La contraseña será cifrada.
<b>RF</b>	RF-1

Tabla 4.27: Restricción Semántica - 2



<b>RS</b>	3
<b>DESCRIPCIÓN</b>	Para poder registrarse el email no puede estar en uso por una cuenta ya existente.
<b>RF</b>	RF-1

Tabla 4.28: Restricción Semántica - 3

<b>RS</b>	4
<b>DESCRIPCIÓN</b>	Para poder iniciar sesión, se necesita una cuenta registrada en el sistema.
<b>RF</b>	RF-2

Tabla 4.29: Restricción Semántica - 4

<b>RS</b>	5
<b>DESCRIPCIÓN</b>	Para poder cerrar sesión tiene que haber iniciado sesión antes.
<b>RF</b>	RF-3

Tabla 4.30: Restricción Semántica - 5

<b>RS</b>	6
<b>DESCRIPCIÓN</b>	Un usuario puede registrar un o más dispositivos.
<b>RF</b>	RF-4

Tabla 4.31: Restricción Semántica - 6

<b>RS</b>	7
<b>DESCRIPCIÓN</b>	Un usuario solo tendrá acceso a los dispositivos creados por el mismo.
<b>RF</b>	RF-4, RF-5

Tabla 4.32: Restricción Semántica - 7

<b>RS</b>	8
<b>DESCRIPCIÓN</b>	Un usuario puede crear una o más alarmas, pero solo para variables de dispositivos asignados a su cuenta.
<b>RF</b>	RF-6

Tabla 4.33: Restricción Semántica - 8

<b>RS</b>	9
<b>DESCRIPCIÓN</b>	Un usuario solo tendrá acceso a las alarmas creadas por el mismo.
<b>RF</b>	RF-7, RF-8, RF-9

Tabla 4.34: Restricción Semántica - 9

<b>RS</b>	10
<b>DESCRIPCIÓN</b>	Un usuario solo podrá apagar una alarma que haya sido creada por él mismo y que no esté apagada ya.
<b>RF</b>	RF-9

Tabla 4.35: Restricción Semántica - 10

<b>RS</b>	11
<b>DESCRIPCIÓN</b>	Un usuario solo podrá encender una alarma que haya sido creada por él mismo y que no esté encendida ya.
<b>RF</b>	RF-8

Tabla 4.36: Restricción Semántica - 11

<b>RS</b>	12
<b>DESCRIPCIÓN</b>	Un usuario podrá tener una o más plantillas.
<b>RF</b>	RF-11

Tabla 4.37: Restricción Semántica - 12

<b>RS</b>	13
<b>DESCRIPCIÓN</b>	Un usuario solo tendrá acceso a las plantillas creadas por él.
<b>RF</b>	RF-11, RF-12, RF-20

Tabla 4.38: Restricción Semántica - 13

<b>RS</b>	14
<b>DESCRIPCIÓN</b>	Un usuario podrá añadir uno o más widgets a una plantilla.
<b>RF</b>	RF-13

Tabla 4.39: Restricción Semántica - 14

<b>RS</b>	15
<b>DESCRIPCIÓN</b>	Solo el usuario con acceso a la plantilla podrá borrar un widget.
<b>RF</b>	RF-14

Tabla 4.40: Restricción Semántica - 15

<b>RS</b>	16
<b>DESCRIPCIÓN</b>	Solo el usuario que creo la alarma recibirá la notificación de que esta se ha roto.
<b>RF</b>	RF-10, RF-17

Tabla 4.41: Restricción Semántica - 16

<b>RS</b>	17
<b>DESCRIPCIÓN</b>	El dispositivo podrá recibir y enviar datos solo por los tópicos acordados, así como en el formato establecido.
<b>RF</b>	RF-16, RF-18

Tabla 4.42: Restricción Semántica - 17

<b>RS</b>	18
<b>DESCRIPCIÓN</b>	Solo el usuario al cual pertenece el dispositivo podrá decidir si los datos recibidos se guardan o no.
<b>RF</b>	RF-15

Tabla 4.43: Restricción Semántica - 18

## 4.2. Casos de Uso

### 4.2.1. Tablas de Casos de Uso

<b>CASOS DE USO</b>	Registrarse	<b>ID</b>	CU-01
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-1		
<b>PRECONDICIONES</b>	El usuario no debe tener una cuenta registrada.		
<b>POSTCONDICIONES</b>	Los datos de usuario han sido registrados en el sistema.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Registrar a un usuario en el sistema.		
<b>RESUMEN</b>	El usuario introduce sus datos, el sistema los valida y los registra en la base de datos correspondiente.		

Tabla 4.44: Caso de uso CU-01

<b>CASOS DE USO</b>	Iniciar sesión	<b>ID</b>	CU-02
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-2		
<b>PRECONDICIONES</b>	El usuario ha de estar registrado.		
<b>POSTCONDICIONES</b>	El usuario es capaz de acceder a las funciones del sistema.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
	Iniciar sesión de un usuario en el sistema.		
<b>RESUMEN</b>			
	El sistema validará si el usuario ha introducido los datos correctos cotejándolos en la base de datos, si es así les dará acceso a las diferentes características del sistema.		

Tabla 4.45: Caso de uso CU-02

<b>CASOS DE USO</b>	Cerrar sesión	<b>ID</b>	CU-03
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-3		
<b>PRECONDICIONES</b>	El usuario ha de estar registrado. El usuario debe haber iniciado sesión.		
<b>POSTCONDICIONES</b>	El usuario se ha desconectado de la aplicación.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
	Cerrar sesión de un usuario en el sistema.		
<b>RESUMEN</b>			
	El sistema cerrará la conexión del usuario que lo ha solicitado.		

Tabla 4.46: Caso de uso CU-03

<b>CASOS DE USO</b>	Crear Dispositivo	<b>ID</b>	CU-04
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-4		
<b>PRECONDICIONES</b>	El usuario ha de estar registrado. El usuario debe haber iniciado sesión.		
<b>POSTCONDICIONES</b>	Creación de un dispositivo elegido por el usuario.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Añadir un dispositivo a la colección del usuario.		
<b>RESUMEN</b>	El usuario meterá los datos del dispositivo seleccionado, el sistema los validará y verificará que no pertenece a otro usuario y se añadirá el mencionado dispositivo al repositorio del usuario.		

Tabla 4.47: Caso de uso CU-04

<b>CASOS DE USO</b>	Borrar Dispositivo	<b>ID</b>	CU-05
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-5		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. El dispositivo tiene que existir. No puede tener ninguna plantilla relacionada.		
<b>POSTCONDICIONES</b>	El dispositivo ya no existe en el sistema.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Eliminar el dispositivo deseado del usuario.		
<b>RESUMEN</b>	El usuario escogerá en una lista el dispositivo que dese eliminar y al darle click sobre él o algún botón para ese uso el sistema borrará el dispositivo de ese usuario.		

Tabla 4.48: Caso de uso CU-05

<b>CASOS DE USO</b>	Borrar Dispositivo	<b>ID</b>	CU-06
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-6		
<b>PRECONDICIONES</b>	El usuario tiene que haber iniciado sesión. El dispositivo tiene que pertenecer al usuario.		
<b>POSTCONDICIONES</b>	El usuario tendrá asociada una Alarma.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Crear Alarma para una variable de un dispositivo del usuario.		
<b>RESUMEN</b>	El sistema creará una Alarma sobre la variable del dispositivo del usuario con las condiciones que el usuario haya impuesto.		

Tabla 4.49: Caso de uso CU-06

<b>CASOS DE USO</b>	Borrar Alarma	<b>ID</b>	CU-07
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-7		
<b>PRECONDICIONES</b>	El usuario tiene que haber iniciado sesión. El dispositivo tiene que pertenecer al usuario. La alarma tiene que existir.		
<b>POSTCONDICIONES</b>	La alarma ya no estará asociada al usuario.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Borrar una Alarma creada por el usuario.		
<b>RESUMEN</b>	El sistema borrará la Alarma creada por el usuario.		

Tabla 4.50: Caso de uso CU-07

<b>CASOS DE USO</b>	Borrar Alarma	<b>ID</b>	CU-08
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-8		
<b>PRECONDICIONES</b>	El usuario tiene que haber iniciado sesión. El dispositivo tiene que pertenecer al usuario. La alarma tiene que existir. La alarma tiene que estar apagada.		
<b>POSTCONDICIONES</b>	La alarma estará encendida.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
	Encender la Alarma de un usuario		
<b>RESUMEN</b>			
	El sistema encenderá la Alarma creada por el usuario.		

Tabla 4.51: Caso de uso CU-08

<b>CASOS DE USO</b>	Apagar Alarma	<b>ID</b>	CU-09
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-9		
<b>PRECONDICIONES</b>	El usuario tiene que haber iniciado sesión. El dispositivo tiene que pertenecer al usuario. La alarma tiene que existir. La alarma tiene que estar encendida.		
<b>POSTCONDICIONES</b>	La alarma estará apagada.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
	Apagar la Alarma de un usuario		
<b>RESUMEN</b>			
	El sistema apagará la Alarma creada por el usuario y no recibirá notificaciones cuando se incumpla la condición.		

Tabla 4.52: Caso de uso CU-09



<b>CASOS DE USO</b>	Enviar Notificación	<b>ID</b>	CU-10
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-10		
<b>PRECONDICIONES</b>	El usuario tiene que haber iniciado sesión. El dispositivo tiene que pertenecer al usuario. La alarma tiene que existir. La alarma tiene que estar encendida.		
<b>POSTCONDICIONES</b>	Se crea una notificación al usuario.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
Enviar una notificación al usuario			
<b>RESUMEN</b>			
El sistema enviará una notificación al usuario cuando la condición de una Alarma se cumpla.			

Tabla 4.53: Caso de uso CU-10

<b>CASOS DE USO</b>	Crear plantilla	<b>ID</b>	CU-11
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-11		
<b>PRECONDICIONES</b>	El usuario debe de estar registrado. Debe existir un dispositivo.		
<b>POSTCONDICIONES</b>	Creación de una plantilla elegida por el usuario.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
Crear una plantilla del usuario			
<b>RESUMEN</b>			
El usuario meterá los datos de la plantilla, el sistema los validará y verificará y se añadirá la mencionada plantilla al repositorio del usuario.			

Tabla 4.54: Caso de uso CU-11

<b>CASOS DE USO</b>	Borrar Plantilla	<b>ID</b>	CU-12
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-12		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. La plantilla tiene que existir.		
<b>POSTCONDICIONES</b>	La plantilla ya no existe en el sistema.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Eliminar la plantilla deseada del usuario.		
<b>RESUMEN</b>	El usuario escogerá en una lista la plantilla que desee eliminar y al darle click sobre él o algún botón para ese uso el sistema borrará la plantilla de ese usuario.		

Tabla 4.55: Caso de uso CU-12

<b>CASOS DE USO</b>	Añadir Widget	<b>ID</b>	CU-13
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-13		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. Debe existir un dispositivo.		
<b>POSTCONDICIONES</b>	El widget ha sido añadido a la plantilla.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Añadir un widget a una plantilla.		
<b>RESUMEN</b>	El usuario meterá los datos requeridos por el widget, el sistema los validará y verificará y se añadirá a la seleccionada plantilla del usuario.		

Tabla 4.56: Caso de uso CU-13

<b>CASOS DE USO</b>	Borrar Widget	<b>ID</b>	CU-14
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-14		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. Debe existir un widget.		
<b>POSTCONDICIONES</b>	El widget no está asociado a la plantilla.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Borrar un widget de una plantilla.		
<b>RESUMEN</b>	El usuario escogerá el widget que desee eliminar y al darle click sobre él o algún botón para ese uso, el sistema borrará el widget de la plantilla de ese usuario.		

Tabla 4.57: Caso de uso CU-14

<b>CASOS DE USO</b>	Guardar datos recibidos	<b>ID</b>	CU-15
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-15		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. Debe existir un dispositivo.		
<b>POSTCONDICIONES</b>	Los datos serán guardados.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Guardar los datos recibidos de un dispositivo.		
<b>RESUMEN</b>	El usuario puede decidir si los datos del dispositivo serán guardados o no.		

Tabla 4.58: Caso de uso CU-15

<b>CASOS DE USO</b>	Procesar datos recibidos	<b>ID</b>	CU-16
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-16		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. Debe existir un dispositivo. Debe haber datos recibidos en el sistema.		
<b>POSTCONDICIONES</b>	Datos estructurados.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Procesar datos recibidos.		
<b>RESUMEN</b>	El sistema procesará los datos para guardarlos de una manera que puedan ser útiles para el usuario.		

Tabla 4.59: Caso de uso CU-16

<b>CASOS DE USO</b>	Recibir notificación	<b>ID</b>	CU-17
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-17		
<b>PRECONDICIONES</b>	El usuario tiene que haber iniciado sesión. El dispositivo tiene que pertenecer al usuario. La alarma tiene que existir. La alarma tiene que estar encendida. La condición de la alarma tiene que haber sido violada.		
<b>POSTCONDICIONES</b>	El usuario recibe una notificación.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>	Recibir notificación.		
<b>RESUMEN</b>	El usuario recibirá una notificación del sistema cada vez que una alarma encendida sea avisada.		

Tabla 4.60: Caso de uso CU-17

<b>CASOS DE USO</b>	Visualizar datos de forma gráfica	<b>ID</b>	CU-18
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-18		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. Debe existir un dispositivo. Debe haber datos recibidos en el sistema.		
<b>POSTCONDICIONES</b>	El usuario podrá ver de forma gráfica los datos.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
Mostrar gráficamente los datos recibidos.			
<b>RESUMEN</b>			
El sistema mostrará de forma gráfica los datos que el usuario le pida a través de los distintos widgets de las plantillas.			

Tabla 4.61: Caso de uso CU-18

<b>CASOS DE USO</b>	Enviar datos al dispositivo	<b>ID</b>	CU-19
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-19		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. Debe existir un dispositivo.		
<b>POSTCONDICIONES</b>	Se enviará un mensaje al dispositivo.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
Enviar un mensaje a un dispositivo del usuario.			
<b>RESUMEN</b>			
El usuario a través de los widgets podrá enviar mensajes al dispositivo.			

Tabla 4.62: Caso de uso CU-19

<b>CASOS DE USO</b>	Cambiar plantilla	<b>ID</b>	CU-20
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-20		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. La plantilla tiene que existir.		
<b>POSTCONDICIONES</b>	La plantilla mostrada es otra.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
	Cambiar plantilla visualizada.		
<b>RESUMEN</b>			
	El usuario dispondrá de una lista de plantillas que él haya creado, de la cual podrá ir seleccionando la plantilla que quiera ver en ese momento y podrá cambiarla por otra de la lista.		

Tabla 4.63: Caso de uso CU-20

<b>CASOS DE USO</b>	Mostrar Plantilla	<b>ID</b>	CU-21
<b>ACTORES</b>	A-01		
<b>TIPO</b>	Esencial		
<b>REFERENCIAS</b>	RF-21		
<b>PRECONDICIONES</b>	El usuario tiene que estar registrado. La plantilla tiene que existir.		
<b>POSTCONDICIONES</b>	Se mostrará la plantilla elegida.		
<b>AUTOR</b>	Fco Javier Merchán Martín	<b>Versión</b>	1.0
<b>PROPÓSITO</b>			
	Mostrar una plantilla del usuario.		
<b>RESUMEN</b>			
	El usuario dispondrá de una lista de plantillas que él haya creado, de la cual podrá seleccionar la plantilla que quiera ver.		

Tabla 4.64: Caso de uso CU-21

#### 4.2.2. Esquemas de casos de uso

##### Usuarios

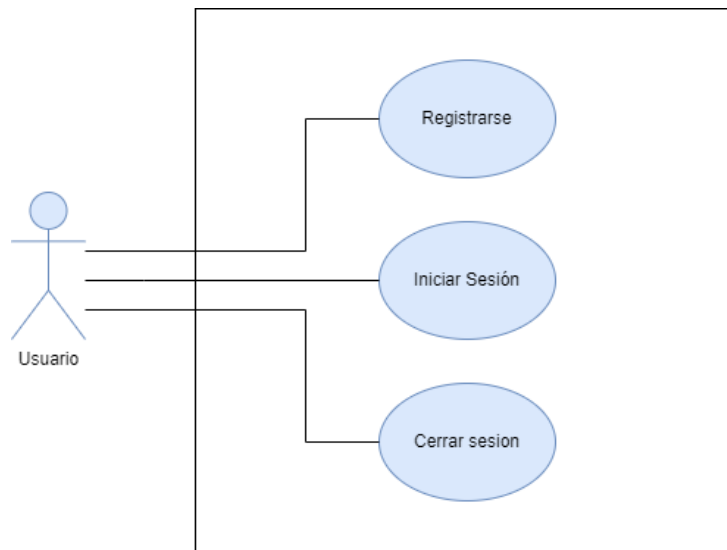


Figura 4.1: Esquema Casos de uso Usuarios

## Dispositivos

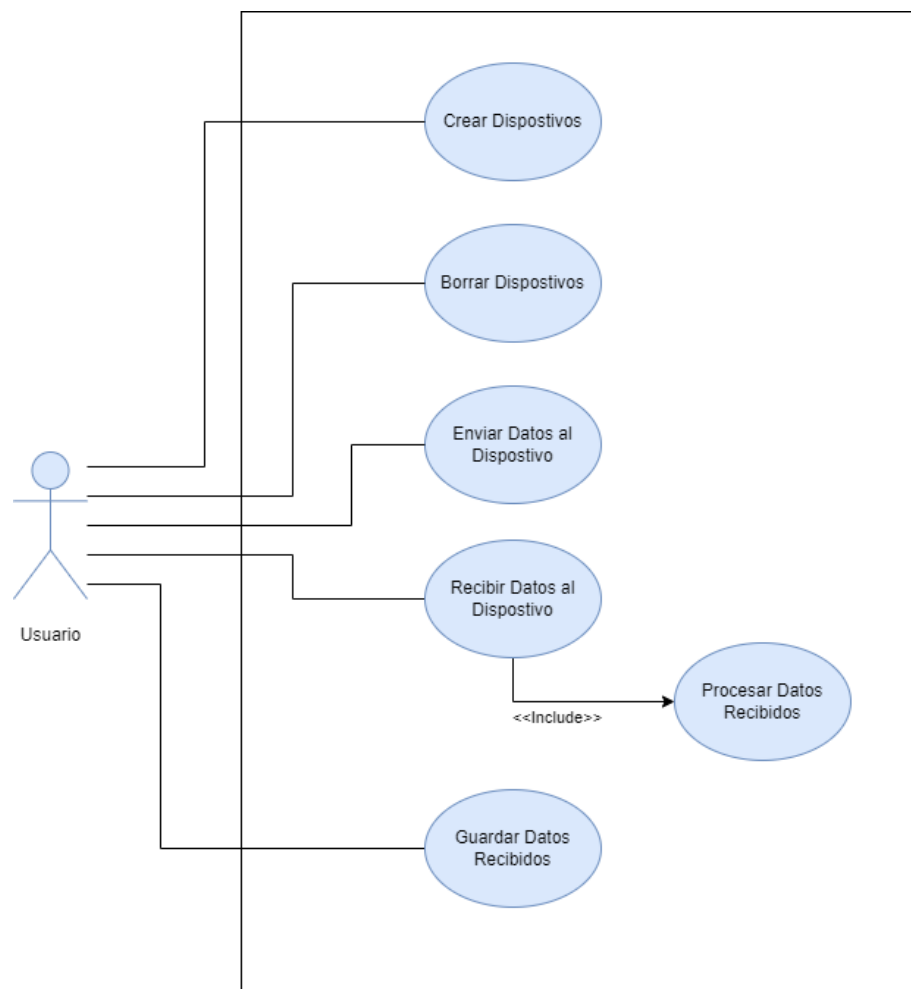


Figura 4.2: Esquema Casos de uso Dispositivos



## Alarmas y Notificaciones

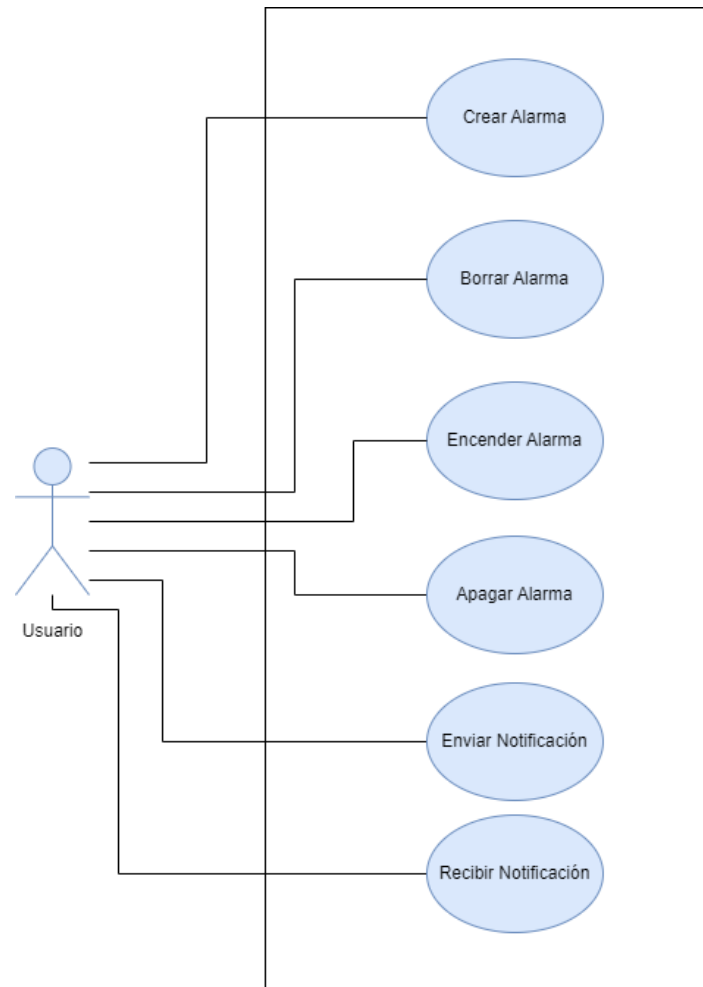


Figura 4.3: Esquema Casos de uso Alarmas y Notificaciones

## Plantillas

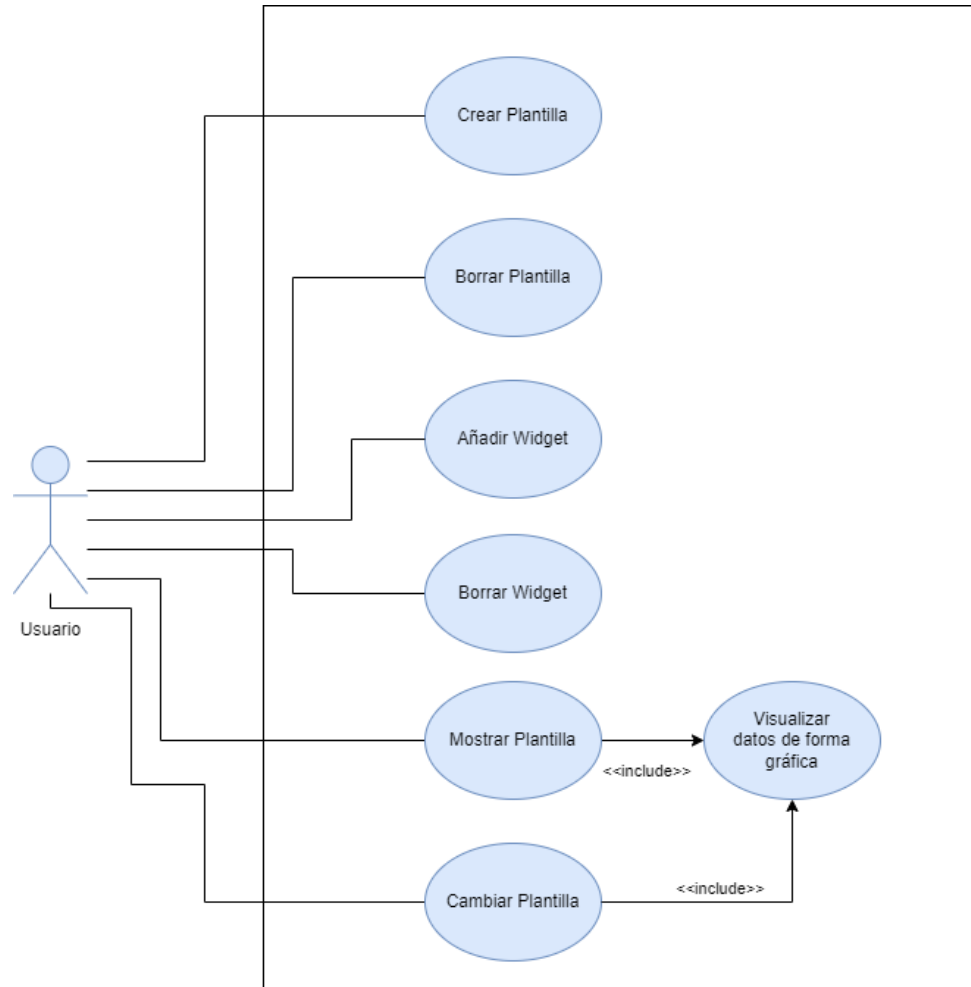


Figura 4.4: Esquema Casos de uso Plantillas

### 4.3. Diseño arquitectónico

Abajo podemos ver un esquema general de los componentes del diseño inicial. Nos ayudará a hacernos una idea de las partes que contiene un programa desarrollado para la plataforma, como hemos realizado una demo, este será su esquema.

El diseño inicial (Figura 4.5) consiste en una aplicación que usa el usuario que recibe y envía datos al servidor, unos dispositivos que hacen lo mismo y el servidor se comunica con la Base de datos correspondiente. Este tipo de comunicación es común en muchos tipos de aplicaciones.

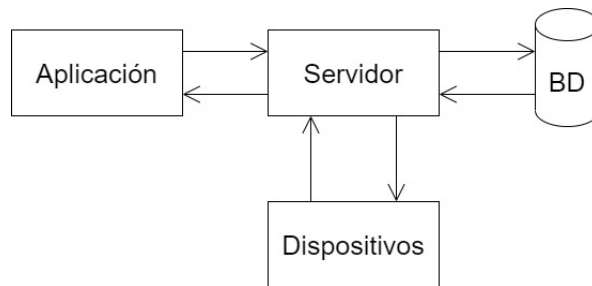


Figura 4.5: Diseño general de la propuesta

A continuación, se muestra un esquema más específico donde vemos los actores y otros elementos como la API del servidor, así como las partes necesarias para el protocolo MQTT.

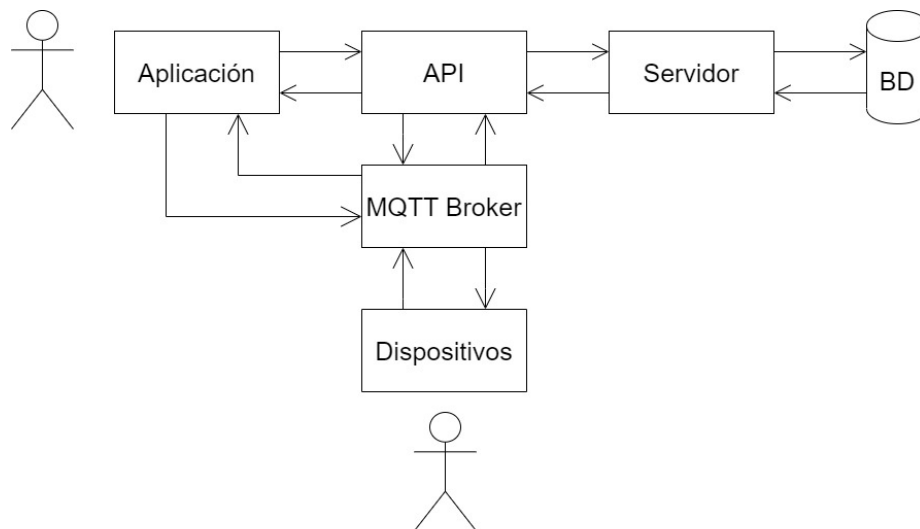


Figura 4.6: Diseño específico

En el diagrama anterior (Figura 4.6), podemos apreciar una particularidad, y es que a diferencia de otras soluciones para el paradigma de IoT como Zigbee donde cada dispositivo se conecta con otro haciendo de enlace con el siguiente,

en MQTT es un poco diferente ya que se necesita de un intermediario para poder gestionar esa mensajería entre dispositivos. .

A este intermediario lo llamamos broker, es el encargado de distribuir los mensajes de un publicador a los distintos clientes suscritos a el topic del publicador. Se ha dibujado esa manera para ejemplificar que es el broker el corazón del protocolo ya que sin él no podrían dispersarse los mensajes. Eso no significa que el broker tenga que estar situado en un único espacio. Ahí entra una de las elecciones que se han hecho y es que el broker EMQX nos da la facilidad para crear distintos nodos a lo que poder conectarnos, quedando así una red distribuida y altamente escalable.

En esta imagen siguiente (Figura 4.7) podemos ver cómo funciona el protocolo MQTT y ejemplifica muy bien lo que estamos diciendo. El cliente A solo recibe el mensaje cuando se ha conectado al broker y se ha suscrito al topic, el cliente B como no se ha suscrito a nada no recibe ningún mensaje.

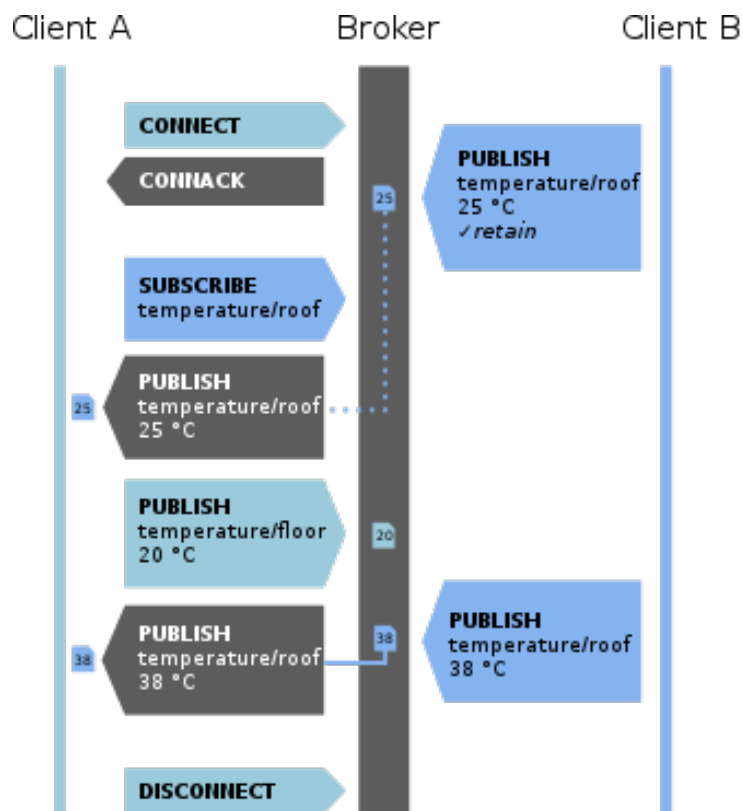


Figura 4.7: Diagrama con un un ejemplo de paso de mensaje [13]

Otro tipo de implementación diferente a la vista en MQTT es la Zigbee, este protocolo IoT en que como hemos dicho antes, cada dispositivo se conecta con otro haciendo que todos ellos forman una red, comunicándose entre ellos directamente. Aquí puede suceder lo mismo, pero para ello ambos dispositivos deberían incluir un broker y un cliente al mismo tiempo para poder comunicarse entre sí.

El problema de Zigbee es que pertenece a una empresa privada, por lo que esta es la que tiene que certificar los aparatos compatibles, así como la necesidad dentro de estos de un módulo específico para el uso de este protocolo. Esto hace referencia al problema que hemos dicho anteriormente al que cada empresa desarrolla su solución sin tener en cuenta la integración en otros ecosistemas. MQTT no es así, ya que se basa en TCP/IP por lo que cualquier dispositivo que permita este protocolo es potencial a que se pueda ejecutar un cliente MQTT, ya que este protocolo es en el que se basa internet, pues tenemos millones de dispositivos potencialmente compatibles con él, además de que es Open Source.

Volviendo a las partes del esquema (Figura 4.6), hemos hablado de las comunicaciones de los dispositivos a través del protocolo MQTT. Ahora vamos hablar un poco sobre la API. Esta parte se encarga de hacer de intermediario entre los dispositivos y el servidor junto con la base de datos. En un principio se optó por hacer una API de tipo REST, pero exigía muchos requisitos muy estrictos, por lo que se optó por una API normal, que mezclaba varias cualidades de REST (utilización de peticiones Delete para borrar o de PUT para actualizar, etc.) con las de una API web (peticiones par servir a paginas web), ayudando así a montar las parte de front-end de la plataforma.

Esta API también incluye un cliente MQTT para poder obtener mensajes y así enviarlos de manera automática a la base de datos o para enviar datos a otros dispositivos. Gracias a una característica exclusiva del broker EMQX, los recursos, los cuales hacen de webhook o gancho filtrando mensajes cuando llegan al broker entre otras funciones disponibles, podemos llevar acabo lo dicho anteriormente. Un ejemplo de uso en nuestra plataforma, es que se crean dos recursos, uno de ellos se encarga de redirigir los menajes entrantes para guardar y el otro se encarga de las alarmas, que son condiciones que ponen el usuario que cuando se cumple se le avisa mediante una notificación.

Para poder comunicarse con la API en la mayoría de sus end-points es necesario un token, el cual dice si estás autorizado o no. Se ha utilizado JWT que significa Json Web Token para generar y verificar ese token, haciendo así la plataforma más segura. Volviendo a los end-points, más adelante en el apartado correspondiente se ahondará en las distintas urls accesibles. Antes de acabar de hablar de la API, decir que no todos los end-points han sido creados para controlar el acceso a los datos de la BDD y por lo tanto podrían ser públicos, si no que otros se han creado para servir información al front-end o para la gestión interna de la plataforma.

Continuando con el esquema (Figura 4.6), podemos apreciar que se incluye un servidor, que aunque se encuentre separado es el que albergará la API y esta se conectará con la base de datos, haciendo que solo se tenga acceso a través de la API que hace de filtro o cortafuegos.

Por último, tenemos los dispositivos y la aplicación. Esta última hace referencia al front-end pero podría pasar como un dispositivo más ya que puede obtener la misma información, se puede decir que es un tipo de dispositivo especial, el cual no tiene ningún sensor. Con esto ya podemos definir que es un dispositivo en nuestro sistema.

Un dispositivo es en esencia un sensor o conjunto de estos que se comunican a través de un único cliente MQTT al broker. Estos dispositivos pueden programarse como se quieran, mientras sigan unas normas a la hora de enviar o recibir información del broker para así crear una gramática común a la hora de comunicarse y que sea posible entenderse entre dispositivos. Estos dispositivos también se comunican con la API ya que necesitan de una autorización para conectarse al broker y que así haya más seguridad de que no se conecta nadie externo o acceda a información que no le pertenece.

Podemos verlos en parte como módulos, los cuales pueden cambiarse, modificarse o ampliarse de manera lo más fácil posible. Es más, el modelo de datos pensado inicialmente se cambió a mitad de proyecto, lo cual hizo palpable esa idea de que se pudiera cambiar ciertas partes por otras de la forma más sencilla posible.

Algunos cambios son más complicados que otros, por ejemplo, el uso del broker está estrechamente relacionado con varias funciones de la aplicación, ya que se hace uso de características exclusivas que hemos mencionado enteramente para funcionalidades de la plataforma, como el sistema de recursos que está relacionado con la parte de notificaciones, alarmas y webhooks.

## 4.4. Modelo de datos

Para la plataforma se han desarrollado un modelo de datos en los que los dispositivos tiene la mayor importancia, adicionalmente se ha desarrollado una entidades complementarias para los distintas partes del la plataforma como son las Plantillas, las Alarmas, ect.

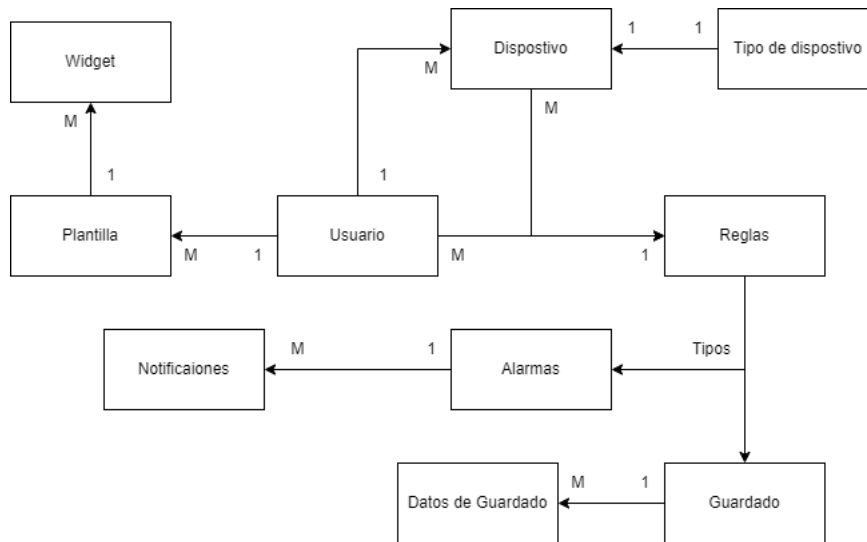


Figura 4.8: Esquema de relación de los datos

Como podemos ver en el esquema (Figura 4.8) podemos ver 5 entidades de datos distintos.

- **Tipo de Dispositivo:** Esta entidad enmarca a los datos básicos de un dispositivo y alberga datos comunes entre aquellos que son del mismo tipo, como por ejemplo los sensores que contienen.
- **Dispositivo:** Esta entidad trata de describir a los distintos dispositivos pertenecientes a un usuario. Cada dispositivo tiene que tener asignado un tipo de dispositivo que es el que le asigna las distintas variables. Estas variables hacen referencia a los sensores y/o alguna otra cosa que reciba información y la aplique (por ejemplo, un switch o conmutador), aunque los generalizaremos como sensores. Al final podemos decir que un dispositivo es un conjunto de uno o más sensores que envía y recibe información del sistema o usuario y/o permite hacer acciones sobre elementos reales. Una variable está compuesta por:
  - **Nombre:** El nombre del sensor.
  - **Tipo:** El tipo de dato, por ejemplo, boolean, number u otro.
  - **Nº de buffer:** El número de datos que se envían al mismo tiempo, con valor estático en 1 (preparado para futuras ampliaciones).
  - **Widget Compatibles:** Lista con los widgets que son compatibles con estos datos.
  - **Frecuencia:** Periodo de tiempo que tarda en enviar los datos, en ms.

- **Usuario:** Aquí se intenta representar que es el usuario, su campo clave es el correo electrónico y que está puesto para que sea único, aunque para asegurar el anonimato cuando se transmite información solo se usará para acceder, en este caso puede ser sustituible por un nombre de usuario o nick, aunque se querido incluir el correo electrónico para posibles futuras funciones. Como identificador se utilizará el id único que genera Mongo cuando se crea un documento.
- **Plantilla:** Esta entidad hace referencia a una plantilla que es una versión del tablón o Dashboard hecho por el usuario. Está compuesta por Widgets, que es la representación gráfica de las variables de los dispositivos y al final es la interacción de este con el usuario. Se pueden identificar con el nombre, pero puede ser poco escalable en grandes sistemas por lo que se identifican por el id que le otorga Mongo al crear el objeto. Un usuario puede tener más de una plantilla y estas tendrán diferentes Widgets que serán configurados por el usuario.
- **Reglas:** Esta entidad representa las reglas que se crean para el funcionamiento de distintas funciones para el usuario, por ejemplo, la opción de guardar o no los datos. Estas reglas pueden activarse o desactivarse. Se han considerado dos subtipos respecto:
  - **Reglas de Guardado:** Que presenta si se guardan o no los datos de un dispositivo concreto.
  - **Reglas de Alarma:** Que representa una condición que configura el usuario sobre una variable de un dispositivo y que si se cumple se notifica al usuario.
- **Notificación:** Aquí se enmarcan las notificaciones que se envían al usuario cuando una regla de tipo alarma se cumple la condición. Al igual que en los casos anteriores, se identifican con el ID que da Mongo, aunque en la implementación propuesta también pueden identificarse por ID que le da el broker EMQX ya que este es único dentro del sistema.
- **Emqx Auth:** En esta entidad se representa las autenticaciones para conectarse al broker y poder acceder de forma segura, se representa por el valor ID que le da Mongo al crearlos. Además, también se limita a los tópicos que te puedes suscribir o publicar por lo que también obtenemos privacidad.
- **Data dispositivos:** Esta entidad representa a los datos guardados de cada dispositivo, en cada momento y de cada variable distinta. Están identificados por los ID que otorga Mongo cuando se crean.



Con estas especificaciones el resultado es la siguiente distribución llevada a Mongo:

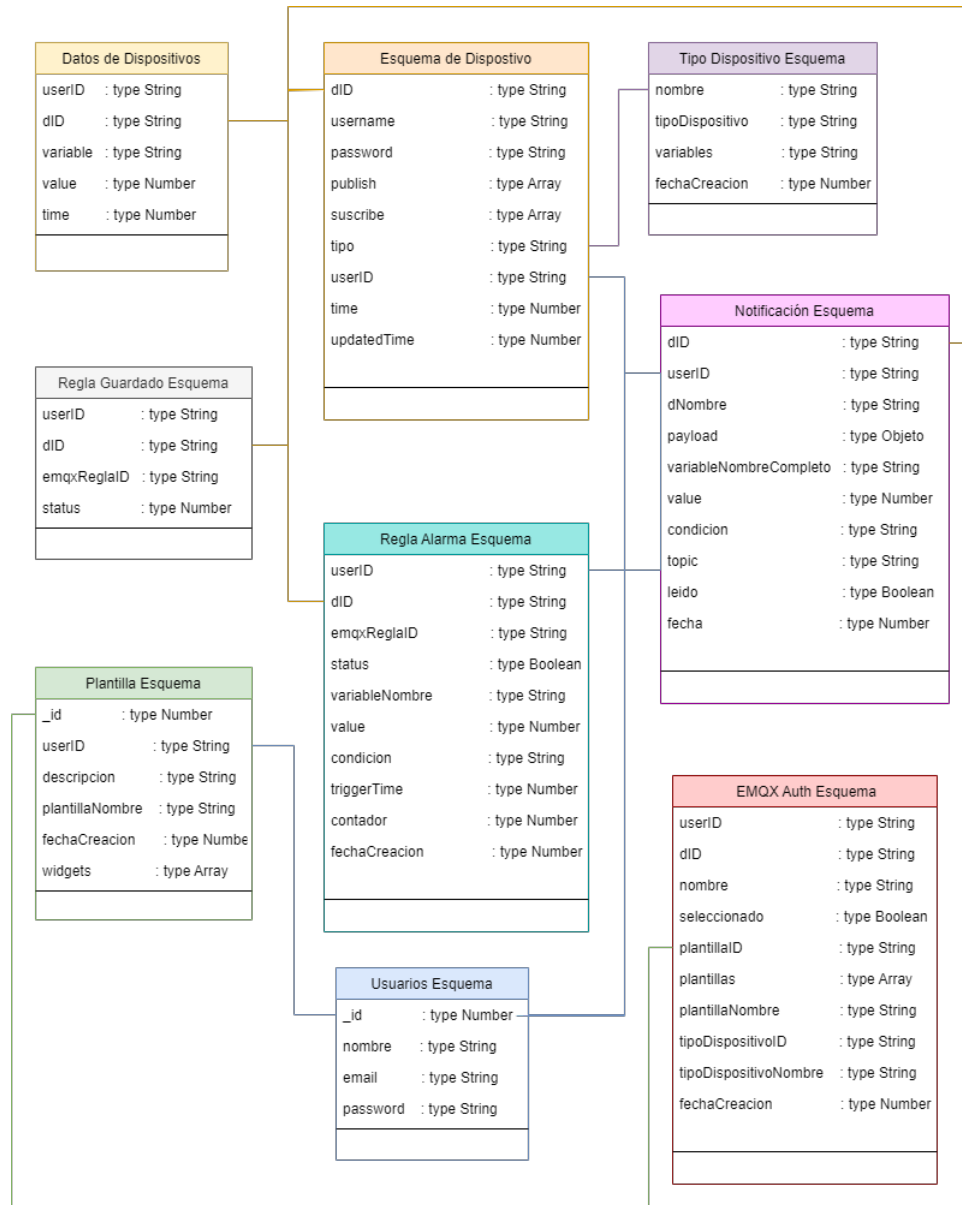


Figura 4.9: Diagrama de las base de datos

## 4.5. Front-end

Para la parte de front-end se ha barajado varias opciones de frameworks de desarrollo, entre los que están React y Nuxtjs. Debido a la gran versatilidad del último y a su suave curva de aprendizaje se eligió este, además tiene una jerarquía de carpetas bastante útil para tenerlo todo ordenado. Aunque este front-end desarrollado es solo un ejemplo y con las adaptaciones necesarias se puede construir en React o cualquier otro framework de node.js, ya que se hace una abstracción por parte de la API que sirve los datos al front-end para que sea lo más independiente posible.

Como hemos dicho antes este front-end está desarrollado sobre el framework llamado Nuxt, que a su vez está basado sobre otro framework llamado Vue.

Para el front-end se ha escogido una distribución de los contenidos en la que en una columna, el usuario tiene las distintas funciones de la aplicación y a la derecha el contenido de esta página. Esto ayuda a que si en un futuro se tiene que añadir más funcionalidades Solo hay que crear una página en vue para esa funcionalidad esto da una mayor claridad a la aplicación.

También se han incluido dos páginas para hacer el logueo y el registro de nuevos usuarios. Con el primero de ellos se obtiene un token para no tener que loguearse de nuevo pasado un tiempo o hasta que el usuario quiera salir.

Para optimizar el tiempo de desarrollo se ha decidido usar una plantilla de estilo de un dashboard [16], donde todo el css ya viene incluido, así como una paleta de colores. Además, se ha escogido un fondo oscuro ya que aporta una mayor comodidad a la vista en periodos largos de uso, así como ahorro energético en algunas pantallas como las OLED.

Adicionalmente esta plantilla de estilo también incluía un sistema de mensajes emergente que han ayudado para poder desplegar las notificaciones de manera más eficiente. Estas plantillas tienen una versión de pago y una gratuita con menos características, se ha escogido la gratuita (Licencia MIT) ya que la única condición es que sea usada sin ánimo de lucro, cosa que nuestra plataforma es y se referencia al autor, cosa que hemos hecho en la bibliografía[16] y en la propia plataforma en el footer.

La organización de las carpetas es la siguiente:

- **Asserts:** En la que encontraremos todos los archivos o elementos no compilados que necesita nuestra aplicación, por ejemplo, imágenes, documentos, etc.
- **Components:** Aquí encontraremos los distintos componentes de nuestra aplicación. En nuestro caso la mayoría de estos componentes son partes

de la maquetación de la plantilla de estilo que hemos usado. En esta carpeta también se incluirían componentes desarrollados por nosotros por ejemplo los widgets, aunque en este caso se han ubicado fuera para mayor comodidad.

- **Layouts:** Esta carpeta contiene las templates usadas para las páginas generales, es decir, cada página de la carpeta `pages` tiene una plantilla asignada. Existe una por defecto si no se le asigna ninguna.
- **Middleware:** Contiene como su nombre indica los Middleware de la aplicación, es decir, las cosas que se ejecutan entre medias de lo que ve el usuario y la aplicación. Por ejemplo, se han creado dos para ver si están o no autenticados dependiendo de las necesidades de la página.
- **Pages:** Aquí encontramos con lo que va a ver el usuario, las páginas, al final el usuario va a ir saltando de una a otra dependiendo de donde haga click o si lo redirigimos nosotros manualmente. A diferencia de las páginas normales, en `.html`, estas terminan en `.vue`. Tienen asignada un template cada una que le indica todo el entorno.
- **Plugins:** En esta carpeta se encuentran plugins en Javascript que se ejecutan antes de montar el proyecto de Vue.
- **Static:** Contiene los elementos estáticos de la plataforma y estos están maquetados.
- **Store:** Contiene archivos de la Vuex Store, para nuestra aplicación es una especie de almacenamiento privado a la que solo tiene acceso la aplicación.
- **Widgets:** En esta carpeta se almacenan las componentes que hemos realizados nosotros para su uso en las plantillas del dashboard, es decir, los widgets. Deberían estar en la carpeta `components` pero se sacó para mayor facilidad en el desarrollo.

También encontramos otros archivos, como los de configuración para la ejecución de node.js como es por ejemplo el archivo `package.json`, que contiene los módulos necesarios para lanzar la plataforma. O archivos generados cuando ejecutamos el proyecto por ejemplo `node_module`, que contiene los módulos que se han listado en el archivo antes mencionado.

Metiéndonos un poco más en la organización, cada página tiene un template como referencia. En este caso el de las páginas es distinta a las de login y registro ya que estos últimos solo incluyen el recuadro o card (término que denomina a una de las componentes usadas) con lo que necesitas para ambos propósitos.

Todo lo que escribamos en ambas páginas deben estar entre las marcas `<templates>` para que renderice en el template. Y esto se “pegará” dentro de esta etiqueta .

```

1 <!-- Panel central-->
2 <div class="main-panel" :data="sidebarBackground">
3   <!--Barra de arriba-->
4   <dashboard-navbar></dashboard-navbar>
5   <router-view name="header"></router-view>
6   <div
7     :class="{ content: !isFullScreenRoute }"
8     @click="toggleSidebar">
9     <zoom-center-transition :duration="200" mode="out-in">
10       <!-- your content here -->
11       <nuxt>
12
13     </zoom-center-transition>
14   </div>
15   <!--Footer-->
16   <content-footer v-if="!isFullScreenRoute"></content-footer>
17 </div>

```

Código Fuente 4.1: Main panel de un template

Para almacenar las distintas variables necesarias para el funcionamiento del front-end se usa la función de store de Nuxt y el localStorage de la máquina del usuario. En este último lo único que se guarda es una copia del token para iniciar sesión. En store de Nuxt podemos almacenar state o estados que guardaran datos para uso entre las páginas, es como el local storage pero sin que usuario final tenga acceso, con ello se pueden dejar de usar las cookies. En este archivo también debemos de definir los métodos para mutar los datos, es decir, para modificar los datos. También podemos definir otras funciones auxiliares o acciones sobre el store.

```

1 //Estados o variables
2 export const state = () => ({
3   dispositivos: [],
4   auth: null
5 })
6
7 //Mutaciones de los stores
8 export const mutations = {
9   setAuth(state, authRecibido){
10     state.auth = authRecibido
11   },
12   setDispositivos(state, dispositivosEntrada) {
13     state.dispositivos = dispositivosEntrada;
14   }
15 }
16
17 //Acciones
18 export const actions = {
19   leerToken() {
20     let auth = null;
21     try {
22       auth = JSON.parse(localStorage.getItem('auth'));
23     } catch (error) {
24       console.log(err);
25     }

```

```

26 //Guardamos el token en state
27 this.commit('setAuth', auth)
28 }
29 }

```

Código Fuente 4.2: Ejemplo Store de Nuxt

Al igual que el back-end también se pueden crear middleware para poder hacer acciones entre páginas o antes de que carguen, por ejemplo, este en el que se comprueba si el usuario tiene un token válido para poder redirigirlo al login si no es así.

```

1 //Enviamos al usuario al login si no tiene token
2 export default function ({ store, redirect }){
3   //Ejecutamos la acción de leer el token
4   store.dispatch("leerToken");
5   //Si no tiene token lo mandamos al login
6   if (!store.state.auth){
7     return redirect("/login");
8   }
9 }

```

Código Fuente 4.3: Middleware de autenticación

Para poder hacer uso de distintas funcionalidades se pueden crear componentes, algunos ya vienen con el tema como son por ejemplo las tablas o la componente card. Aunque para este proyecto se han desarrollado otras componentes para poder hacer un uso mejor de los datos y añadir funcionalidades, en nuestro caso se ha creado bajo el nombre de widgets los cuales pueden ampliarse en un futuro ya que su desarrollo es muy simple y su utilización también lo es.

### 4.5.1. Widgets

Estos son los componentes que hemos desarrollado para mostrar información e interactuar con los distintos dispositivos. Son solo un ejemplo de lo que podemos construir y más abajo veremos cómo el usuario puede configurarlos. Estos Widgets como hemos dicho anteriormente son la forma visual de las variables, que a su vez son los distintos sensores de los dispositivos.

#### 4.5.1.1. Botón

Se trata de un botón que manda una señal al dispositivo cuando se pulsa un botón, viene con un símbolo personalizable.

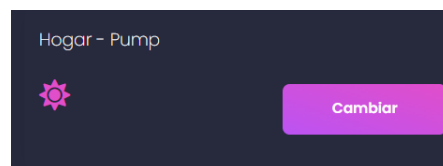


Figura 4.10: Widget Botón

#### 4.5.1.2. Switch

En esta ocasión se ha desarrollado la misma funcionalidad que el botón, pero con un botón deslizable, como si fuera un interruptor.



Figura 4.11: Widget Switch

#### 4.5.1.3. Indicador

Este widget está asociado a una variable la cual muestra su situación en dos valores. Pero no permite la interacción.

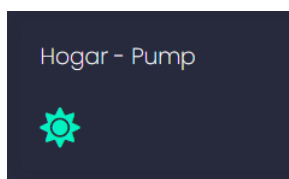


Figura 4.12: Widget Indicador

#### 4.5.1.4. Gráfica numérica

Se trata de una gráfica que va, la cual es totalmente configurable tanto el tamaño, el periodo, etc.

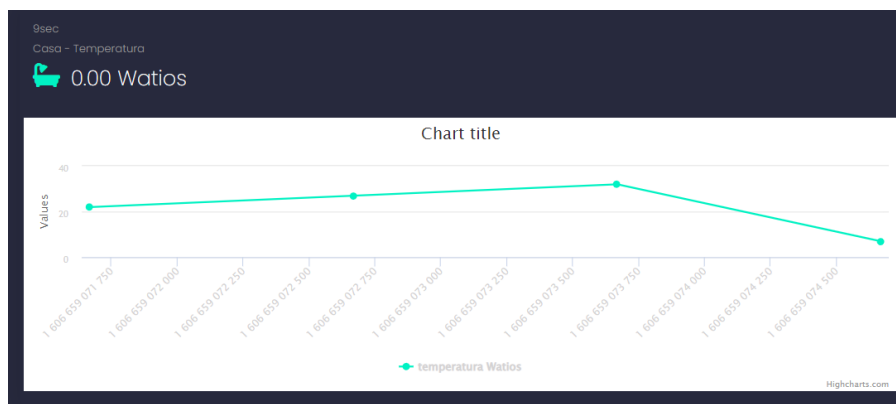


Figura 4.13: Widget Gráfica

## 4.5.2. Páginas

Se han desarrollado distintas páginas, son las secciones que se definirán más abajo y que ocupan la parte central de la portada, para las distintas funciones de la plataforma, a continuación, vamos a mostrar cuales han sido. Pero antes vamos a ver cómo es la parte en común entre ellas.

### 4.5.2.1. Marco

En esta ocasión cuando hablamos del marco o template, concretando más nos estamos refiriendo a todo lo que envuelve a lo escrito en cada página y están contenido de la carpeta *layouts* antes mencionada. Estos archivos contienen todo lo común entre páginas. Esto hace más fácil cuando se quiere cambiar algo que está en todas las páginas ya que solo se cambia 1 vez en el template correspondiente y no en cada página. El contenido de lo que nosotros hemos definido como página, se encuentra en lo que se llama panel central o main si hablamos de etiquetas de HTML5. Esto forma una jerarquía dentro del front-end donde las paginas dependen de los templates para mostrarse de forma completa.

Un poco más arriba hemos hablado de donde se sitúa el contenido , dentro de las etiquetas `<nuxt>`, es ahí, donde Nuxt coloca nuestra página.

Ahora vamos a hablar un poco de los demás elementos que vemos en la imagen (Figura 4.14). Arriba tenemos una barra de navegación en ella nos encontramos varios botones como las notificaciones (Flecha roja en la Figura 4.14), que si lo pulsamos desplegará una lista de las notificaciones sin leer. Justo al lado un icono el cual representa al usuario (Flecha verde en la Figura 4.14) y que si pulsamos nos aparecerá la opción de cerrar sesión.

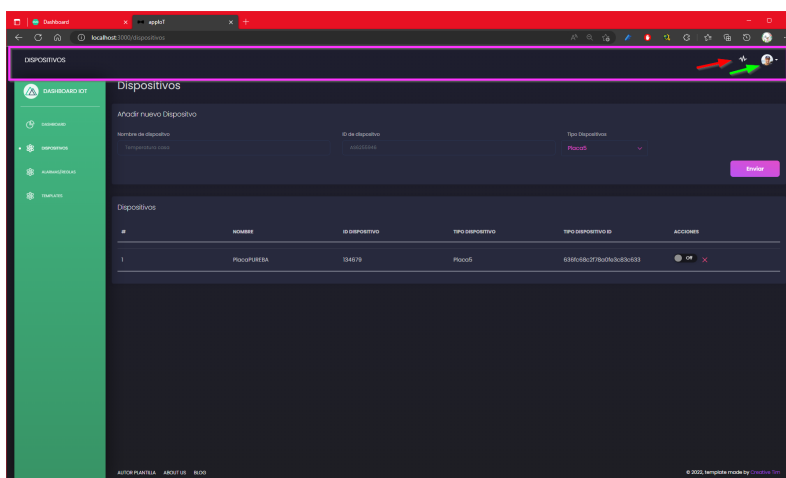


Figura 4.14: Página Dispositivos señalando la parte superior

Ahora si nos vamos a la izquierda nos encontramos una barra de navegación en la que encontraremos nuestras páginas (Flechas rojas en la Figura 4.15), al ser una página web SPA (Single Page Application), funciona como una aplicación y no hay tiempo de carga entre ellas, por lo que solo cambia la parte diferente en este caso la parte derecha, es decir la parte central (Recuadro azul en la Figura 4.15).

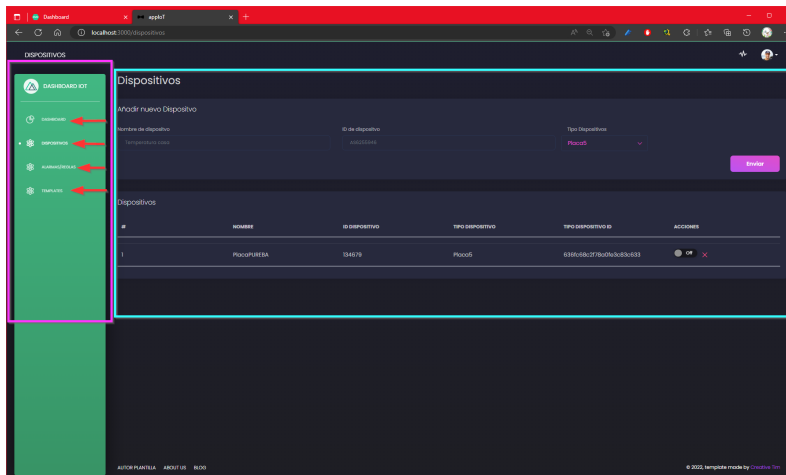


Figura 4.15: Pagina Dispositivos señalando la parte central

Por último, tenemos el footer donde tenemos varios enlaces, entre ellos a la vez del creador de dicha plantilla y links a otros sitios (Figura 4.16).

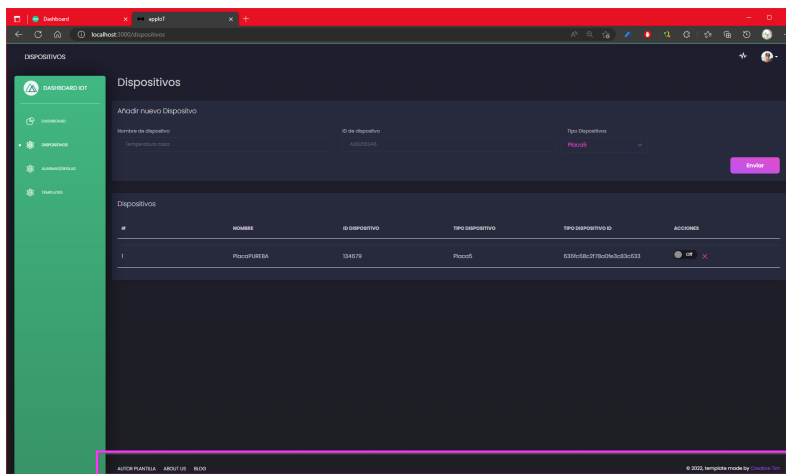


Figura 4.16: Pagina Dispositivos señalando la parte inferior



Si hablamos de la estructura del código, tenemos esta estructura:

```

1 <!-- Template por defecto de Creative Tim -->
2 <template>
3   <div class="wrapper" :class="{ 'nav-open': $sidebar.showSidebar
4     }">
5     <notifications></notifications>
6
7     <side-bar
8       :background-color="sidebarBackground"
9       short-title="DashIOT"
10      title="DashBoard IoT"
11    >
12      <template slot-scope="props" slot="links">
13        </template>
14    </side-bar>
15
16    <!--Share plugin (for demo purposes). You can remove it if
17    don't plan on using it-->
18    <!-- <sidebar-share :background-color.sync="
19    sidebarBackground"> </sidebar-share -->
20
21    <!-- Panel central-->
22    <div class="main-panel" :data="sidebarBackground">
23      <!--Barra de arriba-->
24      <dashboard-navbar></dashboar-navbar>
25      <router-view name="header"></router-view>
26
27      <div
28        :class="{ content: !isFullScreenRoute }"
29        @click="toggleSidebar"
30      >
31        <zoom-center-transition :duration="200" mode="out-
32        in">
33
34          <!--your content here-->
35          <nuxt> </nuxt>
36          </zoom-center-transition>
37        </div>
38        <!--Footer-->
39        <content-footer v-if="!isFullScreenRoute"></content-
40        footer>
41      </div>

```

Código Fuente 4.4: Parte html del archivo default.js de la carpeta layouts

En la parte de arriba tenemos todo el contenido en una etiqueta template, y dentro de esta otra etiqueta div y dentro de esta última desarrollamos todas las demás funciones como son las notificaciones, la barra lateral o el panel principal. Si necesitamos añadir algo en el sentido que queremos que esté en todas las páginas este es el lugar. Respetando siempre que la etiqueta nuxt ( Líneas 33 del código 4.4) será la que contendrá el contenido que irá cambiando, es decir, nuestra página.

Abajo tenemos la etiqueta de script (Código fuente 4.5), que contiene todas las demás funciones, aquí podemos incluir varias cosas, como son funciones que necesitemos en nuestra página o datos. Aquí también debemos importar como vemos los componentes que vayamos a utilizar.

```

1 <script>
2   /* eslint-disable no-new */
3   import PerfectScrollbar from 'perfect-scrollbar';
4   import 'perfect-scrollbar/css/perfect-scrollbar.css';
5   import SidebarShare from '@components/Layout/
   SidebarSharePlugin';
6   function hasElement(className) {
7     return document.getElementsByClassName(className).length >
8     0;
9   }
10
11   function initScrollbar(className) {
12
13     import DashboardNavbar from '@components/Layout/
   DashboardNavbar.vue';
14     import ContentFooter from '@components/Layout/ContentFooter.
   vue';
15     import DashboardContent from '@components/Layout/Content.vue';
16     import { SlideYDownTransition, ZoomCenterTransition } from '
   vue2-transitions';
17     import mqtt from 'mqtt';
18
19     export default {
20       components: {
21         DashboardNavbar,
22         ContentFooter,
23         DashboardContent,
24         SlideYDownTransition,
25         ZoomCenterTransition,
26         SidebarShare
27       },
28       data() {
29         return {
30           sidebarBackground: 'vue', //vue|blue|orange|green|
   red|primary
31           client: null,
32           options: { //Opciones de mqtt
33             port: 8883,
34             host: 'localhost',
35             endpoint: '/mqtt',
36             clean: true,
37             connectTimeout: 5000,
38             reconnectPeriod: 5000,
39
40             clientId: 'web' + this.$store.state.auth.
   datosUsuarios.nombre + "_" + Math.floor(Math.random()*(1-10000)
   * -1).
41             username: 'superuser',
42             password: 'superuser',
43           },

```

```

44     };
45   },
46   computed: {
47     isFullScreenRoute() {
48       return this.$route.path === 'maps/full-screen'
49     }
50   },
51   methods: {
52     toggleSidebar() {
53     },
54     initScrollbar() {
55     },
56     async getMqttCredenciales() {
57     },
58     async getMqttCredencialesReconnect() {
59     },
60     async startMqttClient() {
61     }
62   },
63   beforeDestroy() {
64   },
65   mounted() {
66   }
67 };
68 </script>

```

Código Fuente 4.5: Extracto de la parte script del archivo default.js de la carpeta layouts

Como vemos después de los import (Lineas del 3-5 del código 4.5) sigue una estructura la cual es la que hace que Nuxt funcione, en ella nos encontramos varios ganchos o hooks en inglés. Los más importantes son:

- **Components:** Para añadir los componentes.
- **Data:** Para los datos que se utilizan en la página o en las funciones de más abajo. Es importante seguir el estilo de objeto de javascript y poner los datos dentro del return, ya que lo que devuelve es un objeto.
- **Methods:** Es donde se encuentran las funciones de javascript para realizar tareas.
- **beforeDestroy:** Lo se encuentra dentro de este hook se ejecuta antes de destruir o salir de la página.
- **Mounted:** Lo que se encuentra aquí se ejecuta cuando se monta la página.

Estos son solo unos pocos, hay muchísimos más, los puede encontrar todos en la documentación de Nuxt[41].

Las paginas siguen el mismo estilo o patrón solo que no se encuentra la etiqueta nuxt.

#### 4.5.2.2. Dashboard

Esta es la página más visual y es la página objetivo del proyecto. En ella se muestran las plantillas realizadas por el usuario. Como un usuario puede tener varias plantillas y en ella diferentes widgets, por ello se ha habilitado un selector arriba a la derecha para que el usuario pueda cambiar fácilmente de plantilla.

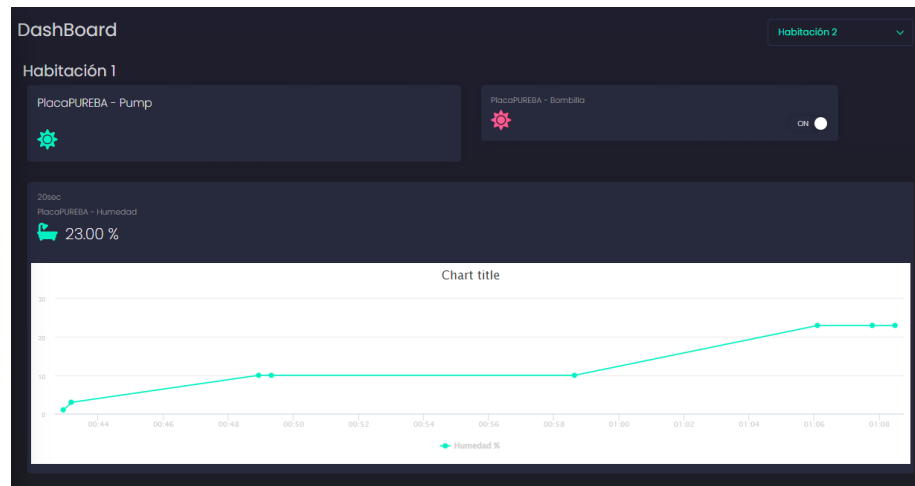


Figura 4.17: Sección de Dashboard

#### 4.5.2.3. Dispositivos

La página de dispositivos se creó para añadir dispositivos al usuario, en ella hay un formulario para añadir dispositivos y una tabla con los que tiene asignado el usuario.

Al final de cada fila de la tabla hay un botón en el que se elimina el dispositivo del usuario. Y un poco más a la izquierda del botón de borrado encontramos un switch con el que indicaremos si queremos guardar o no los datos recibidos por los dispositivos.

## Dispositivos

Nombre de dispositivo

Temperatura casa

ID de dispositivo

AS6255949

Plantillas

Selecciona la plantilla

Enviar

### Dispositivos

#	NOMBRE	ID DISPOSITIVO	PLANTILLA	ACCIONES
1	UnNombre	123445		✕
2	UnNombre3	12344523		✕
3	UnNombre4	1234452332	templateprub	✕

Figura 4.18: Sección de Dispositivos

#### 4.5.2.4. Plantillas

Para la página plantillas, se diseñó para crear tipos de dashboards en los que el usuario pudiera añadir con widgets a su gusto. Estos widgets son los que se comunicarán con la base de datos o los dispositivos y poder así ver los datos que recoge o interactuar con los dispositivos. Al final tenemos una lista con las plantillas y un poco de información de estas. También disponemos de un botón de borrar para borrarlas si ya no nos interesan.

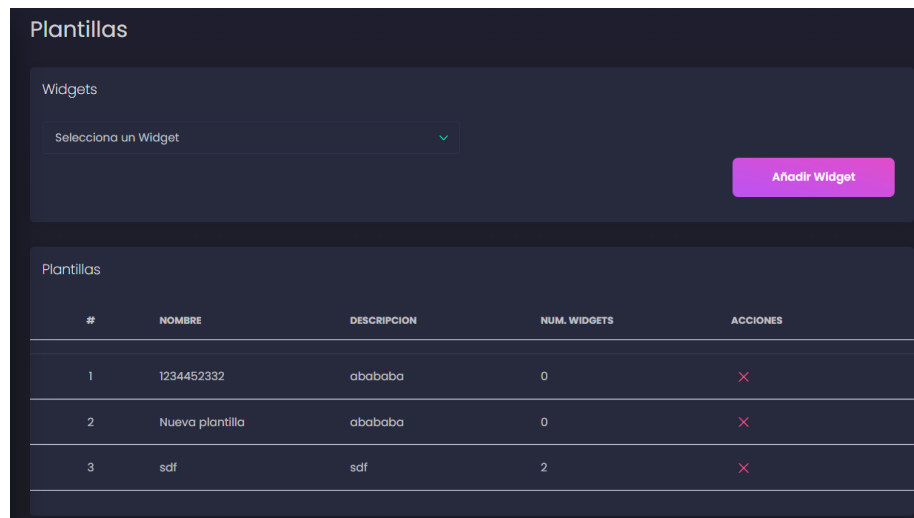


Figura 4.19: Sección de Plantillas

Cada widget es personalizable por lo que cada uno tiene su formulario para configurarlo. Basta con seleccionar el widget de la lista, rellenar los campos y darle a añadir. En la columna de la izquierda podemos ver cómo va quedando el widget, es decir, una preview.

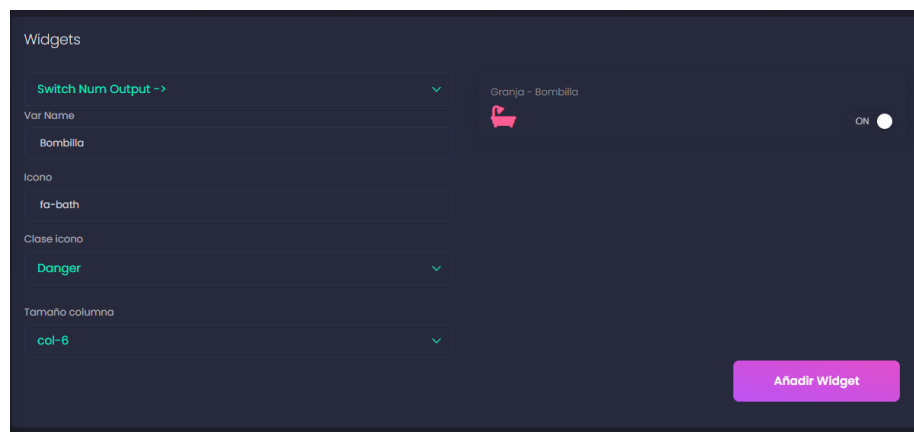


Figura 4.20: Sección de Plantilla con Widgets

Al darle añadir se nos abrirá abajo una previsualización de cómo va quedando la plantilla, cuando la tengamos lista basta con ponerle un nombre y una breve descripción y pulsar guardar (Figura 4.21).

Antes de guardar si nos hemos equivocado en algún widget y queremos borrarlo simplemente deberíamos pulsar el icono de la basura encima del que deseemos eliminar.

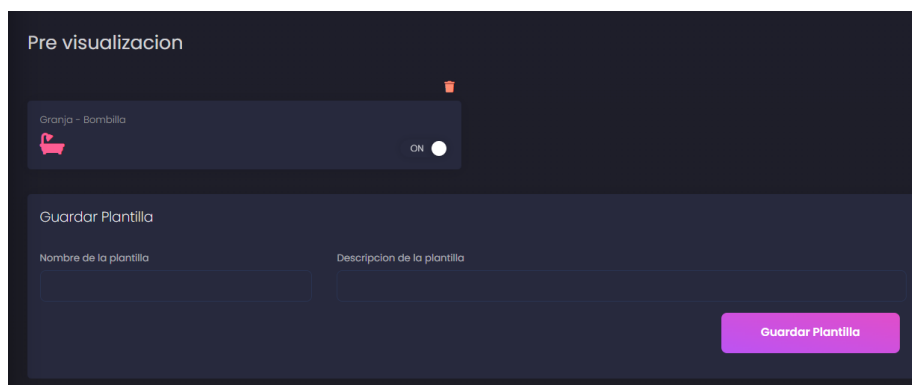


Figura 4.21: Sección de Plantilla con Previsualización y tabla

#### 4.5.2.5. Alarmas

En esta página tenemos la configuración de las alarmas. Estas alarmas son las que avisan al usuario a través de notificaciones de que se ha sobrepasado el umbral configurado. La disposición de la página como ocurre en otras páginas, al principio nos encontramos el formulario para rellenar.

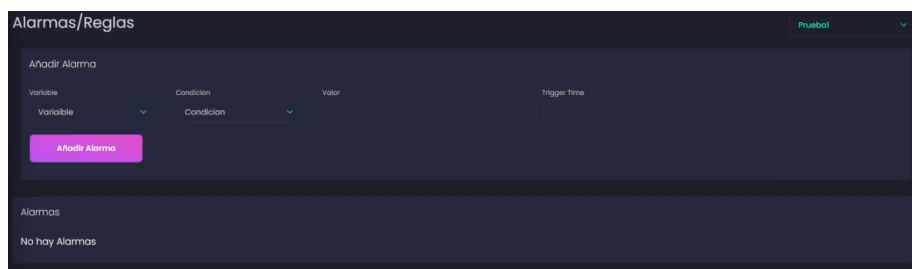
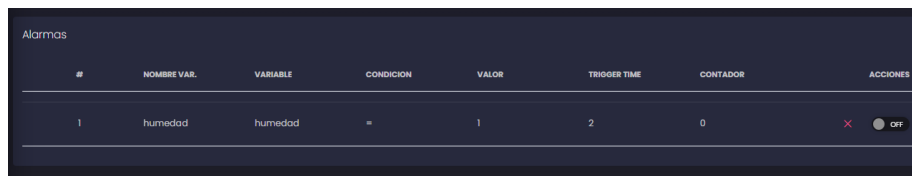


Figura 4.22: Sección de Alarmas

Como podemos observar en la imagen tenemos distintas configuraciones como la variable que queremos evaluar, incluso cada cuanto queremos que nos avise si vuelve a saltar la alarma (Trigger Time). Cuando tengamos todo relleno simplemente bastaría con hacer click en el botón de Añadir Alarma y la alarma aparecerá en la tabla de abajo.



#	NOMBRE VAR.	VARIABLE	CONDICION	VALOR	TRIGGER TIME	CONTADOR	ACCIONES
1	humedad	humedad	=	1	2	0	<span>✖</span> <span>OFF</span>

Figura 4.23: Sección de Alarmas con tabla

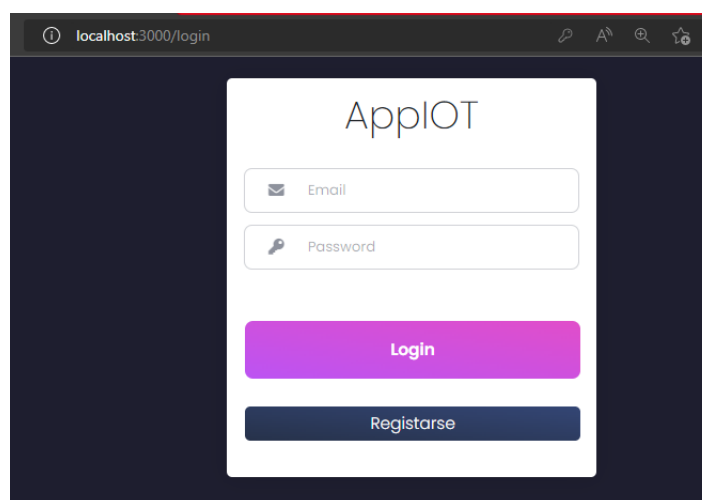
Como observamos la tabla es parecida a las demás, tenemos en cada fila las distintas alarmas con información y al final del todo un botón para borrar la alarma y otro botón para activarla o no. Si no la activamos no nos llegarán notificaciones.

Por último, decir que se ha incluido arriba a la derecha un selector para separar las alarmas dependiendo del dispositivo seleccionado, así se tendrá más control sobre estas. Si se quiere cambiar de dispositivo bastaría con seleccionar otro desde ese selector.

#### 4.5.2.6. Login

Para el login se ha optado por un diseño minimalista en el cual solo nos aparezca lo necesario que el estilo que se suele utilizar ahora en las interfaces de usuario.

También se ha incluido un botón que lleva al formulario para registrarse.



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/login'. The page has a dark blue background. In the center, there is a white rectangular box containing the 'AppIoT' logo at the top. Below the logo are two input fields: one for 'Email' with an envelope icon and one for 'Password' with a key icon. Underneath these fields are two buttons: a prominent purple 'Login' button and a smaller, dark blue 'Registarse' button.

Figura 4.24: Login



#### 4.5.2.7. Registro

Para el registro al igual que el login se ha escogido un diseño minimalista, en el cual solo tenga las entradas necesarias para el registro del usuario además de un botón que lleva al login si el usuario desea loguearse.

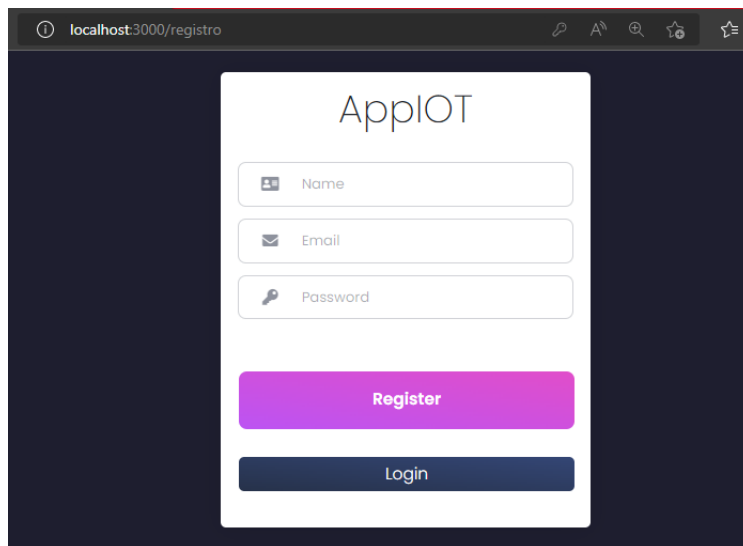


Figura 4.25: Registro

## 4.6. Estructura del Back-end

El back-end está compuesto por la API, el broker MQTT y la base de datos. El stack o pila, es un conjunto de herramientas usadas a la vez de estas tecnologías, la que hemos usado ha sido el Express-EMQX- Mongo. Aunque el broker MQTT se puede cambiar por otro, EMQX nos ofrece muchísimas funcionalidades que hemos mencionado antes y que no nos ofrece otros brokers de MQTT como es Mosquitto. Cada una de estas tres herramientas desarrolla una importante misión en nuestra plataforma, y aunque hay otras opciones estas son las que más se han podido adaptar a la plataforma objetivo.

### 4.6.1. Mongo

Para gestionar todos los datos hemos usado mongo, que es un gestor de base de datos NoSQL de tipo documental, es decir. los datos se guardan colecciones de documentos como hemos explicado en los capítulos anteriores. Es muy bueno a la hora de almacenar datos de forma rápida que es lo que a nosotros más nos interesa.

Mongo al ser software libre, hay multitud de módulos y de clientes para poder comunicarse con ella. En nuestro caso se ha usado la librería mongoose, que es una librería específica para realizar tareas con mongo en javascript. Para poder usarla se ha de importar al proyecto, en nuestro caso ya está añadida en el paquete de instalación, solo bastaría con importarla desde el archivo de javascript que vayamos a utilizar.

#### 4.6.1.1. Conexión

La conexión con la base de datos con mongoose es simple ya que requiere solo que hagamos un require de la librería. Después basta con llamar a la función connect de mongoose con la uri y con las opciones. Para una mayor comodidad se crean variables con estas opciones para así si hay que cambiarlas o volver a crear la conexión no se tengan que escribir de nuevo, abajo tenemos un ejemplo.

```
1 //Requires
2 const mongoose=require("mongoose");
3
4 //Variables Uri Mongo
5 const mongoUsername = "UserDev";
6 const mongoPassword = "PasswordDev";
7 const mongoHost = "localhost";
8 const mongoPort = "27017";
9 const mongoDatabase = "appIoT"
10
11 //Mongo URI
12 var uri="mongodb://" +mongoUsername+": "+mongoPassword+"@"+mongoHost+
13   ":"+mongoPort+"/ "+mongoDatabase;
14
15 //Opciones conexion
16 const opciones={
17   authSource: "admin"
18 };
19
20 //Conexion Moongoose
21 mongoose.connect(uri,opciones).then(=>{
22   console.log("+++++++".green);
23   console.log(" Conexion con Mongo Establecida".green);
24   console.log("++++++\n".green);
25 },(error)=>{
26   console.log("+++++++".red);
27   console.log(" Conexion con Mongo Fallida".red);
28   console.log("++++++\n".red);
29   console.log(error);
30 });
```

Código Fuente 4.6: Ejemplo de conexión con mongoose

Primero vamos con la uri, esta es la dirección con las que nos conectaremos a mongo, entre las opciones encontramos:

- **Username:** Es el nombre de usuario de la base de datos.
- **Password:** Es la contraseña del usuario de la base de datos.

- **Host:** Es la dirección del servidor de mongo.
- **Port:** Es el puerto al que se debe conectar. El puerto por defecto es el 27017.
- **Database:** El nombre de la base de datos.

Con estas opciones solo bastaría generar un string sustituyendo los datos de arriba en la cadena que vemos abajo:

*mongodb://Username:Password@Host:puerto/Database*

Ahora vamos con las opciones, para ello simplemente se genera un objeto javascript con las opciones que queramos, en este caso solamente con las características authSource con el valor de admin, para conectarnos como administradores.

Por último, ya solo queda llamar a la función connect con las dos variables que hemos dicho, uri y opciones. Al ser JavaScript dirigido por eventos podemos establecer varias cosas que sucedan si la respuesta es una cosa u otra, en este caso imprimimos que la conexión ha sido exitosa si ha salido todo bien o el error si ha salido más con el mensaje de que la conexión ha fallado.

#### 4.6.1.2. Modelo

Para poder manejar mejor los datos, se han creado modelos, que son esquema o Schema, de cómo serán los documentos de esa colección. En estos esquemas podemos definir los tipos de campos, si son obligatorios o incluso si son únicos.

Abajo ponemos un ejemplo de un modelo de nuestra base de datos:

```

1 import mongoose from "mongoose";
2 const validadorUnicoMongoose = require('mongoose-unique-validator')
  ;
3
4 const Schema = mongoose.Schema;
5
6 const plantillaSchema = new Schema({
7   userID: { type: String, required: [true] },
8   descripcion: {type: String},
9   plantillaNombre:{ type: String, required: [true], unique: true
10 },
11   fechaCreacion: {type: Number},
12   widgets: {type: Array}
13 })
14 //n 1.- validaci n
15 plantillaSchema.plugin(validadorUnicoMongoose, { message: 'Error,
16   la plantilla ya existe ya existe' })

```

```

17 //To modelo
18 const Plantilla = mongoose.model('Plantilla', plantillaSchema);
19
20 export default Plantilla;

```

Código Fuente 4.7: Modelo de datos de Plantilla

Vemos como la primera línea es un import de la librería (Línea 1 del código 4.7) y la segunda se le asigna a una variable una función (Línea 2 del código 4.7), esta función es la que en el apartado de validación (Línea 15 del código 4.7) valida que los valores designados como únicos se han únicos y que no exista un valor igual en la colección, es decir, con la variable `unique` sea `true`, en este caso solo lo es `nombre` (Línea 9 del código 4.7). Si ya existiera daría el mensaje entre comillas.

Después tenemos la llamada a la clase `Schema` en una constante (Línea 4 del código 4.7) y que abajo nos permitirá construir el objeto de esta clase. Seguidamente tenemos la construcción de la clase propiamente dicha (Línea 6-12 del código 4.7). Utiliza una notación clave-valor, decir de objeto de JavaScript. En este objeto, las claves son los nombres de los campos que tendrá nuestro documento en la colección de mongo y el valor es a su vez otro objeto javascript, que le asigna el tipo o `type` de la variable, así como otras características, como por ejemplo, `unique` si es único o `required` si es obligatorio que esté relleno.

Y por último tenemos la creación del modelo propiamente dicho para su uso en la aplicación (Línea 18 del código 4.7). El nombre de esta constante es se usará para llamar a las funciones que hace referencia a esa colección, por ejemplo *find*, para buscar. Si nos fijamos cuando se hace la asignación, se le pasa una cadena de caracteres que es el nombre que tendrá la colección en mongo y la variable con el objeto esquema que hemos creado antes (*plantillaSchema*, línea 6 del código 4.7).

Es importante no olvidarse al final de hacer la línea de `export default` (Línea 20 del código 4.7) con el nombre del modelo para que se exporte y podamos usarla en donde lo necesitemos.

Para usar el modelo simplemente basta con hacer un import en el archivo JavaScript donde lo vayamos usar. Un ejemplo sería el que vemos abajo (Código 4.8), donde importamos el modelo `Plantilla` que hemos ejemplificado arriba desde el archivo donde se encuentra el modelo.

```

1 //Modelos
2 import Plantilla from '../modelos/plantilla.js';

```

Código Fuente 4.8: Import del modelo de datos Plantilla

Una vez realizado este paso ya podemos hacer uso libremente del modelo en funciones como *find* para buscar un documento en base a uno o más de sus

campos en la colección o `create` para crear un nuevo documento en la colección. Abajo tenemos un ejemplo de `create` al cual se le pasa una variable que contiene un objeto de JavaScript con valores y los campos del documento a crear.

```
1 //Creamos la plantilla
2 const plantilla = await Plantilla.create( nuevaPlantilla );
```

Código Fuente 4.9: Ejemplo de uso del modelo con la función *create*

Otro ejemplo con *find*, por ejemplo, es el que vemos abajo donde se busca un documento cuyo campo *userID* sea igual a “213423432”.

```
1 //Búsqueda de plantillas
2 const plantillas = await Plantilla.find({ userID: "
  213423432" });
```

Código Fuente 4.10: Ejemplo de uso del modelo con la función *find*

Para más información se recomienda mirar la documentación de mongoose al respecto.[56]

#### 4.6.2. Broker EMQX

Para el broker hemos elegido EMQX, hemos comentado anteriormente por que lo hemos seleccionado, y es que no solo por su gran versatilidad, sino que también es altamente escalable gracias a la creación de clusters con nodos conectados entre sí, de los cuales hablaremos un poco más adelante.

Otra característica que ofrece EMQX es poder administrarla desde una interfaz gráfica, esta interfaz es accesible desde el puerto 18083 y puede ser desactivada si el usuario lo desea. Abajo podemos ver una foto de esta interfaz (Figura 4.26) y como vemos nos ofrece muchísima información desde datos analíticos hasta información de los clientes conectados a nuestro broker o los nodos disponibles. A la derecha tenemos distintos accesos a páginas que definen varias funciones.

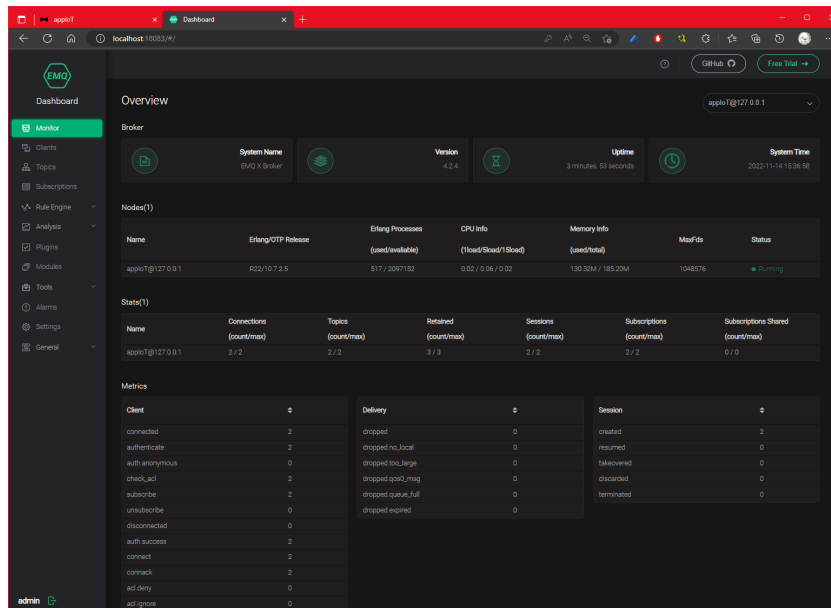


Figura 4.26: Pantalla principal del dashboard de EMQX

Navegando un poco encontramos pestañas donde podemos ver los clientes conectados a nuestro broker, por ejemplo.

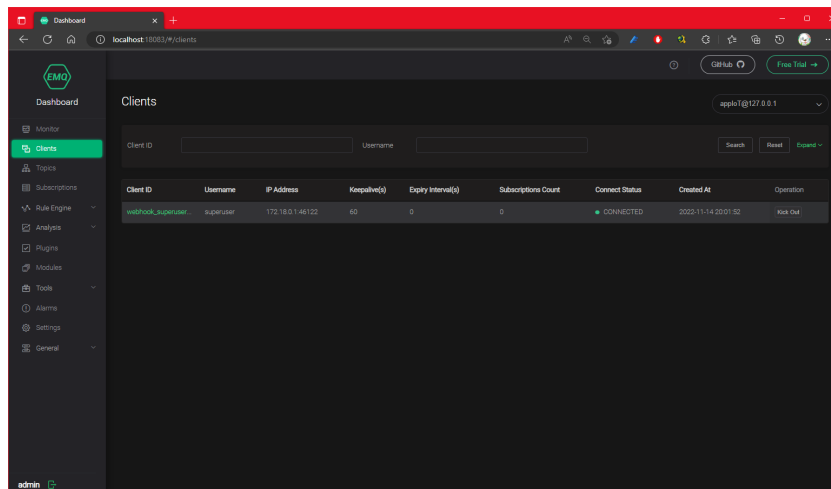


Figura 4.27: Pestaña de Clientes conectados

En esta otra, Modules podemos ver los módulos activos.

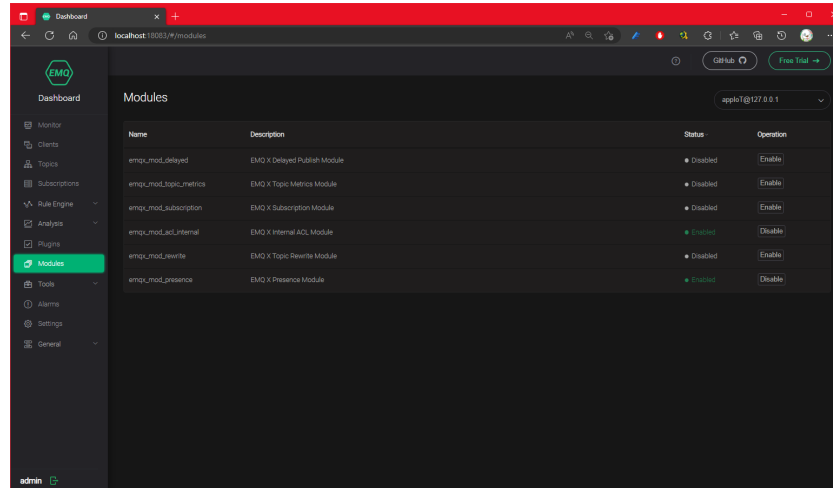


Figura 4.28: Pestaña de módulos activos

O en esta otra podemos ver los end-points de la API de EMQX. Esta API utiliza una autenticación, por lo que se ha creado un usuario específico para la API.

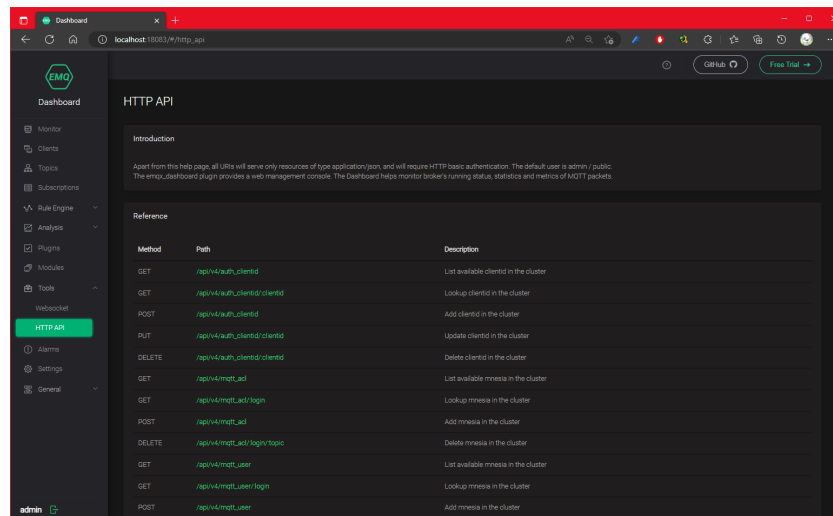


Figura 4.29: Pestaña de EMQX API HTML

Por último, fijarnos que nos ofrece su propio websocket para probar conexiones con la base de datos o el envío de mensajes.

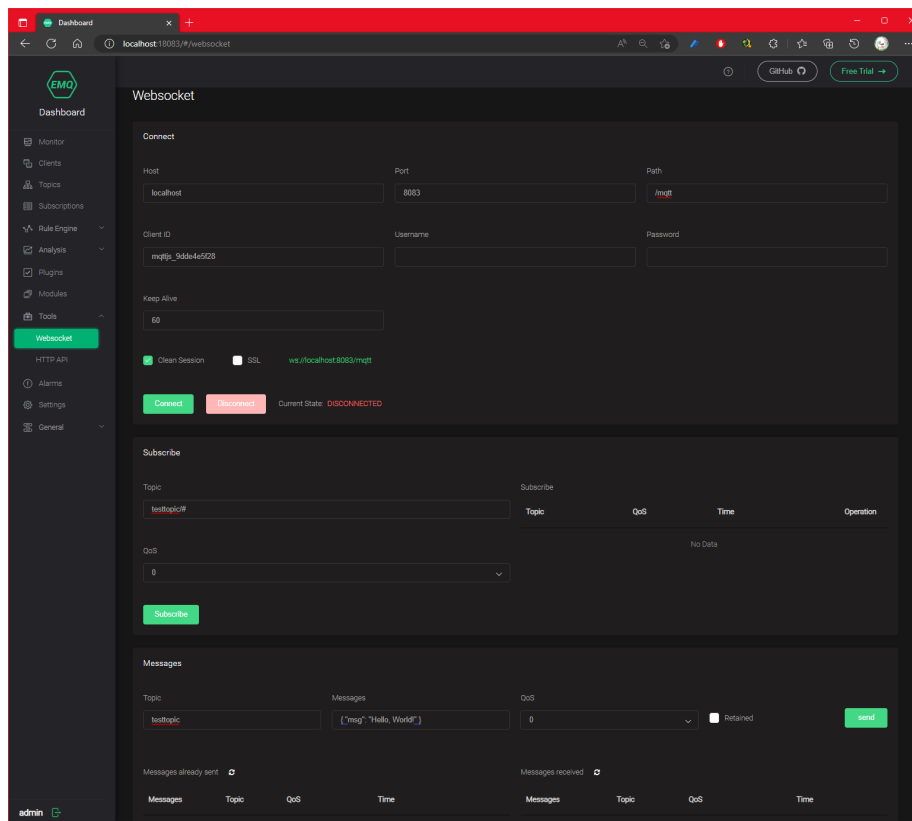


Figura 4.30: Pestaña del cliente Websocket

#### 4.6.2.1. Motor de Reglas

Parte de la causa de que hayamos cogido EMQX como broker es el Motor de Reglas o Rule Engine, esto nos ha permitido desarrollar las funcionalidades de Alarmas y Notificaciones, así como la funcionalidad de guardar o no los datos en la base de datos. Este motor de Reglas se divide en dos componentes, Resources y las reglas.

El primer componente es el Resource o Recurso y establece la conexión entre el exterior del broker y la regla. Hay de varios tipos, por ejemplo, el de *web hook* o gancho web, pero hay más tipos como de *EMQX Bridge*, que sirve para crear una conexión puente con otro broker.



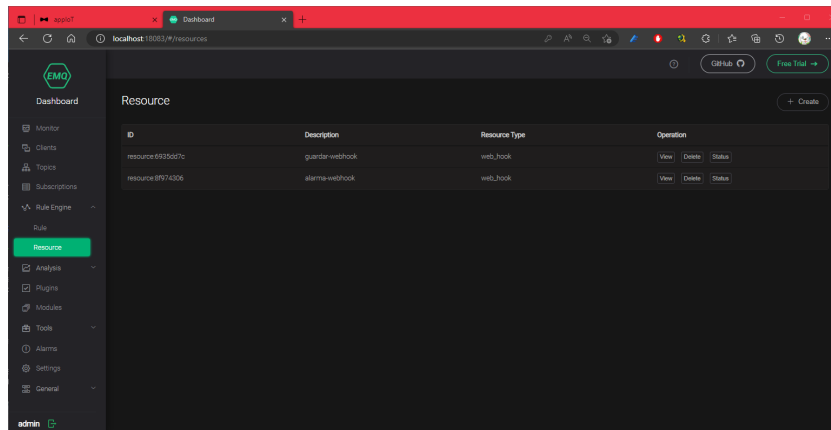


Figura 4.31: Pestaña Resources EMQX

Los datos básicos que debe tener un recurso dependen del tipo, el que se ha usado para el presente proyecto es el de WebHook, que nos hará de gancho para filtrar los mensajes que reciba el broker. Entre los datos importantes que encontramos en este tipo de recurso son:

- **Request URL:** Es la url donde impactará el gancho cuando se active.
- **ResourceID:** Es el nombre único que recibe nuestro recurso y es aleatorio.
- **Descripción:** Pequeña descripción que queramos ponerle.

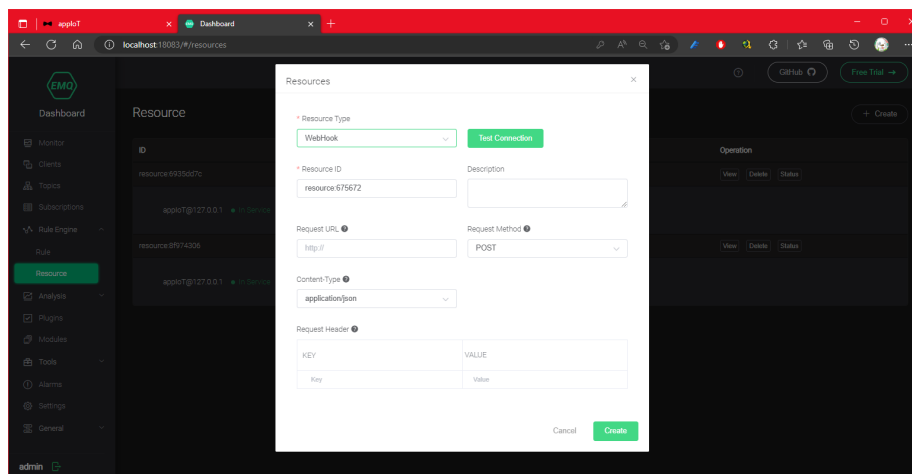


Figura 4.32: Ventana emergente de creación de un recurso

Nosotros tenemos dos recursos para el funcionamiento de la plataforma, el recurso de guardado, que como indica que gestiona el guardado los datos y por otro lado tenemos el recurso de alarma, que se encarga de gestionar cuando las alarmas se activan.

Para crearlos usamos la API que nos ofrece EMQX, para ello hemos creado un usuario especial para conectarnos desde nuestra API como hemos dicho antes y se ha generado en el objeto de JavaScript:

```
1 //Autenticacion para la API
2 const auth = {
3   auth: {
4     username: 'admin',
5     password: 'emqxsecret'
6   }
7 }
```

Código Fuente 4.11: Ejemplo de Objeto Auth

Se recomienda cambiar la clave ya que es información sensible (se puede cambiar desde la pestaña de usuarios en el interfaz gráfica de EMQX).

Los ID de los recursos se han guardado en variables globales en nuestra API para así poder usarlas en otra parte de esta. Cuando se ejecuta la API se ejecuta la función *listarRecursos* del archivo *emqxApi.js* que encontramos dentro de nuestra API. Esta función comprueba que estos dos recursos están creados, si no, los crea. Un ejemplo de configuración es el que vemos abajo (Código 4.12, es la configuración del recurso de guardar.

```
1 const recursoGuardado = {
2   "type": "web_hook",
3   "config": {
4     url: "http://localhost:3001/api/guardar-webhook",
5     headers: {
6       token : "121212" //Para confirmar que lo
7       enviamos nosotros, puede ser otro cadena
8     },
9     method: "POST"
10   },
11   description: "guardar-webhook"
12 }
```

Código Fuente 4.12: Ejemplo de datos Recurso

Como podemos ver en la configuración hemos añadido características extras, como un header o cabecera, la cual contiene un objeto con un solo par de clave-valor (Líneas 5-7 del código 4.12). Esta clave-valor hace referencia a un token que es usado como seguridad para confirmar cuando nos llegan datos a la API lo está haciendo el recurso correcto y no uno fraudulento. La url (Línea 4 del código 4.12) que aparece es lo que hemos dicho antes como Request URL y es

donde enviará los datos.

Si estos recursos quedan fuera de servicio, las reglas asociadas a estos no funcionarían, por lo que si esto sucede, se tendrían que levantar manualmente desde la pestaña *Resource*.

El otro componente del Motor de Reglas son las propias reglas. Estas reglas son condiciones que si se cumplen, realizan acciones específicas. Estas reglas utilizan SQL para funcionar y utilizan el payload del mensaje MQTT para obtener información.

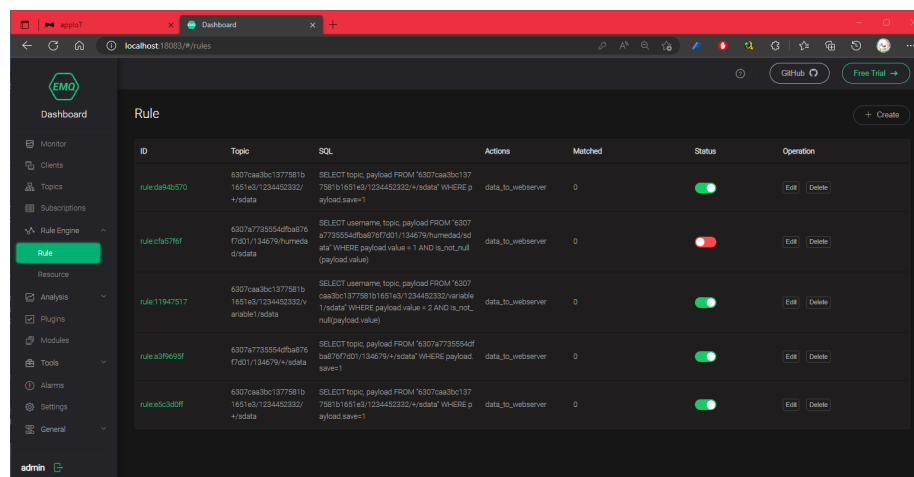


Figura 4.33: Pestaña de Reglas EMQX

Por ejemplo, nosotros tenemos una regla por cada dispositivo, esta regla se encarga de saber si se guarda o no el mensaje, para ello tiene que revisar si en el mensaje dentro del payload, la variable *save* tiene un valor 1, si es así impactará sobre el end-point del recurso que hemos creado y lo guardará en la base de datos, si por el contrario es un 0 o está apagada la regla no impactará sobre el end-point del recurso, y no lo guardada en el la base de datos. Abajo tenemos un ejemplo de cómo se crea una regla(Código 4.13) desde nuestra API utilizando la API de EMQX, recordemos que necesitamos la autenticación para acceder a al API de EMQX (Código 4.11).

```
1 //Url de la api
2 const url = "http://localhost:8085/api/v4/rules";
3
4 //Contruccion del topic
5 const topic= userID + "/" + dID + "/+/sdata";
6
7 //Sentencia SQL
```

```

8      const rawsql= 'SELECT topic, payload FROM "' +topic+' " WHERE
      payload.save=1';
9
10     //Objeto nueva regla
11     var nuevaRegla = {
12         rawsql: rawsql,
13         actions: [{
14             name: "data_to_webserver",
15             params:{
16                 $resource: global.recursoGuardado.id,
17                 payload_tmpl: '{"userID":"' +userID+' ","payload
18                 ":${payload},"topic":"${topic}"}'
19             }
20         }],
21         description: "GUARDAR - REGLA",
22         enabled: status
23     };
24
25     //Llamada a la API para guardar la regla
26     const respuesta =await axios.post(url, nuevaRegla, auth)

```

Código Fuente 4.13: Ejemplo de creación de una regla

Vemos como para crear la regla necesitamos varias características entre ellas encontramos:

- **Rawsql:** Es la sentencia SQL de la regla y es la sentencia condicional.
- **Action:** Es un array de objeto JSON, ya que una regla puede hacer más de una acción, he incluye algunas características:
  - **Name:** Es el tipo de acción.
  - **Params:** Objeto JSON, que contiene el recurso de la regla y el *payload\_tmpl* que es una plantilla de lo que contendrá el payload cuando llegue a la regla.
- **Description:** Una breve descripción de la regla.
- **Enabled:** True si la regla estará activa, false si no.

Para la regla de alarmas es igual sin embargo la sentencia SQL y el payload serán más largos ya que contendrán más datos.

#### 4.6.2.2. Autenticación

Otra de la cualidades de EMQX es que posee módulos adicionales o plugins que amplían sus funcionalidades, para el proyecto se activan los siguiente:

- **emqx\_recon:** Sirve para el debug en EMQX.
- **emqx\_retainer:** Se encarga de almacenar los mensajes MQTT retenidos, por ejemplo, los que están en cola.
- **emqx\_management:** Administración sobre la API de EMQX.

- `emqx_dashboard`: Habilidad de WebDasboad.
- `emqx_auth_mongo`: Autenticación con MongoDB

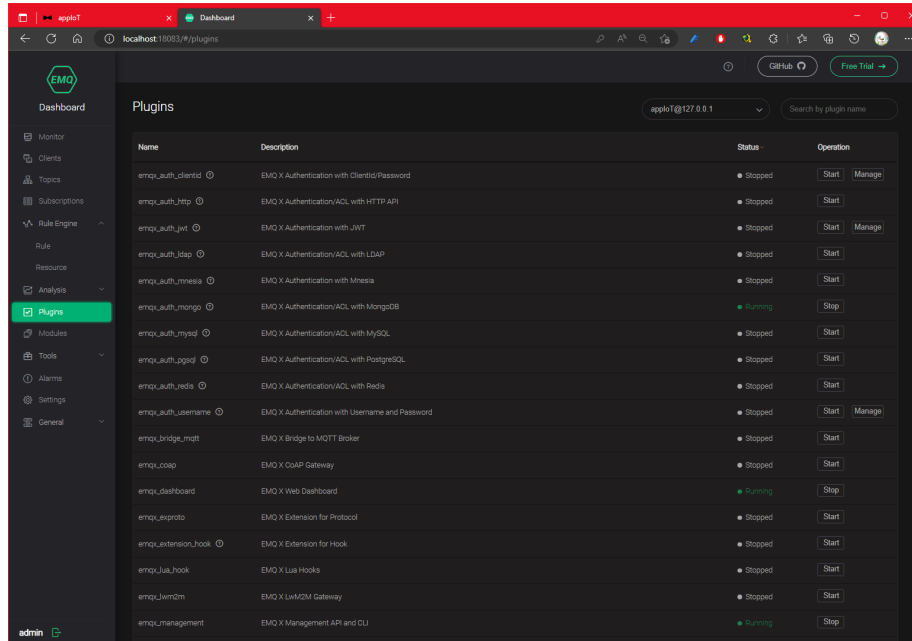


Figura 4.34: Pestaña de Plugins EMQX

En la documentación de sobre los plugins disponibles se recomienda la lectura de la documentación de EMQX sobre los módulos[14].

Para la autenticación nos tenemos que fijar en el módulo `emqx_auth_mongo` ya que este nos permite tener una colección específica en esta tener un control sobre los usuarios permitidos para conectarse a la plataforma. Solucionando así el problema que tiene MQTT con la seguridad.

La colección nosotros la hemos llamado `emqxauthrules`, pero puedes llamarla como quieras para ello simplemente tienes que abrir el archivo de configuración `etc/emqx_auth_mongo.conf` y configurarlo con los datos necesarios, este caso se han rellenado:

- `auth.mongo.type`: Con el valor de `single`.
- `auth.mongo.server`: Con el valor de `IP:27017` que es el puerto de mongo.
- `auth.mongo.login`: Con el usuario de acceso a mongo.

- `auth.mongo.password`: Con la contraseña del usuario para acceder a mongo.
- `auth.mongo.auth_source`: Con admin para tener permisos de administradores.
- `auth.mongo.database`: Con el nombre de la base de datos.
- `auth.mongo.auth_query.collection`: Con el nombre de la colección en la base de datos, `emqxauthrules` para el proyecto.
- `auth.mongo.auth_query.password_field`: El campo de contraseña en la colección.
- `auth.mongo.super_query`: Con off ya que no queremos activar el superuser query.
- `auth.mongo.super_query.collection`: Con el nombre de la colección en la base de datos, `emqxauthrules` para el proyecto.
- `auth.mongo.acl_query.collection`: Con el nombre de la colección en la base de datos, `emqxauthrules` para el proyecto.

Con ello ya solo bastaría de añadir los datos en la colección *emqxauthrules* para que se pueda conectar, además de añadir el usuario y la contraseña para el acceso, se añaden más campos como el *publish* y *subscribe*, los cuales restringen los topic a los que pueden publicar y suscribirse respectivamente para que un usuario no pueda recibir o enviar datos a otro usuario de forma fortuita o malintencionada.

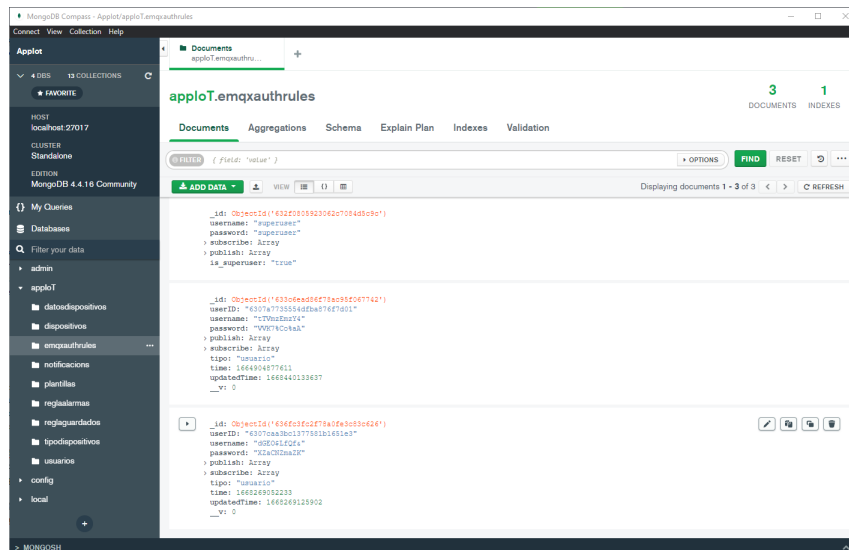


Figura 4.35: Datos de usuarios en la colección *emqxauthrules* en MongoDB

Para generar estos datos se han creado dos end-points específicos para el manejo de estos datos. Con ello nos aseguramos que la API es la que controla el acceso y verificamos que solo los usuarios registrados obtienen credenciales y se pueden subscribir solo a lo que le está permitido. Como medida extra de seguridad, estas claves cambian después de su uso por lo que si alguien obtiene las claves solo podrá usarlas si el usuario no se ha metido aún o si no pide otras nuevas.

Los end-points son *api/getMqttcredenciales* y *api/getMqttcredencialesReconnection* y se encuentran en el archivo de usuarios.js de la API. Abajo podemos ver un ejemplo (Código 4.14) cuando se crea un usuario lo que se genera en la API, siendo *userID* el ID único del usuario, por lo que vemos que es totalmente personalizado.

```
1 const nuevasCredenciales ={
2     userID: userID,
3     username: realizarID(10),
4     password: realizarID(10),
5     publish: [userID + "/"#"],
6     subscribe: [userID + "/"#"],
7     tipo: "usuario",
8     time: Date.now(),
9     updateTime: Date.now()
10 };
```

Código Fuente 4.14: Ejemplo de datos de un usuario para MQTT

Podemos ver más información en la documentación del módulo[57].

#### 4.6.2.3. Cluster

Uno de los requisitos es que la plataforma es que sea escalable por ello se escogió EMQX como broker ya que es uno de los que mejor escalan entre los brokers estudiados. Esto es gracias a que tiene implementado un sistema de clustering sobre una estructura de red TCP que proporciona el lenguaje en el que está escrito Erlan/OTP[14].

Para lograrlo, EMQX utiliza nodos, es decir cada broker EMQX es un nodo independiente hasta que lo conectas con otro nodo, en ese momento se orquestan y funcionan como un cluster. Puedes añadir a la red tantos nodos como quieras. Para hacernos unas ideas tenemos el esquema de abajo.

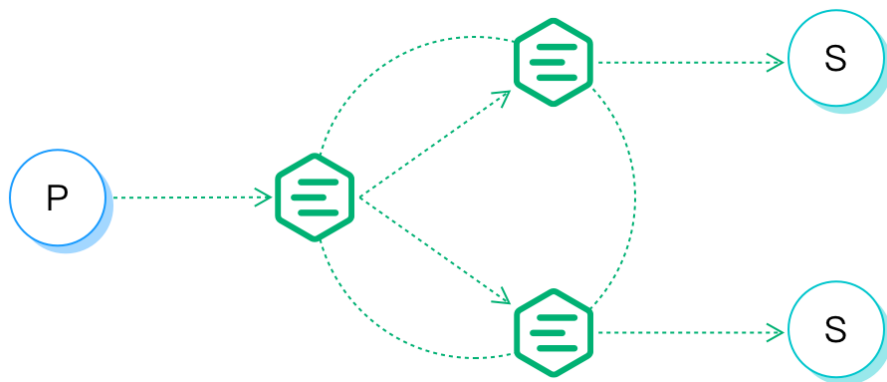


Figura 4.36: Esquema de un cluster EMQX [14]

En la Figura 4.36 también podemos apreciar como al publicarlo un cliente (Círculo con la letra P) el nodo (los hexágonos) que recibe el mensaje lo transmite hacia los otros nodos, y estos a su vez lo envían a los clientes que se han suscrito (Círculo con la letra S) al topic de la publicación.

Esto es gracias a que todos tienen en común una tabla de ruta, la cual contiene la ruta de los topic existentes además de un árbol de tópicos, por lo que lo único que tiene que hacer es enviarla a los nodos que están en la ruta por el árbol de tópicos. Además, en cada broker de forma separada también existe una tabla donde se actualizan las suscripciones que se han hecho en ese nodo.

Para generar un cluster, lo primero que hay que generar es un nombre único para cada nodo, para ello hay que ir al archivo *emqx/etc/emqx.conf* y configurar la directiva *node.name* por un nombre del tipo *nombre@host*. Por ejemplo *node1@192.168.0.12*.

Lo segundo es generar cookie común para todos los nodos del cluster ya que así es como identifica EMQX que los otros nodos son del mismo cluster. La cookie es un conjunto de caracteres, por ejemplo, *cookieSuperSegura*. Esta cookie la asignaremos a la directiva *node.cookie* del archivo *emqx/etc/emqx.conf*.

Tercero cambiaremos la forma de descubrir nodos del cluster a manual, para ello nos dirigimos al archivo *emqx/etc/emqx.conf* a la directiva *cluster.discovery* y le asignaremos el valor manual. Aunque hay otras formas, como la static que se conecta solo a los nodos que hay en la directiva *cluster.static.seeds*.

Estos pasos hay que repetirlos en cada nodo para prepararlos, lo único que cambia es en primer paso el nombre que se le da a cada nodo, que deberá ser distinto.



Ahora que está todo listo, el último paso es unirlos, para ello deberemos ejecutar en los nodos el programa `/bin/emqx_ctl cluster join` seguido del nombre de nodo con el que queremos formar un cluster. El nombre del nodo es el que hemos escrito en el paso uno.

En un cluster se pueden unir todos los nodos que se necesite en cualquier momento, así como quitar nodos con por ejemplo las sentencias *leave* o *remove* del programa *emqx\_ctl*.

Si se necesita más información se puede consultar la información de EMQX en el apartado de Cluster[58].

#### 4.6.2.4. Topics

Para aunar las comunicaciones, se ha establecido unas normas a la hora del envío de mensajes, estas normas enmarcan el uso de los topics así como el contenido del payload.

Primeramente vamos a ver los topics, los cuales llevan siempre la misma estructura, lo que cambia es el final que detalla la finalidad del mensaje, la parte comun se compone de los siguiente elementos

- **User ID:** Es el ID del usuario, concretamente el generado por mongo cuando se crea el usuario, así salvaguardamos si privacidad
- **Dispostivo ID:** Es el ID del dispositivo, que es el que se introduce cuando se crea por parte del usuario y identifica al dispositivo individualmente.
- **Variable:** Es el nombre de la variable que representa al sensor del dispositivo, este nombre es introducido en la creación del tipo de dispositivo.

La parte variable se establecen las siguientes opciones:

- **actdata:** Usada para los mensajes donde se actualiza el valor de una variable al dispositivo, por ejemplo, el envío de *true* o *false* para encender un interruptor. Normalmente lo envía el usuario al dispositivo.
- **sdata:** Se usa para los mensajes que el dispositivo envía a la plataforma. Por ejemplo, cuando el envío de la lectura de un sensor de temperatura. Normalmente lo envía el dispositivo a la plataforma.
- **saverdata:** Reservada para el envío de mensajes que se quiera guardar en la base de datos. En la actualidad no se hace uso ya que se usa un mecanismo diferente con los recursos de EMQX, pero esta previsto aquí por si se necesita cambiar de broker u otro motivo.

Por lo que los tres posibles topic quedarían así:

- *UserID/DispositivoID/variable/actdata*
- *UserID/DispositivoID/variable/sdata*
- *UserID/DispositivoID/variable/saverdata*

Por ultimo, los mensajes deben incluir en el *payload* dos campos:

- **value:** Es el valor de la variable que se envía.
- **save:** Booleano, que es usado por los recursos y las reglas para saber si un mensaje tiene que ser guardado o no en la base de datos. El valor *true* o 1 significa que se guarda y si es *false* o 0 no se guardará.

#### 4.6.3. JWT

Para la autenticación en el front-end, pese a la multitud de sistemas que hay actualmente, se optado por usar JWT, que es un protocolo sencillo y muy potente y que para mayor seguridad se ha optado por hacer que el token tuviera una validez menor.

Para un proyecto de este tipo donde la comunicación tiene que ser rápida y eficiente, JWT es perfecto por lo dicho anteriormente. Hay otras opciones como son Oauth por ejemplo, pero son más costosas de implementar, por lo que se ha dejado como una opción a mejorar si el proyecto crece lo suficiente.

Se ha usado la implantación de un middleware para que esté entre la API y el front-end, para así verificar que quien llama a la API es un usuario lícito. Este componente se ubica en la carpeta de *middleware* en el archivo *autenticadortoken.js*.

Para incorporarlo a los archivos de nuestra API basta con hacer un *require* del componente (Código 4.15).

```
1 const {checkAuth} = require('../middlewares/autenticadortoken.js')
  );
```

Código Fuente 4.15: Requiere del middleware

Y se coloca en medio de la función del endpoint, como por ejemplo:

```
1 router.post('/reglaAlarma', checkAuth, async (req,res) => {})
```

Código Fuente 4.16: Ejemplo de utilizacion del middleware de autenticación

Este token se genera en la API cuando se hace un login correcto, es decir, cuando el usuario y la contraseña introducida coincide con la que hay en la base de datos. Cuando esto ocurre el token generado se envía al front-end o quien haya hecho la petición de login en la respuesta. En el caso del front-end de la

plataforma lo guarda en el local store del navegador y en el store de nuxt para poder usarlo en las consultas además de loguearse automáticamente si no se ha hecho un log out o no ha caducado el token.

#### 4.6.4. API

Para poder controlar el acceso a los datos de la base de datos por parte del exterior se ha visto requerido de desarrollar una API. Además de hacer de filtro de acceso la API cumple otros requisitos, pero el más relevante es el de capa de abstracción respecto al front-end y obtener modularidad.

Como su objetivo principal es el de hacer de intermediario entre la base de datos y el exterior se optó por dividir la API en secciones dependientes de los modelos, que coinciden con los modelos presentados en la base de datos en anteriores capítulos que son:

- Dispositivos (dispositivo)
- Plantillas (plantilla)
- Usuarios (usuarios)
- Alarmas (reglaAlarma)
- Tipos de Dispositivos (tipodDispositivo)
- Regla Guardado
- Notificaciones

Cada uno lleva su propia uri para usarse, en este caso `<IP del servidor>/api/<nombre del modelo entre paréntesis>`. Para acercarse lo más posible al modelo REST, se han creado con los distintos métodos en vez de crearlas todas con el método POST. Esto también ha ayudado a reducir la cantidad de uris que se van usar ya que se selecciona el método de encabezado que se va a usar en cada caso como por ejemplo DELETE para borrar o GET para obtener datos.

Hay otros endpoint que no entran dentro de esa estandarización con el nombre de modelo en la uri, como es el endpoint que provee a las gráficas, el que comunica las credenciales de MQTT para conectarse al broker o el login.

Todos actualmente excepto los endpoints de login, registro, alarma-webhook y guardar-webhook requieren un token de JWT por el header para su autenticación y verificar así que se hace de manera lícita y que no se borran datos de otros usuarios ya que en el token se incluyen el id del usuario.

Para llevar a cabo la API se ha utilizado el framework *Express*. Express se ejecuta sobre node.js y está escrito en JavaScript, se usa en la creación de APIS

y aplicaciones cliente-servidor. Es uno de los clientes más populares a la hora de desarrollar APIs ya que es OpenSource y muy apoyado por la comunidad.

Para utilizarlo es muy simple, solo hay que hacer un *require* y después crear una instancia de él (Código 4.17).

```
1 // requires
2 const express = require('express');
3
4 //instancias
5 const appExpress = express();
```

Código Fuente 4.17: Ejemplo de crear una instancia de Express

Con ello ya lo tenemos instanciado (Código 4.17), podemos configurarlo de distintas maneras. Para la plataforma se ha incluido la siguiente configuración:

```
1 //Configuraci n Express
2 appExpress.use(morgan("tiny"));
3 appExpress.use(cors());
4 appExpress.use(express.json());
5 //Para poder pasar varaibles por url
6 appExpress.use(express.urlencoded({
7   extended: true
8 }));
```

Código Fuente 4.18: Configuración Express

La primera configuración se refiere al uso de Morgan ( Línea 2 del código 4.18 ), que es un middleware para mostrar los request junto a información extra, hay varias configuraciones, la *tiny* es la más escueta de todas. La segunda configuración es cors ( Línea 3 del código 4.18 ) para controlar y permitir las peticiones asíncronas con diferente origen, cors es el acrónimo de *Cross-Origin Resource Sharing*. La tercera es el middleware para cargar la información con respuestas en JSON ( Línea 4 del código 4.18 ). Y la última configuración habilita el paso de variables a través de la url( Línea 6 del código 4.18 ). Tanto la primera( Línea 2 del código 4.18 ) como la segunda( Línea 3 del código 4.18 ) necesitan hacer sus *requires* correspondientes para poder utilizarlas.

Después debemos añadir las rutas de los end-point, si las tenemos en distintos archivos, se debe de usar *instanciaExpress('/Pre Dominio', require('Ruta del Archivo'))*. Por ejemplo, si queremos exportar las rutas que hay en el archivo *plantillas.js* (Código 4.19).

```
1 appExpress.use('/api', require('./rutas/plantillas.js'))
```

Código Fuente 4.19: Exportación de la ruta del archivo *plantillas.js*

Después haremos un export de los módulos con *module.exports=instanciaExpress* (Código 4.20).

```
1 module.exports=appExpress;
```

Código Fuente 4.20: Exportación del modulo

Por último pondremos a la instancia a escuchar por un puerto las peticiones con *instanciaExpress.listen(3001, () => {})*. Por ejemplo, si queremos escuchar por el puerto 3001:

```
1 //listener
2 appExpress.listen(3001,() =>{
3   console.log("Escucho API por el 3001");
4 });
```

Código Fuente 4.21: Ejemplo de escucha por el puerto 3001

Y listo, ya tendremos express escuchando por el puesto solicitado y preparado para responder a lo que pida.

Para crear rutas dentro de los archivos que hemos mencionado antes se debe de hacer de la siguiente manera. Primero ponernos los *requires* necesarios, estos son *express* y *router* (Código 4.22), que es el que almacena las rutas en *express*.

```
1 // requires
2 const express = require('express');
3 const router = express.Router();
```

Código Fuente 4.22: Requiere para el uso en archivos de rutas

Después ponemos el end-point propiamente dicho, para ello usamos el *require* de router seguido del protocolo de la petición (POST, GET, DELETE, PUT, ...) y los argumentos de esta que contienen la ruta y la función a realizar cuando se activa el end-point. Por ejemplo, una petición POST para guardar una plantilla (Código 4.23), ya que POST se utiliza para enviar datos del cliente al servidor.

```
1 router.post("/plantilla", checkAuth, async (req,res) =>{ })
```

Código Fuente 4.23: Ejemplo de un end-point

Como podemos ver dentro del paréntesis después del POST (Código 4.23), ponemos la cadena de caracteres que es la ruta (recordemos cuando la usamos tenemos que ponerla completa, incluido lo que pusimos en *use* cuando la exportamos arriba, */api/plantilla* en este caso), seguido del middleware que creamos para la autenticación y que explicamos antes, seguido de la función flecha. En este caso la función flecha lleva la palabra reservada *async* porque es un método asíncrono. Los argumentos de la función flecha son el *request* o *req* y el *response* o *res*. En los puntos suspensivos dentro de los corchetes iría el contenido de la función.

Al final del documento, cuando hayamos escrito las funciones y las rutas necesarias es imprescindible realizar un *export* de la variable *router* ya que así

lo estaremos exportando a express. Se utiliza la sentencia *module.exports = router*, aunque *router* puede variar de nombre dependiendo del que le hayas puesto cuando lo exportaste al principio.

Los endpoints utilizados son:

#### 4.6.4.1. Usuarios

##### Registro - POST

`api/registro`

Se utiliza para registrar un usuario a través del método post, los datos requeridos se pasan a través de objeto JSON en el body. El objeto debería tener el siguiente aspecto:

```
1 {  
2   "nombre": "testname",  
3   "password": "13215",  
4   "email": "prueba@test.com"  
5 }
```

Código Fuente 4.24: Objeto JSON enviado por el Body

Si todo sale bien obtendremos esto:

```
1 {  
2   "status": "succes",  
3 }
```

Código Fuente 4.25: Respuesta correcta

##### Login - POST

`api/login`

Con él se obtiene el token de logeo en el sistema y que es necesario para utilizar todas las funcionalidades de la API excepto login y registro.

El objeto que se debe suministrar a través de body como parámetro del método Post tiene que seguir el siguiente formato:

```
1 {  
2   "password": "13215",  
3   "email": "prueba@test.com"  
4 }
```

Código Fuente 4.26: Objeto JSON enviado por el Body

La respuesta será un token dentro de un objeto JSON que se verá tal que así:

```
1 {
2   "status": "success",
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRvc1VzdWFyaW8iOnsiaX2lkIjoiaWJmN2NhYTNiYzEzZmVzIj0iOiJ1bWUzIiwibm9tYnJlIjoiaWoidGVzdG5hbWV1X1c6NmNtTOXT_3lJD30TIdGUKwHuPJzjTm6vrJMGH-WVo",
4   "datosUsuarios": {
5     "_id": "6307caa3bc1377681b1681e3",
6     "nombre": "testname",
7     "--v": 0
8   }
9 }
```

Código Fuente 4.27: Ejemplo de Respuesta

## Mqtt Credenciales - POST

```
api/getMqttcredenciales
```

Petición usada a través del método POST para la obtención de claves para la reconexión del cliente MQTT con el broker y que corresponden con un usuario encontrado cuyo id que está guardado en el token. Por lo que hay que suministrarle en el header el token de autenticación tal que así:

token:UnTokenValido

La respuesta será un objeto en formato JSON contenido por otros objetos en el mismo formato con los datos de cada plantilla. Un ejemplo puede ser:

```
1 {
2     "status": "success",
3     "username": "oalv&GBTnb",
4     "password": "pwP$TKGA2p"
5 }
```

Código Fuente 4.28: Ejemplo de Respuesta

## Mqtt Credenciales Reconnect - POST

```
api/getMqttcredencialesReconexion
```

Petición usada a través del método POST para la obtención de claves para la conexión del cliente MQTT con el broker y que corresponden con un usuario encontrado cuyo id que está guardado en el token. Por lo que hay que suministrarle en el header el token de autenticación tal que así:

token:UnTokenValido

La respuesta será un objeto en formato JSON contenido por otros objetos en el mismo formato con los datos de cada plantilla. Un ejemplo puede ser:

```
1 {
2     "status": "success",
```

```

3  "username": "v8 JIKY2W%",
4  "password": "D2oM4%kLKc"
5  }

```

Código Fuente 4.29: Ejemplo de Respuesta

#### 4.6.4.2. Plantillas

##### Creación - POST

/api/plantilla

Esta petición se hace a través del método POST para la creación de un objeto del tipo plantilla. Se solicita mediante el body un objeto en formato JSON con estos parámetros:

```

1  {
2    "plantilla": {
3      "descripcion": "abababa",
4      "plantillaNombre": "Nueva plantilla",
5      "templateID": "165asd"
6    }
7  }

```

Código Fuente 4.30: Objeto JSON enviado por el Body

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```

1  {
2    "status": "succes",
3  }

```

Código Fuente 4.31: Respuesta correcta

##### Listado - GET

/api/plantilla

Petición usada a través del método GET para la obtención de una lista de plantilla que corresponden con el id del usuario que está guardado en el token. Por lo que hay que suministrarle en el header el token de autenticación tal que así:

token:UnTokenValido

La respuesta será un objeto en formato JSON contenido por otros objetos en el mismo formato con los datos de cada plantilla. Un ejemplo puede ser:



```

1 {
2   "status": "success",
3   "data": [
4     {
5       "_id": "6314dff05119a8a4b0e89107",
6       "userID": "6307caa3bc137581b1651e3",
7       "descripcion": "abababa",
8       "plantillaNombre": "Nueva plantilla 3",
9       "fechaCreacion": 1662312432375,
10      "widgets": [].
11      "__v": 0
12    }
13  ]
14 }

```

Código Fuente 4.32: Ejemplo de Respuesta

### Borrado - DELETE

/api/plantilla?plantillaID=

En esta petición se usa para borrar una plantilla a través del método delete y se utiliza el ID de la plantilla en el query o directamente en la uri para identificar el objeto y borrarlo, el parámetro es plantillaID. Por ejemplo:

api/plantilla?plantillaID=630cda3f0e613edfce241cf3

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```

1 {
2   "status": "success",
3   "datos": {
4     "acknowledged": true,
5     "deletedCount": 1
6   }
7 }

```

Código Fuente 4.33: Ejemplo de Respuesta

El apartado deletedCount hace referencia a los elementos borrados.

#### 4.6.4.3. Dispositivos

##### Creación – POST

/api/dispositivo

Esta petición se hace a través del método POST para la creación de un objeto del tipo dispositivo. Se solicita mediante el body un objeto en formato

JSON con estos parámetros:

```
1 {
2   "nuevoDispositivo": {
3     "userID": "abababa",
4     "dID": 1234452332,
5     "nombre": "UnNombre4",
6     "plantillaNombre": "templateprub",
7     "plantillaID": "165asd",
8     "tipoDispositivoID": "636fc8182f78a0fe3c83c643",
9     "tipoDispositivoNombre": "Placa6"
10  }
11 }
```

Código Fuente 4.34: Objeto JSON enviado por el Body

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```
1 {
2   "status": "succes",
3 }
```

Código Fuente 4.35: Respuesta correcta

### Listado - GET /api/dispositivo

Petición usada a través del método GET para la obtención de una lista de dispositivos que corresponden con el id del usuario que está guardado en el token. Por lo que hay que suministrarle en el header el token de autenticación tal que así:

token:UnTokenValido

La respuesta será un objeto en formato JSON contenido por otros objetos en el mismo formato con los datos de cada dispositivo. Un ejemplo puede ser:

```
1 {
2   "status": "success",
3   "data": [
4     {
5       "_id": "636fc8582f78a0fe3c83c64b",
6       "userID": "6307caa3bc1377581b1651e3",
7       "dID": "1234452332",
8       "nombre": "UnNombre4",
9       "seleccionado": true,
10      "plantillaID": "165asd",

```

```

11  "plantillas": [],
12  "plantillaNombre": "templateprub",
13  "tipoDispositivoID": "636fc8182f78a0fe3c83c643",
14  "tipoDispositivoNombre": "Placa6",
15  "fechaCreacion": 1668270168428,
16  "--v": 0,
17  "reglaGuardado": {
18    "_id": "636fc8582f78a0fe3c83c648",
19    "userID": "6307caa3bc1377581b1651e3",
20    "dID": "1234452332",
21    "emqxReglaID": "rule:138f29da",
22    "status": false,
23    "--v": 0
24  },
25  "alarmas": [],
26  "tipoDispositivo": {
27    "_id": "636fc8182f78a0fe3c83c643",
28    "nombre": "Placa6",
29    "tipoDispositivo": "diposivoPTueb",
30    "variables": [
31      {
32        "nombre": "variable1",
33        "tipo": "Number",
34        "tambBuffer": "1",
35        "compWidgets": [
36          "IotBoton",
37          "IotSwitch",
38          "IotIndicador",
39          "IoTGraficaNum"
40        ]
41      },
42      {
43        "nombre": "humedad",
44        "tipo": "other",
45        "tambBuffer": "1",
46        "compWidgets": [
47          "IotBoton",
48          "IotSwitch",
49          "IotIndicador",
50          "IoTGraficaNum"
51        ]
52      }
53    ],
54    "fechaCreacion": 1668270104186,
55    "--v": 0
56  }
57 }
58 ]
59 }

```

Código Fuente 4.36: Ejemplo de Respuesta

### Borrado - DELETE

/api/dispositivo?dispID=

En esta petición se usa para borrar un dispositivo a través del método delete y se utiliza el ID del dispositivo en el query o directamente en la uri para

identificar el objeto y borrarlo, el parámetro es dispID. Por ejemplo:

`/api/dispositivo?dispID=1234452332`

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

`token:UnTokenValido`

La respuesta será un objeto JSON que se verá tal que así:

```
1 {  
2   "status": "success",  
3   "datos": {  
4     "acknowledged": true,  
5     "deletedCount": 1  
6   }  
7 }
```

Código Fuente 4.37: Ejemplo de Respuesta

El apartado deletedCount hace referencia a los elementos borrados.

### Seleccionar Dispositivo – PUT

`/api/dispositivo`

Petición para seleccionar un dispositivo y cambiar el campo seleccionado a true. Eso hace que todos los demás dispositivos se vuelvan false. La entrada será el ID del dispositivo que se suministrador a través del body en el siguiente formato:

```
1 {  
2   "dID": "12344523",  
3 }
```

Código Fuente 4.38: Objeto JSON enviado por el Body

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

`token:UnTokenValido`

La respuesta será un objeto JSON que se verá tal que así:

```
1 {  
2   "status": "success",  
3 }
```

Código Fuente 4.39: Respuesta correcta

#### 4.6.4.4. Reglas de Guardado

##### Modificar status – PUT

/api/reglaGuardado

Petición para encender o apagar una regla y cambiando el campo status a true o false respectivamente. Eso permite elegir si la regla para guardar los fatos recibido funciona no. La entrada será el ID de la regla de EMQX que se suministrador a través del body, así como el status que se solicita, en el siguiente formato:

```
1 {  
2   "regla": {  
3     "emqxReglaID": "rule:e5c3d0ff",  
4     "status": true  
5   }  
6 }
```

Código Fuente 4.40: Objeto JSON enviado por el Body

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```
1 {  
2   "status": "success",  
3 }
```

Código Fuente 4.41: Respuesta correcta

#### 4.6.4.5. Tipos de Dispositivos

##### Creacion - POST

/api/tipodDispositivo

Esta petición se hace a través del método POST para la creación de un objeto del tipo tipo de dispositivo. Se solicita mediante el body un objeto en formato JSON con estos parámetros:

```
1 {  
2   "nuevoTipoDispostivo": {  
3     "nombre": "Placa5",  
4     "tipoDispositivo": "diposivoPTueb",  
5     "fechaCreacion": "1663",  
6     "variables": [  
7       {  
8         "nombre": "variable1",
```

```

9      "tipo": "Number",
10     "tambBuffer": "1",
11     "compWidgets": [
12         "IotBoton",
13         "IotSwitch",
14         "IotIndicador",
15         "IoTGraficaNum"
16     ]
17 },
18 {
19     "nombre": "humedad",
20     "tipo": "other",
21     "tambBuffer": "1",
22     "compWidgets": [
23         "IotBoton",
24         "IotSwitch",
25         "IotIndicador",
26         "IoTGraficaNum"
27     ]
28 }
29 ]
30 }
31 }

```

Código Fuente 4.42: Objeto JSON enviado por el Body

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```

1 {
2     "status": "success",
3 }

```

Código Fuente 4.43: Respuesta correcta

### Listado - GET

/api/tipodDispositivo

Petición usada a través del método GET para la obtención de una lista de tipos de dispositivos. Y hay que suministrarle en el header el token de autenticación tal que así:

token:UnTokenValido

La respuesta será un objeto en formato JSON contenido por otros objetos en el mismo formato con los datos de cada dispositivo. Un ejemplo puede ser:

```

1 {
2     "status": "success",

```

```

3  "data": [
4    {
5      "_id": "636fc68c2f78a0fe3c83c633",
6      "nombre": "Placa5",
7      "tipoDispositivo": "dispositivoPTueb",
8      "variables": [
9        {
10         "nombre": "variable1",
11         "tipo": "Number",
12         "tambBuffer": "1",
13         "compWidgets": [
14           "IotBoton",
15           "IotSwitch",
16           "IotIndicador",
17           "IoTGraficaNum"
18         ]
19       },
20       {
21         "nombre": "humedad",
22         "tipo": "other",
23         "tambBuffer": "1",
24         "compWidgets": [
25           "IotBoton",
26           "IotSwitch",
27           "IotIndicador",
28           "IoTGraficaNum"
29         ]
30       }
31     ],
32     "fechaCreacion": 1668269708258,
33     "__v": 0
34   }
35 ]
36 }

```

Código Fuente 4.44: Ejemplo de Respuesta

### Borrado - DELETE

/api/tipodDispositivo?TipoDispositivoID=

En esta petición se usa para borrar un tipo de dispositivo a través del método delete y se utiliza el ID del tipo de dispositivo en el query o directamente en la uri para identificar el objeto y borrarlo, el parámetro es dispID. Por ejemplo:

/api/tipodDispositivo?TipoDispositivoID=Placa5

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```

1 {
2   "status": "Success",
3   "datos": {
4     "acknowledged": true,
5     "deletedCount": 1
6   }
7 }

```

Código Fuente 4.45: Ejemplo de Respuesta

#### 4.6.4.6. Alarmas

##### Creacion - POST

/api/reglaAlarma

Esta petición se hace a través del método POST para la creación de un objeto del tipo Alarma. Se solicita mediante el body un objeto en formato JSON con estos parámetros:

```

1 {
2   "regla": {
3     "dNombre": "Placa6",
4     "dID": 1234452332,
5     "userID": "",
6     "variableNombreCompleto": "variable1",
7     "variable": "variable1",
8     "status": false,
9     "condicion": "=",
10    "value": "2",
11    "triggerTime": "23"
12  }
13 }

```

Código Fuente 4.46: Objeto JSON enviado por el Body

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```

1 {
2   "status": "success",
3 }

```

Código Fuente 4.47: Respuesta correcta

##### Modificar status – PUT

/api/reglaAlarma

Petición para encender o apagar una regla y cambiando el campo status a true o false respectivamente. Eso permite elegir si la regla de alarma funciona



no. La entrada será el ID de la regla de EMQX que se suministrador a través del body, así como el status que se solicita, en el siguiente formato:

```
1 {  
2   "regla": {  
3     "emqxReglaID": "rule:675b7cc6",  
4     "status": true  
5   }  
6 }
```

Código Fuente 4.48: Objeto JSON enviado por el Body

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```
1 {  
2   "status": "success",  
3 }
```

Código Fuente 4.49: Respuesta correcta

#### **Borrado - DELETE**

/api/reglaAlarma?emqxReglaID=

En esta petición se usa para borrar uan regla de Alarma a través del método delete y se utiliza el ID de la regla EMQX en el query o directamente en la uri para identificar el objeto y borrarlo, el parámetro es emqxReglaID. Por ejemplo:

/api/reglaAlarma?emqxReglaID=rule:46fbf9d8

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```
1 {  
2   "status": "success"  
3 }
```

Código Fuente 4.50: Respuesta correcta

#### 4.6.4.7. Notificaciones

##### Listado - GET

/api/notificaciones

Petición usada a través del método GET para la obtención de una lista de notificaciones del usuario que no han sido leídas. Y hay que suministrarle en el header el token de autenticación tal que así:

token:UnTokenValido

La respuesta será un objeto en formato JSON contenido por otros objetos en el mismo formato con los datos de cada dispositivo. Un ejemplo puede ser:

```
1 {
2   "status": "success",
3   "data": [
4     {
5       "_id": "6370e0b61487be9b9e01fbf9",
6       "userID": "6307caa3bc1377581b1651e3",
7       "dID": "1234452332",
8       "dNombre": "Placa6",
9       "payload": {
10         "value": 2
11       },
12       "emqxReglaID": "rule:11947517",
13       "variable": "variable1",
14       "variableNombreCompleto": "variable1",
15       "topic": "6307caa3bc1377581b1651e3/1234452332/variable1
16     /sdata",
17     "value": 2,
18     "condicion": "=",
19     "leido": false,
20     "fecha": 1668341942993,
21     "__v": 0
22   }
23 ]
}
```

Código Fuente 4.51: Ejemplo de Respuesta

##### Modificar status – PUT

/api/notificaciones

Petición para cambiar el estado de una notificación y cambiando el campo status a true. Eso permite saber si la notificación ha sido leída o no. La entrada será el ID de la notificación que se suministrador a través del body, así como el status que se solicita, en el siguiente formato:

```
1 {
2   "regla": {
3     "emqxReglaID": "rule:675b7cc6",
4     "status": true
5   }
6 }
```

```

5     }
6 }

```

Código Fuente 4.52: Objeto JSON enviado por el Body

Además, a través es necesario adjuntar el token de autenticación en el header tal que así:

token:UnTokenValido

La respuesta será un objeto JSON que se verá tal que así:

```

1 {
2   "status": "success",
3 }

```

Código Fuente 4.53: Respuesta correcta

#### 4.6.4.8. Web Hooks

Estos end-points son algo más espaciales ya que son lo utilizados por parte del broker para comunicarse con la API, esto se hace a través de los recursos que hemos mencionado en apartados anteriores. Estos recursos envían la información a los end-points que le hayamos dicho cuando los creamos en el momento que alguna de las reglas asociadas a él se cumpla. Por lo que el cometido de estos end-point es la de recibir mensajes del broker. Nosotros hemos realizado dos, uno por cada recurso que se han creado, es decir, uno para las alarmas y otro para el guardado.

##### Guardar Webhook - POST

/api/guardar-webhook

En esta petición se hace a través del método POST y trata de resolver las peticiones de guardado de información. Se encarga de guardar la información correspondiente en la base de datos. Se solicita mediante el body un objeto en formato JSON con estos parámetros:

```

1 {
2   "userID": "6307caa3bc1377581b1651e3",
3   "payload": {
4     "value": 3,
5     "save": 1
6   },
7   "topic": "6307caa3bc1377581b1651e3/1234452332/variable1/sdata"
8 }

```

Código Fuente 4.54: Objeto JSON enviado por el Body

Para más seguridad se habilitado un token de seguridad para que no pueda acceder gente externa y es un token distinto al JWT del usuario (en nuestro caso 121212) y se envía a través del header de la siguiente forma:

token:UnTokenValido

La respuesta será status de http, si todo ha salido bien será un 200.

### Alarma Webhook - POST

/api/guardar-webhook

En esta petición se hace a través del método POST y trata de resolver las peticiones cuando regla alarma encendida ha sido activada, es decir, que ha cumplido la condición del usuario. También se encarga de enviar la notificación al usuario. Se solicita mediante el body un objeto en formato JSON con estos parámetros:

```
1 {  
2   "userID": "6307caa3bc1377581b1651e3",  
3   "dID": "1234452332",  
4   "dNombre": "Placa6",  
5   "payload": {  
6     "value": 2  
7   },  
8   "topic": "6307caa3bc1377581b1651e3/1234452332/variable1/sdata",  
9   "emqxReglaID": "rule:11947517",  
10  "value": 2,  
11  "condicion": "=",  
12  "variable": "variable1",  
13  "variableNombreCompleto": "variable1",  
14  "triggerTime": 23  
15 }
```

Código Fuente 4.55: Objeto JSON enviado por el Body

Para más seguridad se habilitado un token de seguridad para que no pueda acceder gente externa y es un token distinto al JWT del usuario (en nuestro caso 121212) y se envía a través del header de la siguiente forma:

token:UnTokenValido

La respuesta será status de http, si todo ha salido bien será un 200.

También se adjuntan en la documentación de algunos ejemplos en aplicación Postman, que se ha utilizado para realizar la batería pruebas, la colección de estas mismas se han adjuntado junto al código del proyecto, además incluye algunos ejemplos de peticiones realizadas.

## Capítulo 5

# Sprints

En este capítulo se encuentran los Sprints realizados para el desarrollo de la presente plataforma, así como el diagrama de Gran resultante.

### 5.1. Diagrama de Grant

El diagrama de Gantt resultante después del desarrollo del proyecto y después de todos los Sprints ha sido el siguiente.

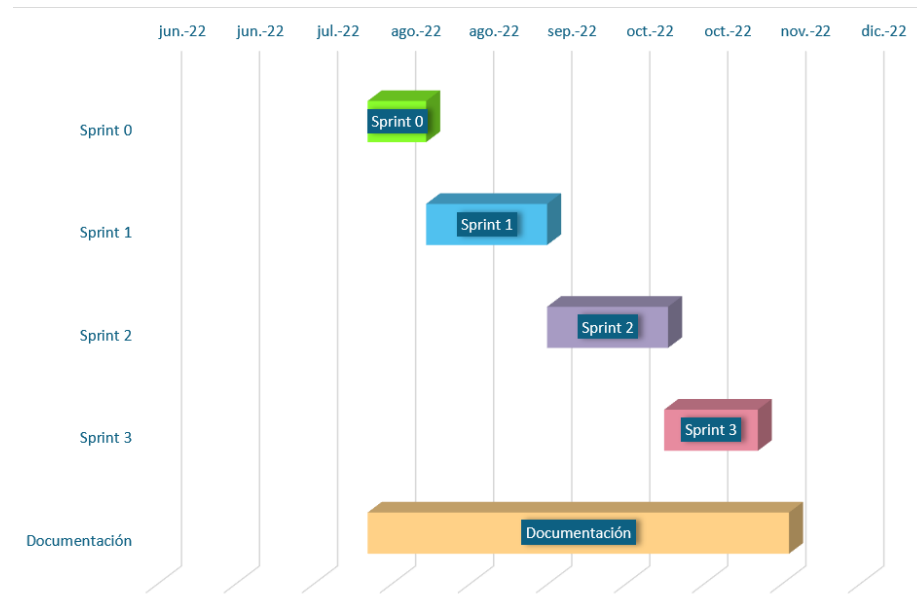


Figura 5.1: Diagrama de Gantt

Como se puede apreciar en el diagrama el proyecto ha durado el periodo de 4 meses entre los cuales se han ejecutado los Sprints, cada uno ha tenido una duración aproximada de 4 semana, exceptuando el Sprint 0 que ha durado 2 y el Sprint que ha durado 3.

Fuera de los Sprint en el periodo final se ha dedicado a la preparación de esta memoria así como en el arreglo de pequeños errores o la optimización del código resultante de los Sprits.

## **5.2. Sprints**

### **5.2.1. Sprint 0**

Este Sprint no tiene ninguna release asociada, se trata de un sprint «de aterrizaje», para buscar información relacionada con el proyecto, conocer y valorar tecnologías y trazar una ruta de guía más o menos sobre cómo organizar la plataforma y el ejemplo.

### **5.2.2. Sprint 1**

#### **5.2.2.1. Objetivos**

Maquetación de un front-end provisional y de una API para obtener recursos de las BBDD.

#### **5.2.2.2. Tareas realizadas**

- Elegir las herramientas para el desarrollo.
- Elegir la plantilla para el front-end.
- Maquetación Front-End.
- Quitar lo no necesario de la plantilla
- Creación de una estructura basada en secciones (Plantillas, Dispositivos, Alarmas y Dashboard).
- Modelos y Acciones sobre estos:
  - Plantillas
    - Create, Get y delete
    - Modelo en BD
  - Dispositivos
    - Create, Get y delete
    - Modelo en BD
  - Usuarios

- Create y Get
  - Modelo en BD
- Autenticación por JWT.
- Documentación con ejemplos de la API en Postman.
- Pequeño Docker de prueba.
- Conexión Front-end con la API.
- Batería de pruebas en Postman para plantillas.
- Login y registro.

#### 5.2.2.3. PMV v.1

El cliente recibe un PMV en el que puede visualizar como quedaría la interfaz, así como, probar algunas funcionalidades sobre Dispositivos y Plantillas, hacer login y registrarse.

#### 5.2.2.4. Pruebas

- Programación de try-catch 1: validación respuesta correcta de la API en `getPlantillas()` en `dispositivos.vue`.
- Programación de try-catch 2: validación respuesta errónea al guardar los datos de la Plantilla en `guardarPlantilla()` en `template.vue`.
- Programación de try-catch 3: validación respuesta correcta de la API en `getPlantillas()` en `template.vue`.
- Programación de try-catch 4: validación respuesta de la API al borrar una Plantilla `borrarPlantilla()` en `template.vue`.
- Programación de try-catch 5: validación respuesta válida en `procesadoDatosRecibidos(data)` en `IotIndicator.vue`.
- Programación de try-catch 6: validación Respuesta correcta de la API en `getPlantillas()`.
- Programación de try-catch 7 validación Respuesta correcta de la API en `getPlantillas()`.
- Programación de try-catch 8: validación Respuesta correcta de la API en `getPlantillas()`.
- Programación de try-catch 9: validación Respuesta correcta de la API en `getPlantillas()`.
- Programación de try-catch 10: validación Respuesta correcta

- Beta Testing 1: validación superada en formulario de registro de Dispositivos
- Beta Testing 2: validación superada en el registro de un usuario.
- Beta Testing 3: validación superada en el login de un usuario.
- Beta Testing 4: validación superada en el registro de una Plantilla.
- Beta Testing 5: validación superada en la visualización de Tablas de Plantillas y Dispositivos.
- Beta Testing 6: validación superada en el botón de borrado de Dispositivos y Plantillas en las tablas.
- Beta Testing 7: validación superada en la visualización de prueba de cada widget.

#### **5.2.2.5. Comentarios**

Para este Sprint se tenía pensado hacer una maqueta para así conseguir un acercamiento visual a la plataforma que se quería alcanzar al final del proceso. También sirve como punto de contacto con muchas de las tecnologías y procesos de desarrollo.

Se pensó un modelo en el que casi todo fuera personalizable, en la que crearás dispositivos y estos tendrían asociados unas plantillas, en las que contendrían widgets que interactuarían con el usuario y mostrarán información, todo esto totalmente personalizable para el usuario.

Para la maqueta no se utilizan datos reales y algunos resultados están fijos debido a que solo son una muestra de lo que contendrán, ya que aún no tiene conexión con la base de datos.

Para el apartado visual se optó por una plantilla con varias opciones que aligeraría bastante el desarrollo y definiendo a la vez el estilo de la aplicación, basado en secciones que puedes ir visualizando haciendo click sobre ellas en un menú en la derecha, a lo que a partir de ahora llamaremos páginas. Esto también me dio la posibilidad de poner en ese menú las futuras ampliaciones de la aplicación como conexiones con otras aplicaciones por poner un ejemplo o facilitar la adicción de nuevas funcionalidades.



#### 5.2.2.6. Capturas de pantalla

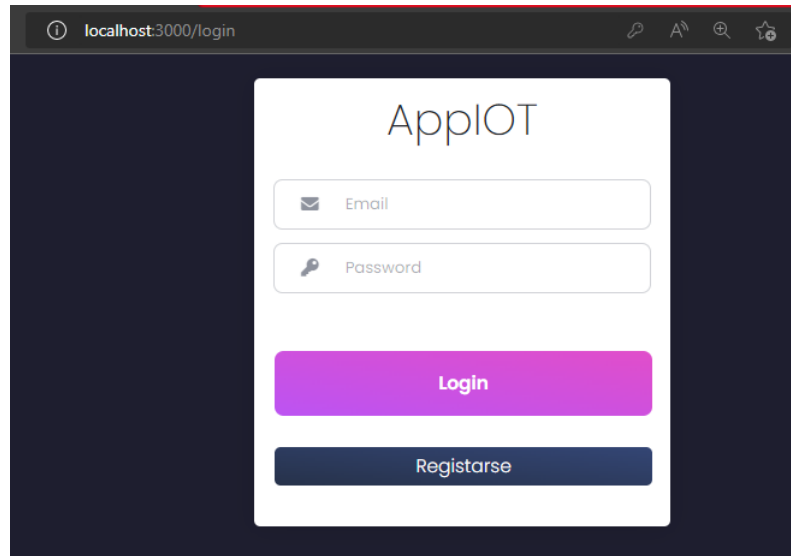


Figura 5.2: Login

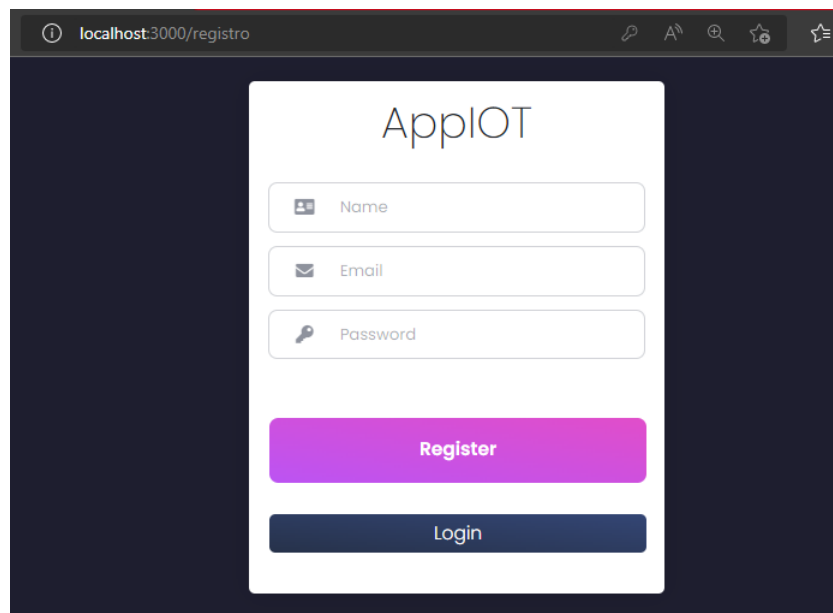


Figura 5.3: Registro

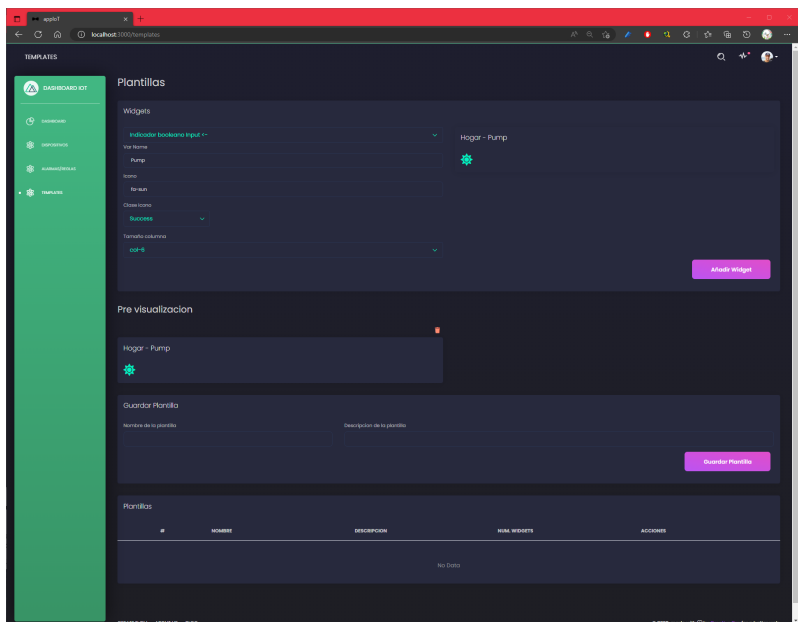


Figura 5.4: Página de Dispositivos

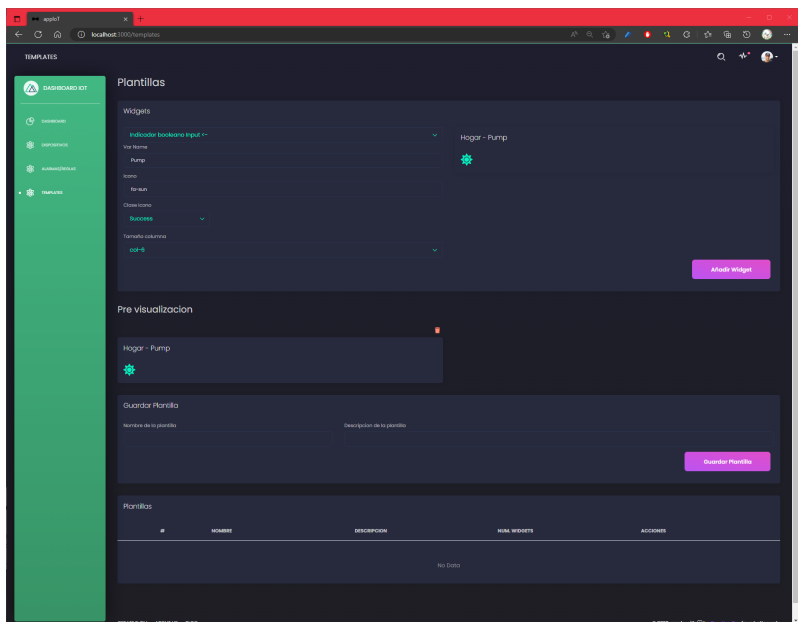


Figura 5.5: Página de Plantillas con la creación de 1 widget

### 5.2.3. Sprint 2

#### 5.2.3.1. Objetivos

Construcción de la conexión con BDD, broker MQTT y la entrada/salida de datos.

#### 5.2.3.2. Tareas realizadas

- Conexión con EMQX (Broker MQTT).
- Finalización de la página de Dashboard.
- Modelos y Acciones sobre estos:
  - Reglas
    - Create, Get y delete
    - Modelo en BD
  - Alarmas
    - Create, Get y delete
    - Modelo en BD
  - Notificaciones
    - Create, Get y delete
    - Modelo en BD
- Modelo de datos adicionales como: Datos de dispositivos, autenticación en broker.
- Notificaciones en pantalla.
- Secciones de Alarmas y Reglas.
- Documentación con ejemplos de la API en Postman.
- Completar el Docker.
- Conexión con Base de datos (Mongo).
- Desconexión.

#### 5.2.3.3. PMV v.2

El cliente recibe un PMV con conexiones con base datos y broker que puede probar la mayoría de las funciones de manera más realista pero aún sin datos.

#### 5.2.3.4. Pruebas

- Programación de try-catch 11: validación de añadir una Alarma en la API en `router.post('/reglaAlarma')` en `alarma.js`.
- Programación de try-catch 12: validación al actualizar una Alarma en la API en `router.put('/reglaAlarma')` en `alarma.js`.
- Programación de try-catch 13: validación al borrar una Alarma en la API en `router.delete('/reglaAlarma')` en `alarma.js`.
- Programación de try-catch 14: validación de la creación correcta de una Alarma en `crearReglaAlarma(nuevaAlarma)` en `alarma.js`.
- Programación de try-catch 15: validación en la actualización correcta del status de una Alarma en `updateStatusAlarma(emqxReglaID, status)` en `alarma.js`.
- Programación de try-catch 16: validación del borrado de una Alarma en `borrarReglaAlarma(emqxReglaID)` en `alarma.js`.
- Programación de try-catch 17: validación de la obtención de datos en `router.get('/get-small-charts-data')` en `dataprovaiders.js`.
- Programación de try-catch 18: validación la petición de Dispositivos en la API en `router.get('/dispositivo')` en `dispostivos.js`.
- Programación de try-catch 19: validación de añadir un Dispositivo en la API en `router.post('/dispositivo')` en `dispostivos.js`.
- Programación de try-catch 20: validación de borrar un Dispositivo en la API en `router.delete('/dispositivo')` en `dispostivos.js`.
- Programación de try-catch 21: validación al actualizar un Dispositivo en la API en `router.put('/dispositivo')` en `dispostivos.js`.
- Programación de try-catch 22: validación actualizar una Regla de Guardado en la API en `router.put('/reglaGuardado')` en `dispostivos.js`.
- Programación de try-catch 23: validación al seleccionar un Dispositivo en `seleccionarDispositivo(userIDrecib, dispIDrecib)` en `dispostivos.js`.
- Programación de try-catch 24: validación al elegir un nuevo Dispositivo cuando se borra en `seleccionarDispositivoBorrado(userIDrecib, dispIDrecib)` en `dispostivos.js`.
- Programación de try-catch 25: validación al conseguir una Regla de guardado en `getReglasGuardado(userID)` en `dispostivos.js`.
- Programación de try-catch 26: validación al crear una Regla de guardado en `crearReglaGuardado(userID,dID,status)` en `dispostivos.js`.

- Programación de try-catch 27: validación al actualizar una Regla de guardado en `updateStatusReglaGuardado(emqxReglaID, status)` en `dispositivos.js`.
- Programación de try-catch 28: validación al borrar una Regla de guardado en `borrarReglaGuardado(dID)` en `dispositivos.js`.
- Programación de try-catch 29: validación al obtener las Plantillas en `getPlantillas(userID)` en `dispositivos.js`.
- Programación de try-catch 30: validación al obtener las Alarmas en `getAlarmas(userID)` en `dispositivos.js`.
- Programación de try-catch 32: validación al borrar Alarmas en `borrarAlarmas(userID, dispID)` en `dispositivos.js`.
- Programación de try-catch 33: validación al borrar las credenciales de MQTT de un dispositivo en `borrarMqttCredenciales(dispID)` en `dispositivos.js`.
- Programación de try-catch 34: validación al listar una Recurso en `listarRecursos()` en `emqxAPI.js`.
- Programación de try-catch 35: validación al crear una Recurso en `crearRecurso(recursoNuevo)` en `emqxAPI.js`.
- Programación de try-catch 36: validación de añadir una Plantilla en la API en `router.post('/plantilla')` en `plantilla.js`.
- Programación de try-catch 37: validación la petición de Plantillas en la API en `router.get('/plantilla')` en `plantilla.js`.
- Programación de try-catch 38: validación de borrar una Plantilla en la API en `router.delete('/plantilla')` en `plantilla.js`.
- Programación de try-catch 39: validación de registrar un Usuario en la API en `router.post('/registro')` en `usuarios.js`.
- Programación de try-catch 40: validación de loguear un Usuario en la API en `router.post('/login')` en `usuarios.js`.
- Programación de try-catch 41: validación al conseguir las credenciales MQTT de un Usuario en la API en `router.post('/getMqttcredenciales')` en `usuarios.js`.
- Programación de try-catch 42: validación al conseguir las credenciales MQTT de un Usuario en la API en `router.post('/getMqttcredencialesReconexion')` en `usuarios.js`.
- Programación de try-catch 43: validación al conseguir las credenciales MQTT de un Usuario en `getMqttCredencialesWeb(userID)` en `usuarios.js`.

- Programación de try-catch 44: validación al conseguir las credenciales MQTT de un Usuario en `getMqttCredencialesWebReconexion(userID)` en `usuarios.js`.
- Programación de try-catch 45: validación al obtener el mensaje para guardar del Webhook en la API en `router.post('/guardar-webhook')` `webhooks.js`.
- Programación de try-catch 46: validación al obtener el mensaje para de alarma del Webhook en la API en `router.post('/alarma-webhook')` `webhooks.js`.
- Programación de try-catch 47: validación al obtener las Notificaciones en la API en `router.get('/notificaciones')` `webhooks.js`.
- Programación de try-catch 48: validación al actualizar la Notificación en la API en `router.post('/notificaciones')` `webhooks.js`.
- Programación de try-catch 49: validación al guardar una Notificación en Mongo en `guardarNotificacionMongo(alarmaEntrante)` en `webhooks.js`.
- Programación de try-catch 50: validación al actualizar el contador de una Notificación en Mongo en `actualizarAlarmaContador(emqxReglaID)` en `webhooks.js`.
- Programación de try-catch 51: validación al obtener las Notificaciones de un usuario en `getNotificaciones(userID)` en `webhooks.js`.
- Programación de try-catch 52: validación al borrar Alarmas en `borrarAlarmas(userID, dispID)` en `webhooks.js`.
- Beta Testing 1: validación superada en la repetición de los beta testing anteriores.
- Beta Testing 2: validación superada en el formulario de una nueva Alarma.
- Beta Testing 3: validación superada en la visualización de Tablas de Alarmas.
- Beta Testing 4: validación superada en la visualización de Notificaciones.
- Beta Testing 5: validación superada en la desconexión del usuario.
- Beta Testing 6: validación superada en el botón de borrado de Alarmas.
- Beta Testing 7: validación superada en el botón de seleccionar Dispositivo.
- Beta Testing 8: validación superada en la visualización de las Plantillas en la página de Dashboard.

#### 5.2.3.5. Comentarios

En este Sprint se dio prioridad a terminar la mayor parte de las conexiones de la aplicación, tanto interna como externa. Para ello se pudo añadir funcionalidades que se necesitaban conexiones para poder implementarlas como son las Reglas, Alarmas y Notificaciones. Solo una de ellas, Alarmas tiene página propia, las otras dos están integradas en la aplicación.

Para alarmas se buscaba que el usuario pudiera crear avisos con los que se le avisara si la variable de un dispositivo violaba esa regla que había realizado. Aquí viene la parte de reglas, las cuales las utilizamos para poder crear esas alarmas en nuestro broker. Y las notificaciones son la respuesta que recibimos por parte del broker cuando se rompe esa norma. De momento solo puede visualizar las notificaciones y al hacer click sobre ellas desaparecen. Se está estudiando el crear una vista para poder visualizar el histórico de estas notificaciones.

También se aplican las correspondientes restricciones y borrado en cascada para que al borrar un dispositivo se borre las alarmas y reglas asociadas a él, y no poder borrar una plantilla hasta que se borre el dispositivo correspondiente. Aunque, se está estudiando otra lógica de datos, donde se restringe la libertad a la hora de crear dispositivos por unos ya predefinidos para hacer la adicción de nuevos dispositivos de forma fácil y rápida.

También se ha aplicado una serie de reglas para poder estandarizar las conexiones con los dispositivos, como que los tópicos de suscripción de los dispositivos tienen que seguir unas normas, he intentar estandarizar lo máximo posible para poder hacer más fácil el mantenimiento de la aplicación en el futuro.

Por último se ha trabajado en el despliegue, para ello se ha usado la tecnología docker para modularizar las herramientas y con docker-compose orquestarlas para si hacerlas funcionar juntas. En esta parte ha habido varios errores ya que algunos módulos usados en Nodejs no eran compatibles con ello, por lo que se ha dejado como trabajo futuro el cambio correspondiente para subsanar todos los fallos. Pese a esto se ha incorporado un docker-compose para hacerlo, y si se necesita que funcione, se debe eliminar la parte de Nodejs y ejecutarla en local.

### 5.2.3.6. Capturas de pantalla

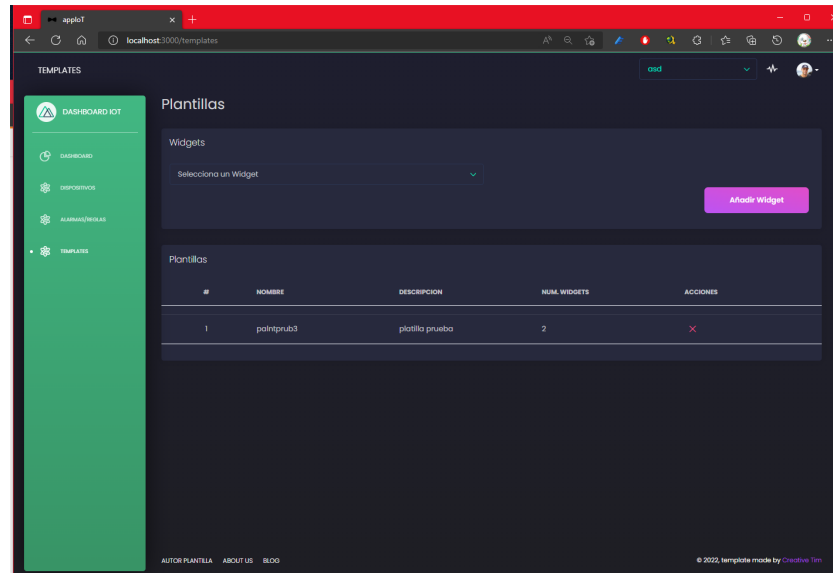


Figura 5.6: Página de Plantillas

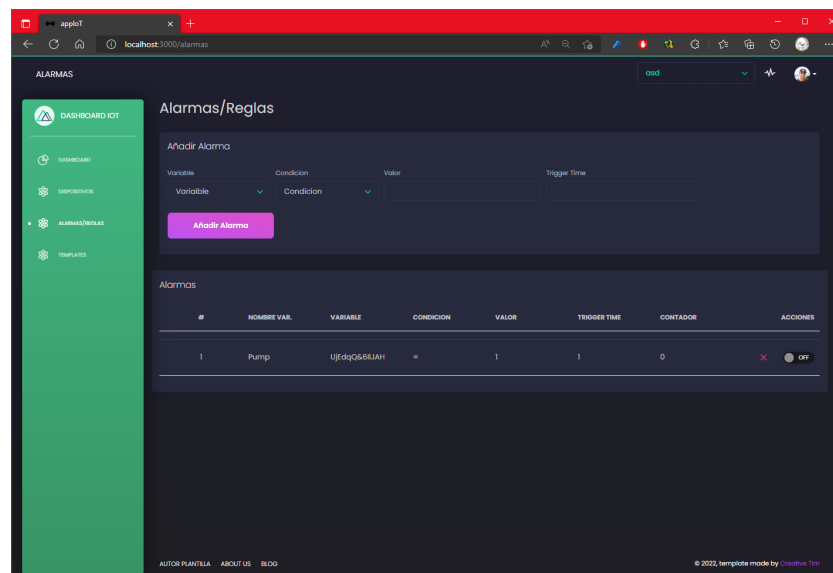


Figura 5.7: Página de Alarmas



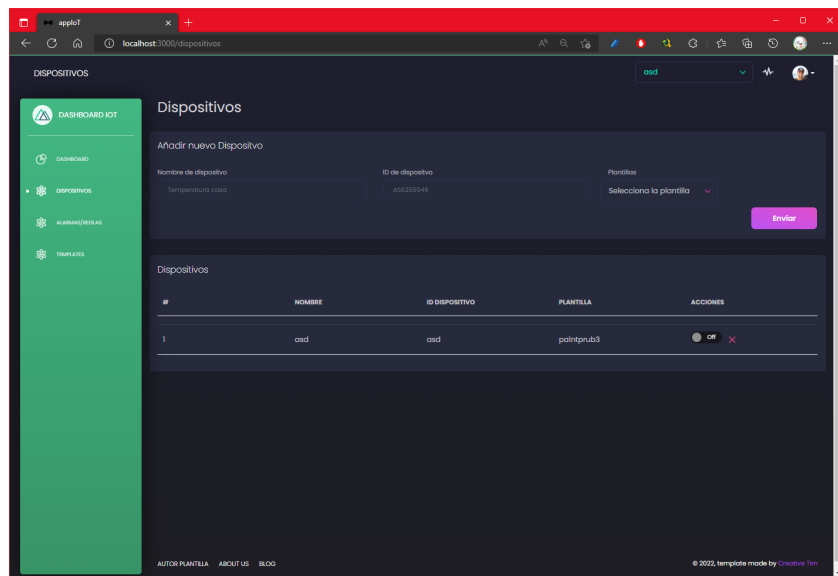


Figura 5.8: Página de Dispositivos

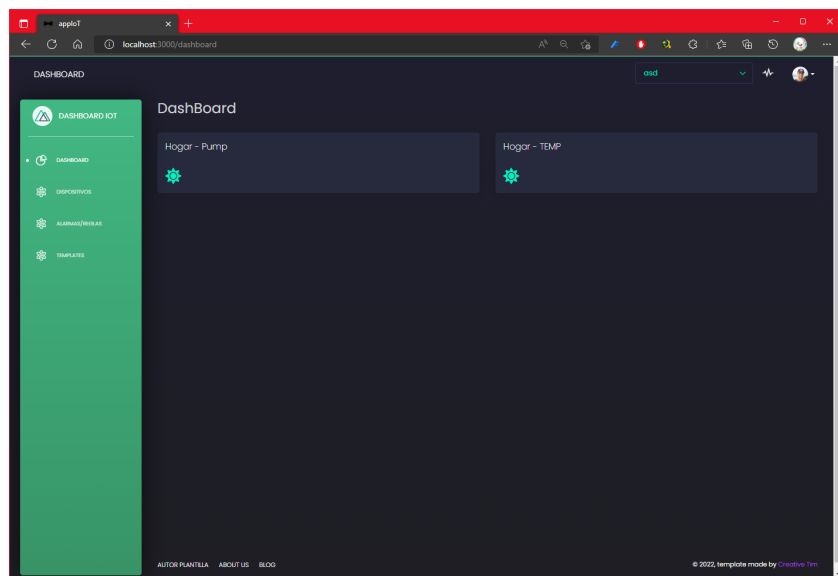


Figura 5.9: Página de Dashboard

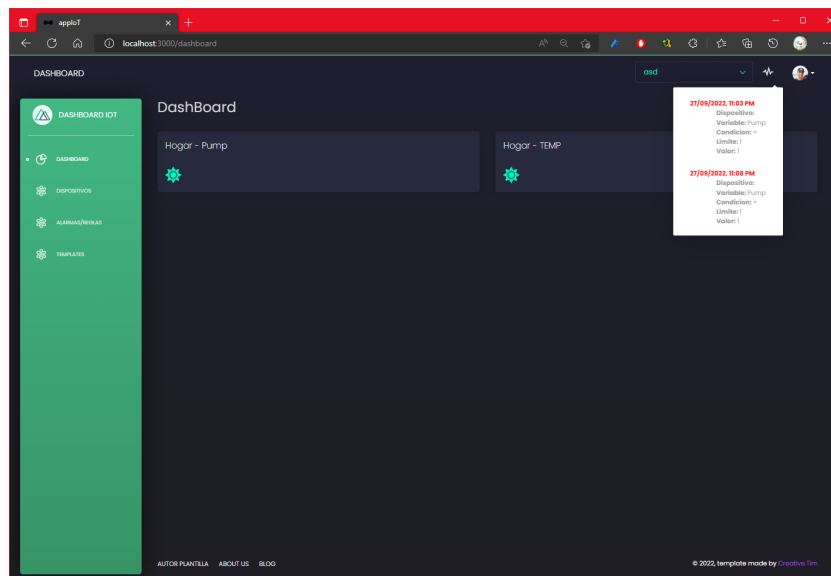


Figura 5.10: Página de Dashboard con Notificaciones abiertas

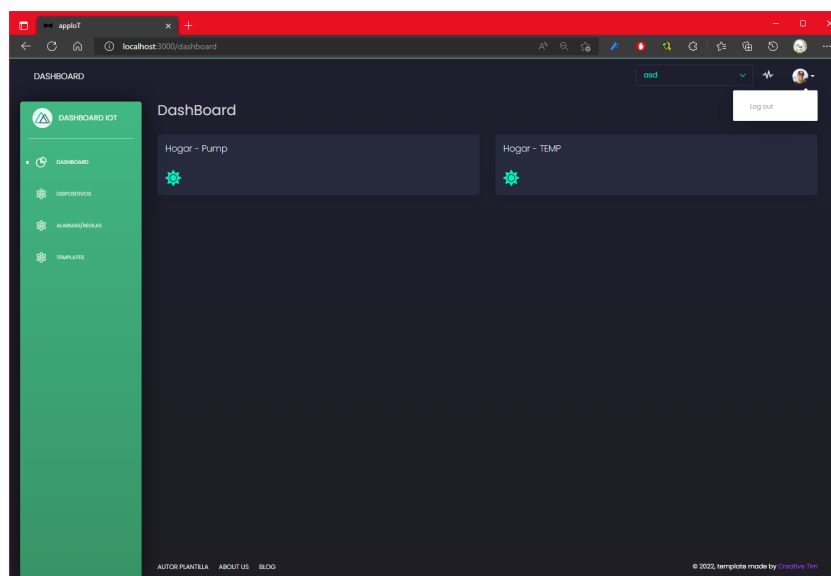


Figura 5.11: Página de Dashboard con el botón de logout

## 5.2.4. Sprint 3

### 5.2.4.1. Objetivos

Modificación del modelo de datos utilizado actual basado en plantillas, a uno basado en dispositivos.

### 5.2.4.2. Tareas realizadas

- Modificación de los modelos de datos actuales.
- Creación de los modelos de datos de Tipos de Dispositivos.
- Mejoras menores en distintas partes del código.

### 5.2.4.3. PMV v.3

El cliente recibe un PMV con las nuevas modificaciones del modelo de datos y con las consecuentes modificaciones internas de la aplicación.

### 5.2.4.4. Pruebas

- Programación de try-catch 31: validación al obtener los Tipos de dispositivos en `getTiposDispositivos()` en `dispositivos.js`.
- Programación de try-catch 53: validación la petición de tipos de Dispositivos en la API en `router.get('/tipodDispositivo')` en `tipoDispositivos.js`.
- Programación de try-catch 54: validación de añadir un tipo de Dispositivo en la API en `router.post('/tipodDispositivo')` en `tipoDispositivos.js`.
- Programación de try-catch 55: validación de borrar un tipo de Dispositivo en la API en `router.delete('/tipodDispositivo')` en `tipoDispositivos.js`.
- Programación de try-catch 56: validación al verificar las variables de los Tipos de dispositivos en `verificaionVariables(variables)` en `tipoDispositivos.js`.
- Beta Testing 1: validación superada en la repetición de los beta testing anteriores.
- Beta Testing 2: validación superada en la adición de un Tipo de dispositivo a través de la API.

### 5.2.4.5. Comentarios

En sprints anteriores se terminó viendo que el modelo usado hasta ahora podía resultar muy versátil pero poco eficiente ya que una misma plantilla podría valer para varios dispositivos, pero un mismo tipo de dispositivo podía enviar variables distintas a las que vemos si por ejemplo nosotros nombramos humedad y el dispositivo tiene como primer sensor temperatura.

Por eso se optó por una solución que es crear un nuevo tipo de dato llamado Tipo de dispositivo y que suplirá esa necesidad de asignar a cada tipo de dispositivo solo los sensores que les corresponde, así como su nombre, aunque el usuario podrá variar de la manera que necesite. Esto también ayuda a establecer unos parámetros fijos en la parte del dispositivo que hasta ahora podían variar, haciendo así unas reglas a la hora de enviar datos.

Ahora también es posible gracias a la inclusión de un desplegable para cambiar de plantillas en la página de Dashboard, y que estas puedan contener Widgets de diferentes dispositivos, cosa que antes no era posible.

La modificación del modelo por otro, me ha ayudado a ver que los pasos que ido dando han sido correctos porque gracias en parte a esa abstracción que se hizo han sido muy sencillo el modificar las partes correspondientes para adaptarlo todo, reforzando la idea de que la plataforma de desarrollo tiene mucho potencial para adaptarse y ser extensible.

#### 5.2.4.6. Capturas de pantalla

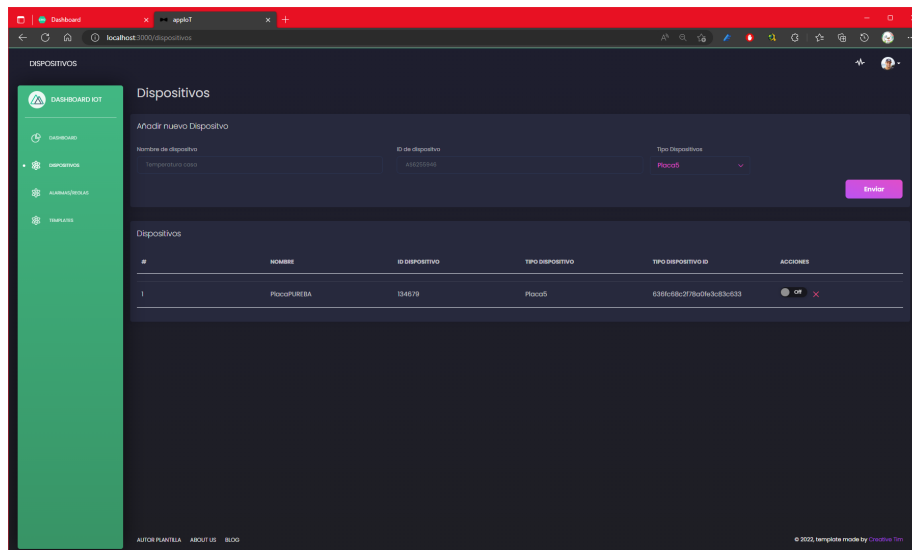


Figura 5.12: Página de Dispositivos

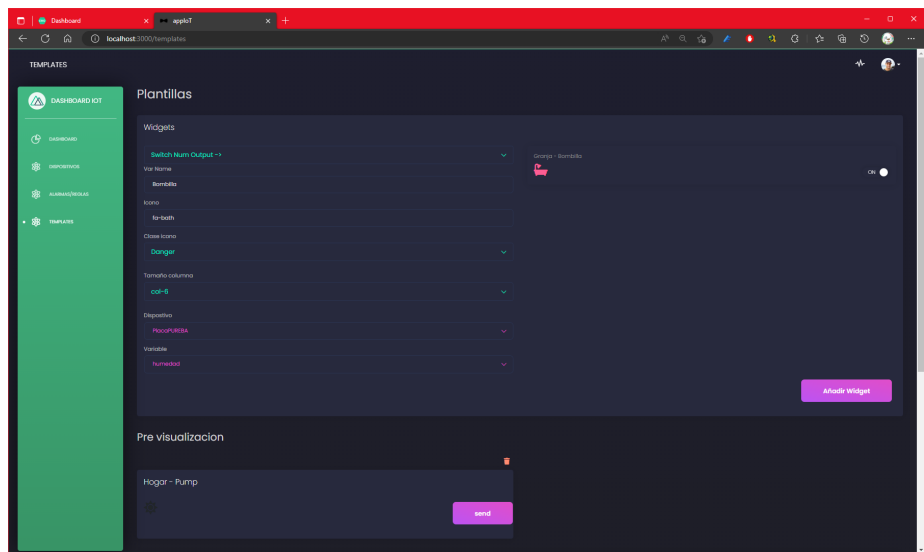


Figura 5.13: Página de Plantillas

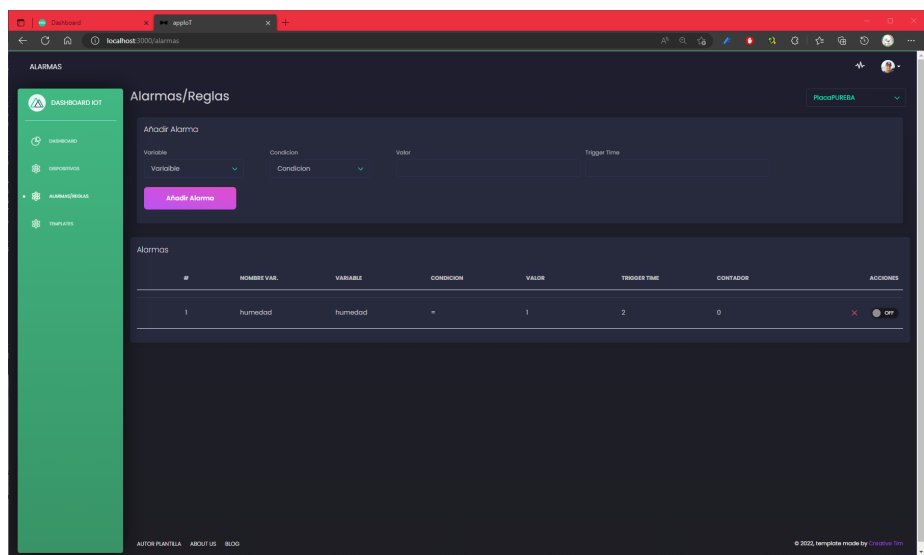


Figura 5.14: Página de Alarmas

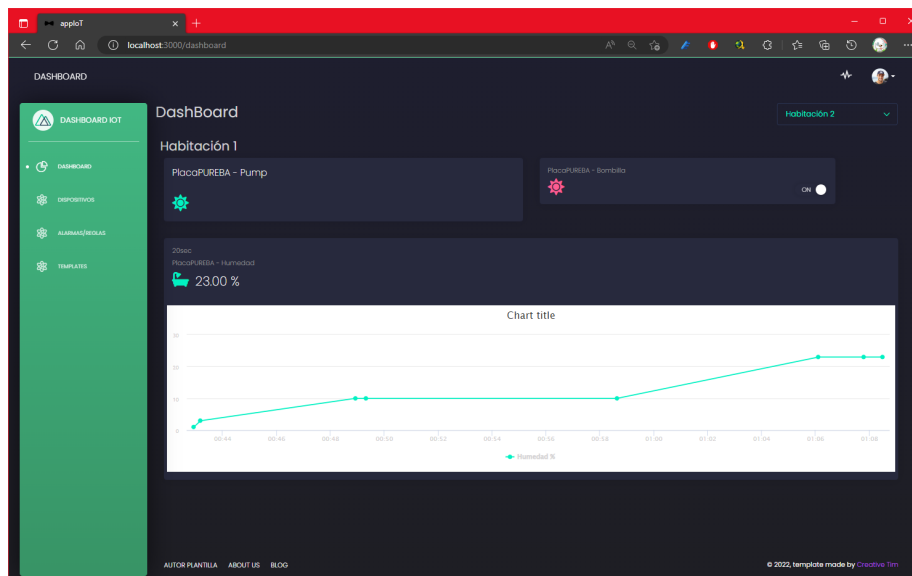


Figura 5.15: Página de Dashboard con un ejemplo

### 5.3. Despliegue

Para el despliegue se ha optado por el uso de contenedores docker para hacer la plataforma lo más portable y extensible posible. El despliegue llevado acabo se ha hecho en local, si se deseara llevarlo a producción se debería de modificar las IPs en el interior de la plataforma y en el Dockerfile.

Se ha dividido en los siguientes módulos:

- **Mongo:** En este docker se encuentra la base de datos.
- **EMQX:** Aquí encontramos el broker EMQX.
- **Node.js:** En este contenedor está situada la API y el Front-end

Con ello se ha creado un archivo de *docker-compose* para orquestar los contenedores. Antes de ejecutar el archivo hay que cambiar las IPs que aparece en el archivo de *docker-compose*, así como las distintas IPs usadas dentro del proyecto ya que están puestas para el trabajo en local. Después solo bastaría con ejecutar el archivo de *docker-compose* con *docker-compose up* y ya estaría funcionando.

Para conectarse al front-end se ingresaría por el puerto 3001 y para la interfaz gráfica de EMQX por el 18083.

Si se quiere pasar bastaría con usar *docker-compose down* sobre la carpeta que está el archivo de *docker-compose*.

El código así como estas instrucciones están en proyecto de Github. Cuyo enlace en el siguiente [link](#) .

## Capítulo 6

# Conclusiones y trabajo futuro

Para concluir el presente proyecto vamos a hacer un repaso de los objetivos y del desarrollo que se ha llevado a cabo, y decisiones tomadas, para que estos se cumplieran. El objetivo principal era el de realizar un prototipo de una plataforma IoT que recopilara, procesara y se pudieran visualizar los datos, así como una gestión de dispositivos, además de que fuera visualmente agradable y de fácil manejo para los usuarios. Con seguridad podemos decir que se han asentado unas bases y que el prototipo realizado es extensible y adaptable para el futuro.

La plataforma recopila los datos gracias a los instrumentos asociados a las reglas y a los recursos del broker EMQX como hemos mencionado en reiteradas ocasiones durante la memoria. Esta funcionalidad se puede habilitar o deshabilitar mediante los correspondientes botones de configuración. Los recursos y reglas procesan la información de manera que queda ordenada para su posterior almacenamiento en la base de datos.

La visualización se delega a los componentes que se han incluido, los Widgets, que realizan diferentes funciones, como la representación de datos o acciones del sobre los sensores, lo que ofrece infinitas posibles formas de visualización, además de ser personalizables por el usuario. Estos widgets se reúnen en plantillas, que también son personalizables por el usuario, y las cuales son mostradas en el dashboard, pudiendo cambiar las distintas plantillas creadas mediante un selector, cumpliendo así todos los objetivos de personalización de las vistas y explotación de los datos que se han marcado en los objetivos.

El objetivo de gestión de dispositivos se ha conseguido mediante la creación de distintos canales de paso de mensajes a través de la especificación de los distintos topics, además el usuario puede gestionar los dispositivos desde la propia



aplicación, añadiéndolo o quitándolo a su gusto, incluyendo la creación de avisos sobre los distintos sensores a través de lo que hemos definido como Alarma, que a su vez implementa un sistema de notificaciones para el aviso del cumplimiento de estas.

Todo esto bajo el paraguas de un front-end sencillo de entender y de manejar pero con todas las funcionalidades y completo, cumpliendo así el último objetivo que era garantizar que se ha cumplido con mejorar la experiencia del usuario se deben haber aplicado técnicas concretas. Esto último se ha conseguido en el front-end gracias al uso del sistema de Templates-páginas de Nuxt.js que nos ayuda a mantener y ahorrar código y en la parte del back-end con el uso de funciones y comentarios en el código.

Por ultimo, decir que la escalabilidad está asegurada gracias a las tecnologías escogidas son altamente escalables como EMQX y MongoDB, las cuales nos ofrecen una alta escalabilidad. En términos de número de dispositivos que pueden incluir las aplicaciones

Para finalizar, a título personal puedo decir que he aprendido mucho del trabajo realizado, ya que la mayoría de las herramientas o tecnologías no las conocía. Esto ha hecho que crezca tanto profesionalmente como personalmente, dando me cuenta de que puedo afrontar retos que antes podía pensar que no podría con ellos y ahora sé que por muy difícil que sea con esfuerzo, tiempo y dedicación se puede conseguir cualquier cosa.

Si hablamos de la plataforma puedo decir que ha sido todo un reto ya que la plataforma ha ido creciendo poco a poco, por lo que he podido ver como crecía poco a poco y partes de las decisiones que se tomaban daban sus frutos como fue el nivel de abstracción en el diseño del front-end y el back-end, ya que me permitió cambiar un modelo de datos por otro con mucha facilidad. También ha habido momentos difíciles, algún error que no sabes dónde está pero que por él no funciona, pero al final acabas encontrado y solucionando.

En términos de funcionalidad he quedado satisfecho con lo que realiza la plataforma, ya que puede tener un gran recorrido por delante y espero poder seguir añadiendo funcionalidades nuevas como por ejemplo tener más tipo de de widgets, incluir otros protocolos IoT o incluso desarrollar una ampliación móvil. El potencial es inmenso al ser una plataforma extensible. También como trabajos futuros se contempla:

- Incluir nuevo Widgets y considerar otros protocolos que se usan en IoT.
- Desarrollo de una aplicación complementaria para dispositivos móviles.
- Documentación necesaria para que desarrolladores puedan desarrollar sobre la plataforma e incluir nuevas funcionalidades o para personalizar la plataforma a sus necesidades.

- Incluir nuevos métodos de manejo de datos no contemplados en el prototipo.
- La posibilidad de compartir dispositivos y plantillas entre usuarios.
- Mejorar el despliegue y hacerlo mas sencillo.

# Bibliografía

- [1] Li Da Xu Shancang Li and Shanshan Zhao. The internet of things: A survey. *Information Systems Frontiers*, (17):243–259, 2015. 10, 25
- [2] The ThingsBoard Authors. **ThingsBoard**, 2022. <https://thingsboard.io/>. 10, 26
- [3] DataDog. **DataDog**, 2022. <https://www.datadoghq.com/>. 10, 27
- [4] KaaIoT Technologies. **Kaa Enterprise IoT Platform**, 2022. <https://www.kaaiot.com/>. 10, 28
- [5] myDevices. **myDevices Cayenne**, 2022. <https://developers.mydevices.com/cayenne/features/>. 10, 30
- [6] npm. **npm Nuxt**, 2022. <https://www.npmjs.com/package/nuxt>. 10, 36
- [7] Grupo NWC10. **Programador VUE.JS**, 2022. <https://www.theblocklearning.com/producto/programador-vue/>. 10, 36
- [8] Amazon Web Services. **¿Qué es Docker?**, 2022. <https://aws.amazon.com/es/docker/>. 10, 37
- [9] Wikipedia. **MongoDB**, 10 2022. <https://es.wikipedia.org/wiki/MongoDB>. 10, 37, 38
- [10] Grupo G2. **NodeJs Development Services**, 2022. <https://www.g2.com/products/nodejs-development-services/reviews>. 10, 38
- [11] EMQ Technologies. **EMQX. The Most Scalable MQTT Broker**, 2022. <https://www.emqx.io/>. 10, 39
- [12] Postman. **Postman**, 2022. <https://www.postman.com/>. 10, 40
- [13] Wikipedia. **MQTT**, 10 2022. <https://en.wikipedia.org/wiki/MQTT>. 10, 66
- [14] EMQX. **EMQX Broker Distributed cluster design**, 2022. <https://www.emqx.io/docs/en/v4.3/advanced/cluster.html#distributed-erlang>. 11, 99, 101, 102

- [15] myDevices. **Infinite Plug-And-Play IoT Solutions**, 2022. <https://mydevices.com/>. 12, 29, 30
- [16] Creative Tim. **Creative Tim - Black dashboard**, 2022. <https://www.creative-tim.com/product/vue-black-dashboard>. 18, 72
- [17] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas. **Manifesto Agil**, 2001. <https://agilemanifesto.org/iso/es/manifesto.html>. 18
- [18] talent . **Salario para Ingeniero Informatico**, 2022. <https://es.talent.com/salary?job=ingeniero+informtico>. 21
- [19] Jobted . **Salario para Ingeniero Informatico**, 2022. <https://www.jobted.es/salario/informtico>. 21
- [20] Nanosystems RFID Working Group of the European Technology Platform on Smart Systems Integration (EPOSS)) Info D.4 Networked Enterprise, RFID Info G.2 Micro. *Internet of Things in 2020, Roadmap for the Future*. 9 2008. 23
- [21] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010. 24
- [22] Luis Llamas. **¿Qué es MQTT? Su importancia como protocolo IoT**. 4 2019. <https://www.luisllamas.es/que-es-mqtt-su-importancia-como-protocolo-iot/>. 25
- [23] The ThingsBoard Authors. **ThingsBoard Documentation**, 2022. <https://thingsboard.io/docs/>. 26
- [24] Wikipedia. **Datadog**, 4 2022. <https://es.wikipedia.org/wiki/Datadog>. 27
- [25] David Bernal González. **¿Qué son los contenedores de software y por qué utilizarlos?** 7 2021. <https://profile.es/blog/contenedores-de-software/>. 32
- [26] Mauricio Améndola. **Contenedores de software: Qué son y qué ventajas ofrecen**. 3 2022. <https://openwebinars.net/blog/contenedores-de-software-que-son-y-que-ventajas-ofrecen/>. 32
- [27] Wikipedia. **Virtualización a nivel de sistema operativo**, 9 2022. [https://es.wikipedia.org/wiki/Virtualizaci%C3%B3n\\_a\\_nivel\\_de\\_sistema\\_operativo](https://es.wikipedia.org/wiki/Virtualizaci%C3%B3n_a_nivel_de_sistema_operativo). 32
- [28] Mario Pérez Esteso. **¿Qué es MQTT?** 6 2015. <https://geekytheory.com/que-es-mqtt/>. 33

- [29] Wikipedia. **OASIS (organization)**, 10 2022. [https://en.wikipedia.org/wiki/OASIS\\_\(organization\)](https://en.wikipedia.org/wiki/OASIS_(organization)). 33
- [30] PickData . **MQTT vs CoAP, la batalla por ser el mejor protocolo IoT**, 10 2019. <https://www.pickdata.net/es/noticias/mqtt-vs-coap-mejor-protocolo-iot>. 34
- [31] Yúbal Fernández. **API: qué es y para qué sirve**. 8 2019. <https://www.xataka.com/basics/api-que-sirve>. 34
- [32] Amazon Web Services. **¿Qué es una API?**, 2022. <https://aws.amazon.com/es/what-is/api/#:~:text=API%20significa%20E2%80%9Cinterfaz%20de%20programaci%C3%B3n,de%20servicio%20entre%20dos%20aplicaciones>. 34
- [33] RedHat. **¿Qué es una API?** 10 2017. <https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces>. 34
- [34] Wikipedia. **Base de datos**, 10 2022. [https://es.wikipedia.org/wiki/Base\\_de\\_datos](https://es.wikipedia.org/wiki/Base_de_datos). 35
- [35] ayudaley. **Conoce las bases de datos SQL a fondo**, 2022. <https://ayudaleyprotecciondatos.es/bases-de-datos/sql/>. 35
- [36] Acens. **Bases de datos NoSQL. Qué son y tipos que nospodemos encontrar**, 2022. <https://www.acens.com/wp-content/images/2014/02/bbdd-nosql-wp-acens.pdf>. 35
- [37] Arimetrics. **Qué es Framework**, 2022. <https://www.arimetrics.com/glosario-digital/framework>. 35
- [38] Edix. **Framework**. 7 2022. <https://www.edix.com/es/instituto/framework/#:~:text=Un%20framework%20es%20un%20esquema,organizaci%C3%B3n%20y%20desarrollo%20de%20software>. 35
- [39] Manz.dev. **¿Qué es Vue?**, 2022. <https://lenguajejs.com/vuejs/introduccion/que-es-vue/>. 36
- [40] Eduardo Ismael García Pérez. **¿Qué es Vue.JS?** 4 2019. <https://codigofacilito.com/articulos/que-es-vue>. 36
- [41] Wikipedia. **Nuxt.js**, 9 2022. <https://en.wikipedia.org/wiki/Nuxt.js>. 36, 81
- [42] Nigmacode. **Qué es NuxtJS y Por qué utilizarlo**, 2022. <https://www.nigmacode.com/javascript/que-es-nuxtjs/>. 36
- [43] Pedro Jiménez Hontanilla. **Qué es Nuxt.js**. 12 2019. <https://openwebinars.net/blog/que-es-nuxtjs-framework-de-vuejs/>. 37

- [44] Wikipedia. **Docker (software)**, 10 2022. [https://es.wikipedia.org/wiki/Docker\\_\(software\)](https://es.wikipedia.org/wiki/Docker_(software)). 37
- [45] RedHat. **¿Qué es Docker?**, 1 2018. <https://www.redhat.com/es/topics/containers/what-is-docker>. 37
- [46] Bob Reselman. **Kubernetes vs Docker Compose: What's the difference?** 11 2021. <https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/What-is-Kubernetes-vs-Docker-Compose-How-these-DevOps-tools-compare>. 37
- [47] Rubenfa. **MongoDB: qué es, cómo funciona y cuándo podemos usarlo (o no)?** 2 2014. <https://www.genbeta.com/desarrollo/mongodb-que-es-como-funciona-y-cuando-podemos-usarlo-o-no>. 38
- [48] Wikipedia. **Node.js**, 9 2022. <https://es.wikipedia.org/wiki/Node.js>. 38
- [49] Node.js. **Node.js**, 2022. <https://nodejs.org/es/about/>. 38
- [50] Tomislav Capan. **Why the Hell Would I Use Node.js? A Case-by-Case Tutorial**. 3 2022. <https://www.toptal.com/javascript/why-the-hell-would-i-use-node-js>. 39
- [51] Erlang. **EMQX**, 5 2021. <https://en.everybodywiki.com/EMQX>. 39
- [52] Wikipedia. **Comparison of MQTT implementations**, 10 2022. [https://en.wikipedia.org/wiki/Comparison\\_of\\_MQTT\\_implementations](https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations). 39
- [53] AutSoft. **Choosing an MQTT broker for your IoT project**. 6 2021. <https://autsoft.net/choosing-an-mqtt-broker-for-your-iot-project/>. 39
- [54] Federico Toledo. **API testing con Postman y SoapUI**. 8 2020. <https://www.federico-toledo.com/api-testing-con-postman-y-soapui/>. 39
- [55] Gustavo Romero. **Cómo realizar pruebas automatizadas con Postman**. 6 2021. <https://www.encora.com/es/blog/como-realizar-pruebas-automatizadas-con-postman>. 40
- [56] Mongoose. **Schemas**, 2022. <https://mongoosejs.com/docs/guide.html>. 91
- [57] zmstone. **EMQX**, 10 2022. [https://github.com/emqx/emqx/tree/main-v4.3/apps/emqx\\_auth\\_mongo](https://github.com/emqx/emqx/tree/main-v4.3/apps/emqx_auth_mongo). 101
- [58] EMQX. **Cluster**, 2022. <https://www.emqx.io/docs/en/v4.3/advanced/cluster.html>. 103

- [59] MQTT. **MQTT: The Standard for IoT Messaging**, 2022. <https://mqtt.org/>.
- [60] PaesslerAG. **¿Qué es MQTT?**, 2022. <https://www.paessler.com/es/it-explained/mqtt>.
- [61] Wikipedia. **Interfaz de programación de aplicaciones**, 9 2022. [https://es.wikipedia.org/wiki/Interfaz\\_de\\_programaci%C3%B3n\\_de\\_aplicaciones](https://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones).
- [62] Wikipedia. **Framework**, 3 2022. <https://es.wikipedia.org/wiki/Framework>.
- [63] TicPortal. **Base de datos SQL**. 9 2022. <https://www.ticportal.es/glosario-tic/base-datos-sql>.
- [64] MongoDB. **¿Qué es MongoDB?**, 2022. <https://www.mongodb.com/es/what-is-mongodb>.
- [65] SaaSHub. **EMQX VS mosquitto**, 2022. <https://www.saashub.com/compare-emqx-vs-mosquitto>.
- [66] DataDog. **Monitor your IoT devices and backend services in a single unified platform**, 2022. [https://www.datadoghq.com/dg/monitor/iot/?utm\\_source=advertisement&utm\\_medium=search&utm\\_campaign=dg-google-iot-emea&utm\\_keyword=iot%20dashboard&utm\\_matchtype=p&utm\\_campaignid=15832880567&utm\\_adgroupid=134676418764&gclid=CjwKCAjw3K2XBhAzEiwAmmgrAsOd\\_1Fbf94U3pL9nvBqQAZyRuthdbTZUiYInwqq-35vo3He98ixVRoC8sYQAvD\\_BwE](https://www.datadoghq.com/dg/monitor/iot/?utm_source=advertisement&utm_medium=search&utm_campaign=dg-google-iot-emea&utm_keyword=iot%20dashboard&utm_matchtype=p&utm_campaignid=15832880567&utm_adgroupid=134676418764&gclid=CjwKCAjw3K2XBhAzEiwAmmgrAsOd_1Fbf94U3pL9nvBqQAZyRuthdbTZUiYInwqq-35vo3He98ixVRoC8sYQAvD_BwE).
- [67] KaaIoT. **IoT Dashboards**, 2022. <https://www.kaaiot.com/iot-dashboards>.
- [68] EMQX. **Plugins**, 2022. <https://www.emqx.io/docs/en/v4.3/advanced/plugins.html#list-of-plugins>.

