MÁSTER EN TECNOLOGÍAS Y APLICACIONES EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

UNIVERSIDAD DE ALMERÍA

ESCUELA SUPERIOR DE INGENIERÍA

MI-FIWARE, un método de desarrollo de componentes web para FIWARE mediante microservicios

Curso 2019/2020

Alumno/a:
Juan Alberto Llopis Expósito

Director/es:
Luis Fernando Iribarne Martínez
Javier Criado Rodríguez

Agradecimientos

Voy a ser breve, aunque espero en un futuro poder dar unos agradecimientos más extensos.

Primero de todo, gracias a mis directores, Luis Iribarne y Javier Criado por su apoyo, tutorización y por confiar en mi para llevar a cabo este proyecto.

Gracias a los profesores del máster por sus enseñanzas, gracias a ellos el máster se ha hecho mucho más ameno y he logrado cosas que no habría conseguido de otra manera.

Por último, gracias a mi familia y amigos por su apoyo constante.

Resumen

Dentro de la vertiente de Smart (industria inteligente, hogar inteligente, ciudad inteligente, etc.) el concepto de ciudades inteligentes es el que mayor crecimiento está presentando, ocasionando que haya un mayor número de desarrolladores enfocados en llevar a cabo esta idea y un incremento en las empresas que contribuyen con herramientas, servicios y plataformas.

Para apoyar el auge de las ciudades inteligentes se propone la creación de un conjunto de microservicios, MI-FIWARE (referente al uso de microservicios, MI, utilizando FIWARE), que libere al desarrollador de la carga de trabajo del back-end y del tratamiento de la información de contexto.

Palabras clave: Ciudad inteligente, Microservicio, Componente Web, FIWARE.

Abstract

Within the Smart solutions (Smart Industry, Smart Home and Smart City), the concept of Smart Cities is the one having the greatest growth, inducing an increment in the number of developers focused on carrying out this idea, and a growth in the number of companies that contributes with tools, services and platforms.

To support the boom of smart cities, the creation of a set of microservices, MI-FIWARE (referring to the use of microservices, MI, using FIWARE), is proposed to free the developer from the back-end workload and from the processing of context information.

Keywords: Smart City, Microservice, Web Componente, FIWARE.

Índice

Índice de figuras

1. Introducción

IoT (Internet of Things, Internet de las Cosas) se trata de un concepto utilizado por una pequeña parte de la población española, un concepto que para la mayoría de los ciudadanos no tiene futuro [1]. Pero no más lejos de la realidad, IoT lleva siendo durante varios años una vertiente en crecimiento, llegando en 2019 a existir más de 8.000 millones de dispositivos IoT activos (véase Figura 1) [2].

El último informe acerca de la población mundial indica que en 2019 la población rondaba los 7.600 millones de personas [3], es decir, existen más dispositivos IoT que personas en el mundo.

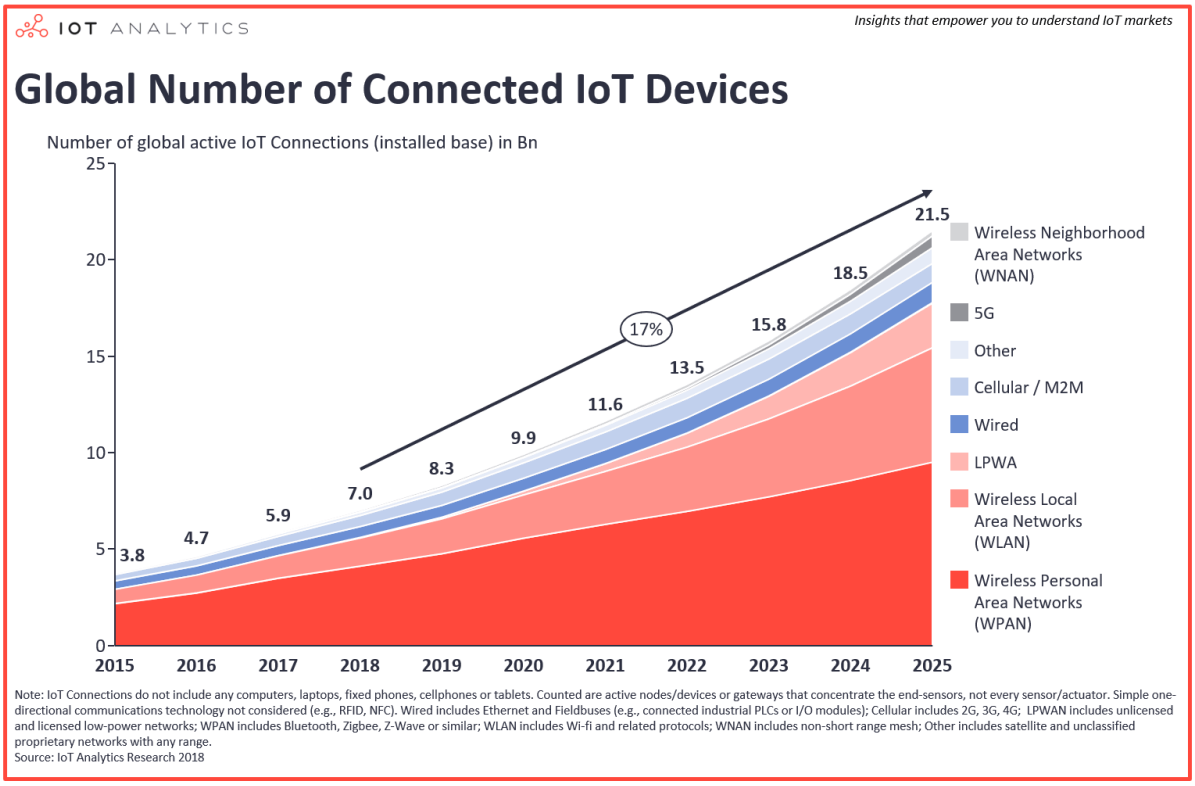

Figura 1. Número de dispositivos IoT activos por año [2].

En el caso de que el número de dispositivos no sea un dato que resulte suficiente para justificar el auge de IoT, si estudiamos las cifras de capital que está moviendo actualmente la industria, según IDC (International Data Corporation), IoT está moviendo un total de 726 miles de millones de dólares estadounidenses, con vistas a superar el billón de dólares estadounidenses para 2022 [4].

Si hablamos de gastos por país, los países que más invierten en IoT actualmente son: Estados Unidos y China, con gastos de 194 y 182 miles de millones, respectivamente [5].

Este aumento de uso de dispositivos IoT se ha visto impulsado por la Industria 4.0, las Smart Home u hogares inteligentes y las Smart Cities o ciudades inteligentes, siendo en los tres casos el objetivo de uso de dispositivos IoT el facilitar el desempeño de una serie de actividades. Por ejemplo, en la Industria 4.0, gracias al uso de dispositivos IoT se ha facilitado la obtención de información de las distintas secciones dentro del proceso de fabricación mediante la intercomunicación entre los distintos sistemas, como pueden ser los sistemas de control de calidad y la cadena de montaje (véase Figura 2).

En resumen, facilita la obtención de información fiable y en tiempo real de todo el proceso de fabricación, lo que ayuda tanto a la fabricación realizada en un momento concreto como a la toma de decisiones para la fabricación futura [6].

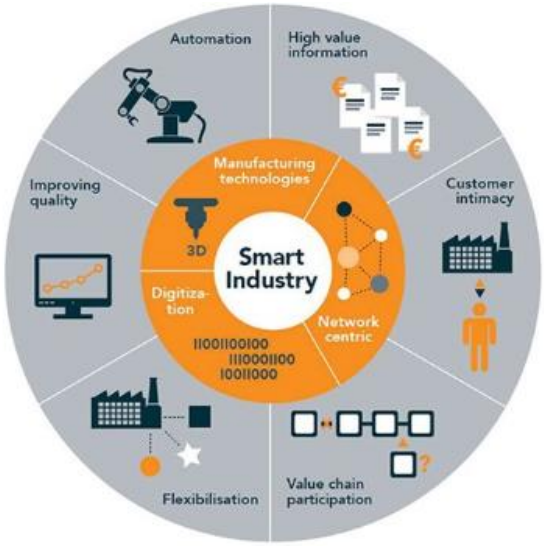

Figura 2. Usos de la Industria 4.0 [7].

Dentro de la vertiente de Smart (industria inteligente, hogar inteligente, ciudad inteligente, etc.) el concepto de ciudades inteligentes es el que mayor crecimiento está presentando (véase Figura 3) [8], ocasionando que haya un mayor número de desarrolladores enfocados en llevar a cabo esta idea, con ejemplos como SmartPort [9] y FI-WARE Smart City Light [10], y un incremento en las empresas que contribuyen con herramientas, servicios y plataformas. Un ejemplo de plataforma es FIWARE, aunque existen también otras plataformas como, por ejemplo, AWS IoT, Azure IoT Suite y Google Cloud IoT, plataformas que se tratarán en las siguientes subsecciones del trabajo.

Para apoyar el auge de las ciudades inteligentes se propone la creación de un conjunto de microservicios, MI-FIWARE (referente al uso de microservicios, MI, utilizando FIWARE), que libere al desarrollador de la carga de trabajo del back-end y del tratamiento de la información de contexto.

Al ser liberado de esa carga, el desarrollador deberá enfocarse únicamente en el front-end, donde también se le ofrecerá apoyo mediante la definición de una estructura de componentes web y, el utilizar el microservicio desarrollado, permitirá al desarrollador hacer uso de datos en tiempo real, además de otras características adicionales como un control de acceso o un histórico de datos.

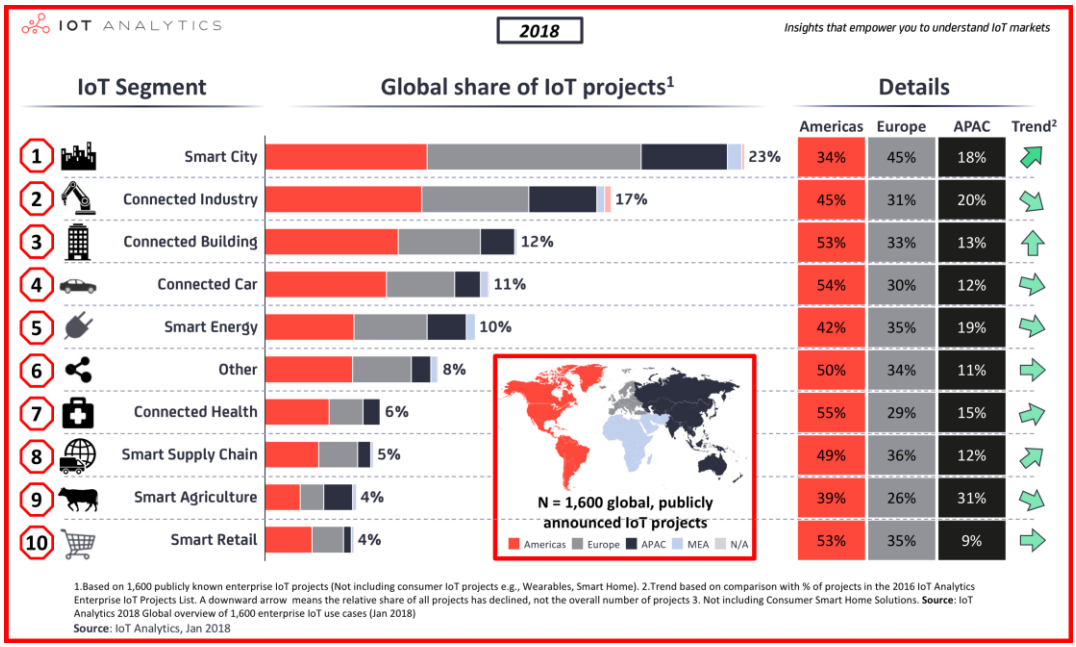

Figura 3. Proyectos IoT por sector y continente [8].

Cuando un desarrollador quiere implementar una aplicación que haga uso de información de sensores, asumiendo que el bróker de contexto ya se encuentra configurado y funcionando, debe realizar los siguientes pasos:

- Realizar un análisis del problema.
- Desarrollar una API:
  - Desarrollar funciones para el procesamiento de los datos.
  - Configurar la conexión con el bróker de contexto.
  - Desarrollar los puntos de acceso.
- Desarrollar la aplicación:
  - Desarrollar los componentes.
  - Configurar la conexión con la API.
- Implementar la comunicación de datos en tiempo real.

El objetivo de la propuesta es dar soporte a este proceso de manera que los pasos que deban acometerse sean:

- Desarrollar la aplicación:
  - Desarrollar los componentes.
- Indicar la dirección del bróker de contexto y del microservicio.

De esta manera el desarrollador es liberado de gran parte de las tareas que debía implementar (véase Figura 4), teniendo solo que desarrollar los componentes con ayuda de la estructura propuesta e indicar al microservicio y al componente las direcciones del bróker de contexto y del microservicio respectivamente para permitir la comunicación entre ellos.

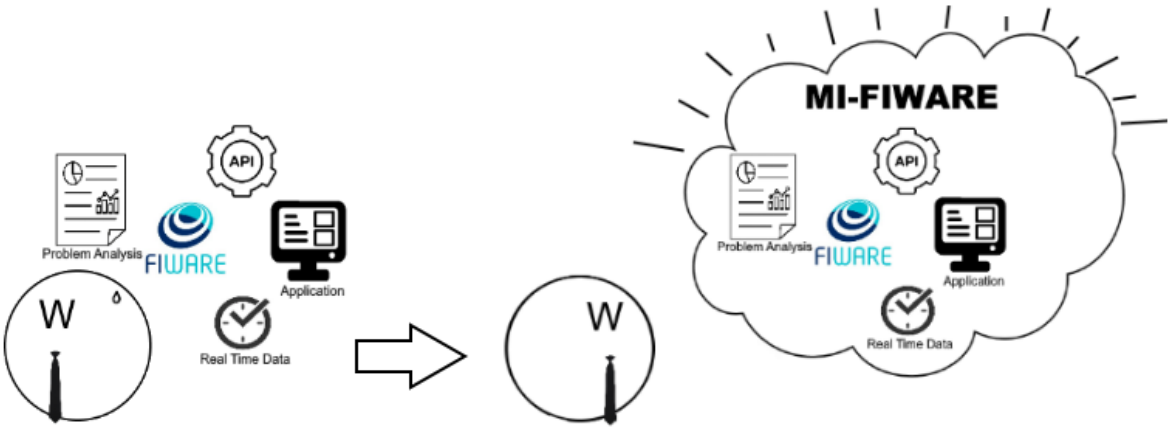

Figura 4. Ejemplo de concepto de la liberación de carga de trabajo de MI-FIWARE.

1.1. Fases de desarrollo y cronograma

El proyecto, por su base en la investigación, se comienza el día 24 de julio y se finaliza el día 2 de febrero. Estando dividido en dos fases, el objetivo de la primera fase es la investigación y el objetivo de la segunda fase el desarrollo de la propuesta junto a un caso de estudio.

Las fases desglosadas son las siguientes:

- Fase de investigación:
  - Pesquisas iniciales del estado del arte.
  - Estructuración del documento.
  - Realización e investigación de la fase 1, introducción y FIWARE.
  - Realización e investigación de la fase 2, previa a la propuesta y propuesta.
- Anteproyecto.
- Fase de desarrollo:
  - Autoaprendizaje de la plataforma FIWARE.
  - Preparación y configuración del entorno de trabajo.
  - Desarrollo de la primera versión del microservicio.
  - Desarrollo de un primer componente web que haga uso del microservicio.
  - Desarrollo de un caso de estudio y segunda versión del microservicio.
- Documentación final y conclusiones.

En la Figura 5 se muestran los tiempos de las distintas fases junto a la duración de cada una.

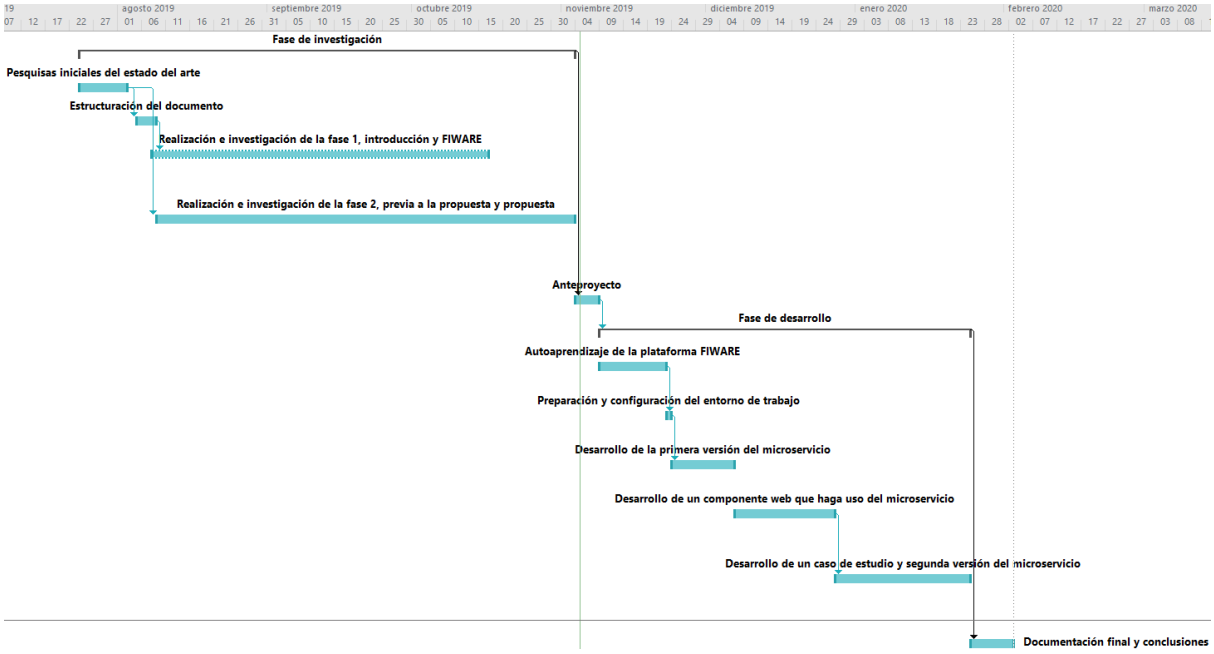

Figura 5. Planificación general del proyecto.

El total de días de trabajo es de 140 a una media de 3 horas diarias, lo que da como resultado un total de 420 horas de las 300 recomendadas para su desarrollo. La mayor parte del tiempo se ha dedicado a la investigación, ya que es la base de este trabajo, teniendo primero que adquirir conocimientos acerca del tema a desarrollar y de temas relacionados, para posteriormente documentarse con el objetivo de poder obtener unas conclusiones de lo previamente estudiado. A partir de las conclusiones, se ha obtenido una idea de mejora que resuelva o ayude a resolver problemas identificados en las lecturas.

Una vez terminada la parte investigadora se han elaborado pruebas de concepto y comprobado la viabilidad de la propuesta, que al ser algo novedoso y único podía llevar más tiempo del planificado. A continuación, se desarrolló un ejemplo sencillo para mostrar en la práctica la utilidad de la propuesta y enseñar su funcionamiento. Para ello se hizo el desarrollo de una primera versión que contenía los conceptos básicos y necesarios del sistema para cumplir las condiciones propuestas.

Una vez desarrollada esta primera versión, se hizo el desarrollo de una segunda versión que contenía funcionalidades o características adicionales que le daban un valor añadido al sistema.

Tras haber finalizado el desarrollo, se terminó de redactar el documento, se estilizó y revisó antes de su fecha de entrega.

1.2. Ciudades inteligentes

Smart Cities o ciudades inteligentes es un concepto que lleva existiendo desde principio de los 90 debido a que la población en las ciudades es cada vez mayor, y con ello la información necesaria para gestionar todos los servicios ofrecidos correctamente [11]. En este punto, los distintos gobiernos buscan la manera de solucionar y controlar el flujo de información para permitir el continuo crecimiento de la población en las ciudades [12].

Por ejemplo, en España según el Instituto Nacional de Estadística, en 2015 un 79,7% de la población se concentraba en las ciudades, mientras que solamente un 20,3% de la población ocupaba las zonas rurales del país (véase Figura 6) [13].

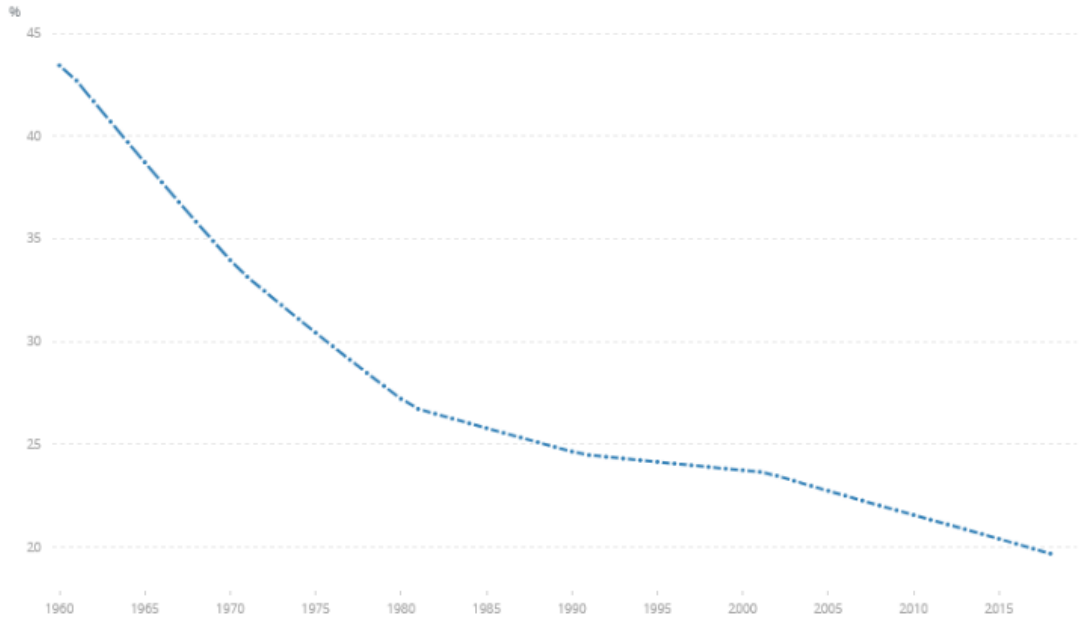

Figura 6. Porcentaje de la población rural en España [14].

Con el fin de mejorar la comunicación surge este concepto, ciudades inteligentes, que utilizando la tecnología de comunicaciones busca simplificar el flujo de información en la ciudad e incluso entre ciudades, y de facilitar la gestión de la ciudad, ocasionando con ello un incremento en la economía. Este flujo de información siempre se debe mantener y mover con precaución, puesto que se trata de una información muy sensible y codiciada por los cibercriminales [15].

Dentro del concepto de ciudades inteligentes hay dos vertientes, la del sector privado y la del sector público porque algunas empresas, debido a su tamaño, necesitan llevar a cabo un desarrollo equivalente al de una pequeña ciudad.

Para llevar a cabo el desarrollo de una ciudad inteligente disponemos de tres maneras de afrontar el problema: contratar a una empresa especializada para que se encargue de todo, realizar el desarrollo utilizando una plataforma destinada a la creación de soluciones para ciudades inteligentes, o utilizar una serie de herramientas que permitan realizar el proyecto.

En las subsecciones siguientes, se van a explicar los distintos tipos de desarrollo con el objetivo de mostrar las ventajas y limitaciones de cada una de ellas, características que se expondrán en una tabla comparativa en secciones posteriores.

1.2.1. Desarrollo por servicios

El desarrollo mediante servicios consiste en la contratación de una empresa externa, generalmente una consultora, para que se encargue de todo el proceso de desarrollo, este concepto es conocido con el nombre de *outsourcing*.

La empresa externa, trabajando mano a mano con la empresa o institución que la contrató, realiza todo el proceso necesario para convertir la institución o empresa en una ciudad inteligente, ofrece formación a sus empleados y finalmente por un coste anual da soporte a la empresa o institución.

Este tipo de desarrollo es el más solicitado [16] debido a que es necesario formar a los ingenieros para llevar a cabo estos desarrollos [17] y las empresas que ofrecen estos servicios. Además de disponer de experiencia previa, dan la posibilidad de formar a los ingenieros para que puedan mantener el producto final, ya sea tras el desarrollo o durante el mismo [18].

Las empresas más importantes que ofrecen este tipo de servicio son IBM [19], CISCO [20] y SAP [21]; empresas líderes dentro del ámbito de las tecnologías de la información [17]. Por ejemplo, Madrid realizó su desarrollo de ciudad inteligente con la ayuda de IBM [22] y Barcelona con ayuda principalmente de CISCO [23].

Pero, el desarrollo por servicios dispone de un gran inconveniente respecto al resto de desarrollos. Ambas partes deben de trabajar juntas como si fueran una única empresa. Si las empresas no consiguen ponerse de acuerdo, el proyecto fracasará y se perderán los recursos dedicados al proyecto.

La incapacidad de trabajar juntos es la mayor causa de fracasos en el desarrollo por servicios, una mala comunicación, una diferencia cultural, incluso la diferencia horaria entre ambas partes puede terminar con el fracaso del proyecto [24]. Para evitar esta situación ambas empresas deben de realizar pasos previos al desarrollo con el objetivo de entenderse mejor y no dar lugar al abandono del proyecto.

1.2.2. Desarrollo por plataformas

El desarrollo mediante plataformas consiste en el uso de un ecosistema software que permita desarrollar un proyecto de ciudad inteligente y que proporcione una serie de estándares, normas o pasos a seguir para realizarlo correctamente.

Existe multitud de plataformas IoT, especialmente en la nube. Cada plataforma se centra en un dominio específico (véase Figura 7) y, en función del dominio sobre el que actúe, presentará una serie de características para resolver las necesidades de ese dominio.

Cuanto más específico sea el dominio, mayor utilidad tendrá para resolver problemas referentes a ese dominio, por ejemplo, ThingSpeak [25] se centra en el desarrollo de aplicaciones y en la monitorización y análisis de datos. La especialización en estos dos dominios permite a la plataforma ser fácil de utilizar y explotar por parte del consumidor.

Application domain specific IoT clouds.

IoT cloud platforms	Domains									
	Application development	Device management	System management	Heterogeneity management	Data management	Analytics	Deployment Management	Monitoring management	Visualization	Research
Aer cloud			√			√		+		
Arkessa		√			+					
Arrayant connect	√	√		+	√					
Axeda		√			+			√		
Ayla's cloud fabric		√	+						√	
Carriots	+	√						√		
Echelon	√	√					+			
Etherios		+						√		
Exosite		√	+					√		
GroveStreams	√							√	+	
IBM IoT		√								+
InfoBright					√	+				
Jasper Control Centre	√					+		√		
KAA	+				√					
Microsoft research lab of things	√									+
Nimbits					+	√				
Oracle IoT cloud			√	√	+		√		√	
OpenRemote	√			+						
Plotly						√		√	+	
SeeControl IoT		+				√			√	
SensorCloud		+						√	√	
Temboo	+									
Thethings.io	√		+					√		
ThingSpeak	√							+	√	
ThingWorx	√				+			√		
Xively	√	+						√		

Abbreviation: + suitable, √ applicable.

Figura 7. Listado de plataformas IoT con sus dominios [26].

Este documento se centra en plataformas que den soporte a un gran número de dominios debido a que, para el desarrollo de soluciones destinadas a ciudades inteligentes, se debe tener en cuenta el proceso desde la captación de datos por parte de los dispositivos hasta el consumo de los datos por el usuario final.

Algunos ejemplos son Microsoft con Azure IoT [27], Amazon con AWS IoT [28], Google con Google Cloud IoT [29] y FIWARE [30], que podría considerarse una plataforma, pero posee características adicionales que lo diferencian del resto.

Las plataformas normalmente proporcionan una serie de pautas más parecidas a una serie de pasos que a un estándar. Para cada una de las fases de consumo de datos, nos indican (1) los posibles servicios a utilizar de su plataforma, (2) los servicios con los que se puede conectar e interactuar y (3) el flujo de datos que se formará al utilizar y conectar el servicio (véase Figura 8).

En cambio, FIWARE proporciona un estándar a la hora de trabajar con dispositivos IoT dando a los desarrolladores un amplio catálogo de componentes preparados para comunicarse entre sí y centrados principalmente en el desarrollo para ciudades inteligentes.

Entre estas plataformas la más usada es la de Amazon, seguida de la plataforma de Microsoft. Por último, se encontraría la plataforma de Google [30]. Este orden se debe a dos factores: fecha de lanzamiento (la primera plataforma en lanzarse fue la de Amazon, seguida de las plataformas de Microsoft y la de Google) y documentación (Amazon, Microsoft y Google proporcionan un amplio repositorio de documentación, pero el acceso a la documentación práctica de Google Cloud IoT no es fácil, provocando que algunos desarrolladores no puedan localizarla).

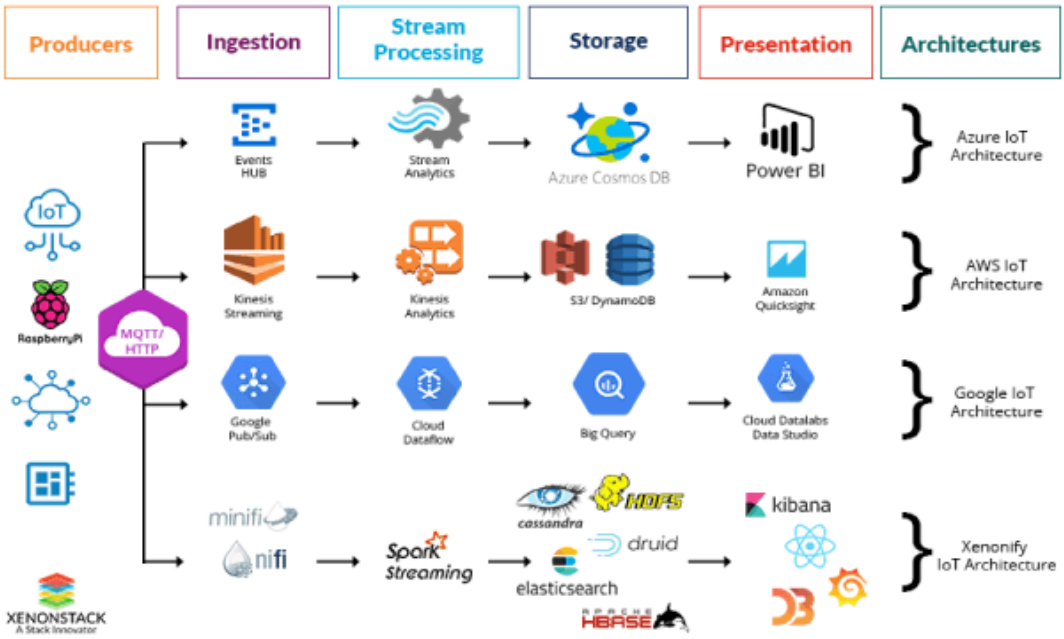

Figura 8. Lista de herramientas y plataformas para cada fase de consumo de datos IoT [31].

1.2.2.1. AWS IoT

Amazon, como una de las tres grandes potencias en la nube, ha lanzado su propia plataforma IoT enfocada en la Industria 4.0, hogares y ciudades inteligentes.

Dentro de su plataforma se ofrecen tres grupos de herramientas (véase Figura 9), donde cada grupo equivale a un dominio: herramientas destinadas a servicios de control y conectividad de los dispositivos desplegados, herramientas destinadas a servicios de datos para tratar y analizar los datos obtenidos de los dispositivos, y herramientas destinadas al software de dispositivos para facilitar su gestión usando software propio de Amazon [28].

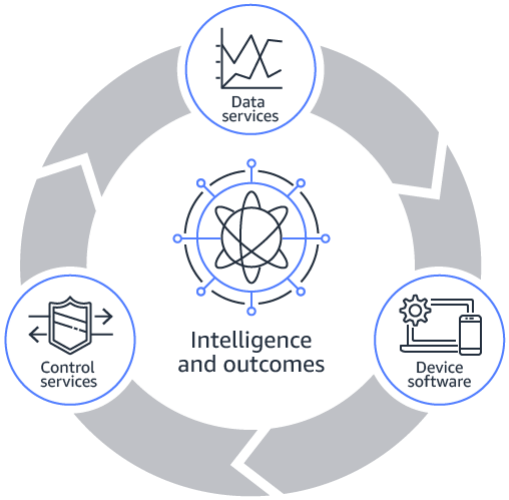

Figura 9. Grupos de herramientas de AWS IoT [28].

Amazon no solo proporciona las herramientas, sino que da unas pautas de como conectar y utilizar los tres grupos de herramientas disponibles para poder desarrollar servicios propios [32].

Además, proporciona una serie de soluciones IoT ya implementadas, así como una guía para implementarlas [33].

Un ejemplo de uso de AWS IoT es el de un sensor de humedad y temperatura, donde un dispositivo capta y envía datos al servicio de Amazon IoT Core al que se suscriben los clientes para obtener la información [34].

Amazon proporciona múltiples servicios dentro del desarrollo de soluciones IoT y el desarrollador elige los servicios que usará, de qué manera conectarlos y si seguir los pasos que proporciona Amazon o establecer una serie de pasos con herramientas externas al servicio de Amazon.

Otro ejemplo, es el de un servicio para el control de tráfico y de puntualidad del transporte público, buscando utilizar todas las características de AWS IoT con el objetivo de comprobar su uso a gran escala; para este ejemplo se hace uso de varios componentes IoT de Amazon [35].

1.2.2.2. Azure IoT Suite

Microsoft, ante el constante crecimiento de IoT, ha lanzado una serie de herramientas englobadas en una plataforma llamada Azure IoT Suite [36], con el objetivo de ayudar en el desarrollo de software centrado en IoT.

Esta plataforma proporciona una serie de pasos a seguir y herramientas a utilizar durante todas las fases del desarrollo del proyecto (véase Figura 10). No es un estándar, pero ayuda a los desarrolladores a seguir unas pautas comunes durante el proceso de desarrollo.

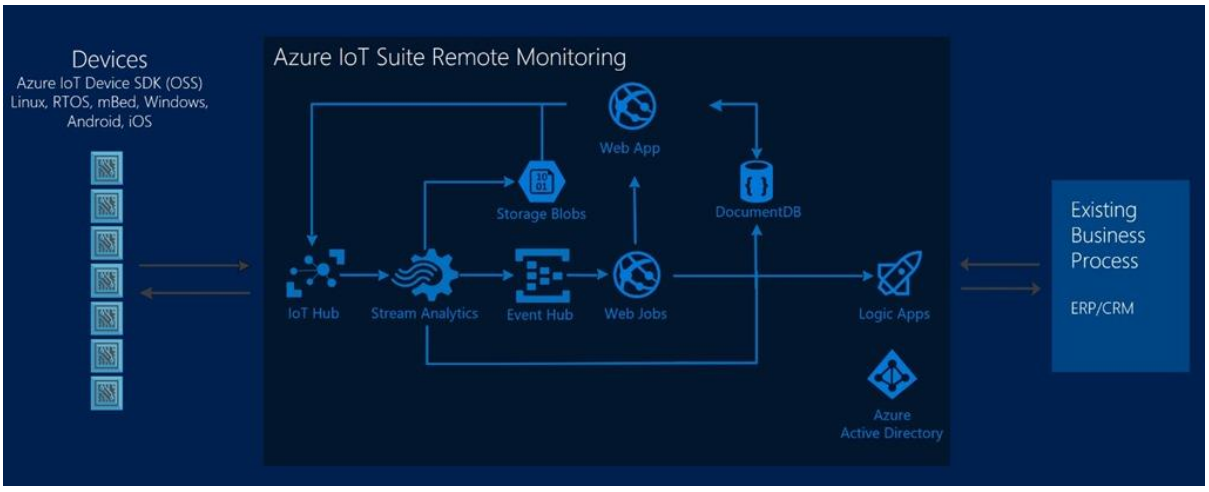

Figura 10. Flujo de servicios de Azure IoT Suite [37].

Al igual que en Amazon, el desarrollador puede decidir si seguir estas pautas o establecer unas propias mediante el uso de herramientas externas a la plataforma proporcionada por Microsoft.

Estas herramientas permiten desde la gestión de los dispositivos físicos hasta el tratamiento y gestión de los datos [37], incluyendo funcionalidades como el lanzamiento de eventos según la información obtenida de los dispositivos [38].

Dentro de este conjunto de herramientas existe una herramienta llamada Azure Logic App Designer [39] que permite al usuario establecer el flujo de información, las acciones y los desencadenantes del sistema. Azure Logic App Designer no ofrece la misma libertad que el desarrollo por herramientas, pero sirve como alternativa para soluciones más simples y menos específicas en las que no se posee mucho tiempo o personal cualificado y se necesita una solución robusta y segura.

A causa de la gran cantidad de servicios que proporciona Azure [16] junto a estas herramientas, Azure IoT Suite se trata de un servicio que permite poder desarrollar en poco tiempo software seguro y de calidad destinado a las ciudades inteligentes, algo con lo que pocas plataformas pueden competir, ya que la seguridad de los datos es un problema en el desarrollo de sistemas que hagan uso de dispositivos IoT y comuniquen datos por la red [40].

Un ejemplo de caso de estudio es el del desarrollo de un sistema de recolección automática de residuos urbanos y su integración con la infraestructura de la ciudad inteligente, realizado en Rumanía, mediante el uso de los servicios proporcionados por Azure IoT Suite (véase Figura 11) [41].

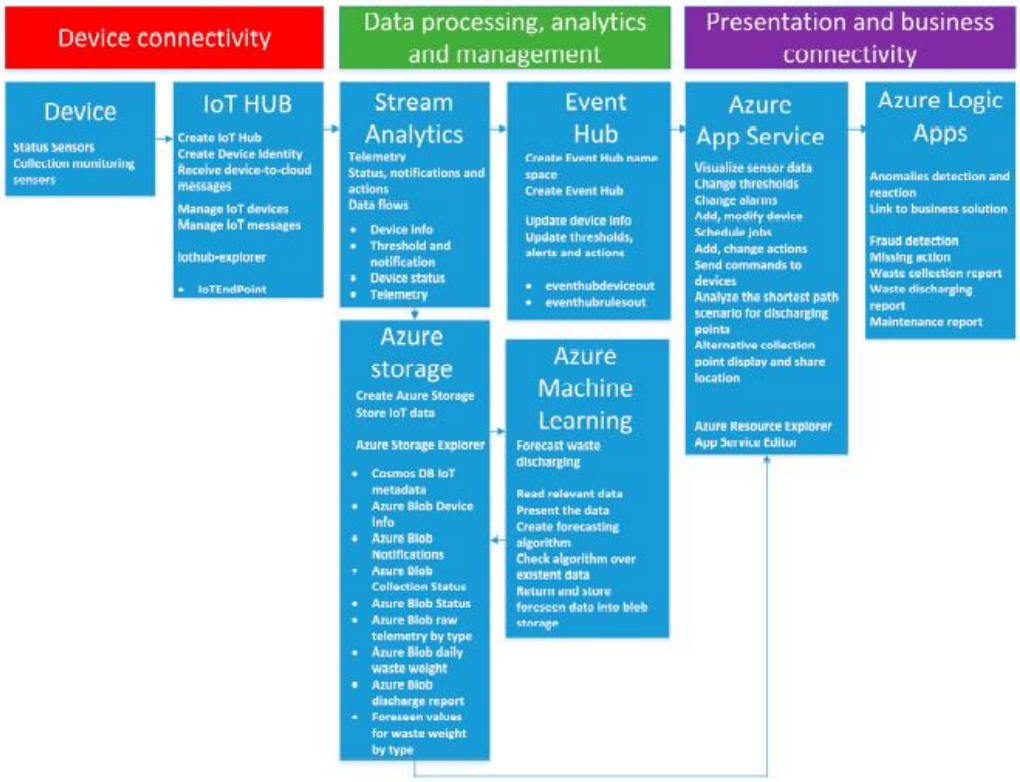

Figura 11. Arquitectura propuesta para el sistema de recolección automática de residuos urbanos [41].

Según el autor, las herramientas y servicios proporcionados por Azure dan una clara ventaja respecto a otras herramientas y servicios existentes, garantizando un mínimo de seguridad y un escalado y rápido desarrollo, pero a cambio trae una serie de inconvenientes [41].

El hecho de que sea tan sencillo y rápido desarrollar soluciones IoT provoca que haya poco control del código fuente, lo que ocasiona que, a la hora de implementar características muy específicas, se pueda producir su ralentización o no puedan ser implementadas.

Las herramientas proporcionadas por Microsoft son muy útiles para empezar o para realizar pequeños desarrollos, pero para un gran proyecto con características muy específicas y con un bajo presupuesto no es una buena solución. Por otro lado, Azure IoT Suite tiene la ventaja de proporcionar formación además de soporte.

1.2.2.3. Google Cloud IoT

Google apuesta por su nube añadiéndole una nueva serie de funcionalidades, llamadas Google Cloud IoT [29], para permitir el desarrollo de soluciones IoT. Este servicio, al igual que AWS IoT y Azure IoT Suite, ofrece una serie de herramientas y pautas durante todas las fases de consumo de datos (véase Figura 12) que pueden seguirse o ser modificadas haciendo uso de herramientas externas, según las necesidades del desarrollador.

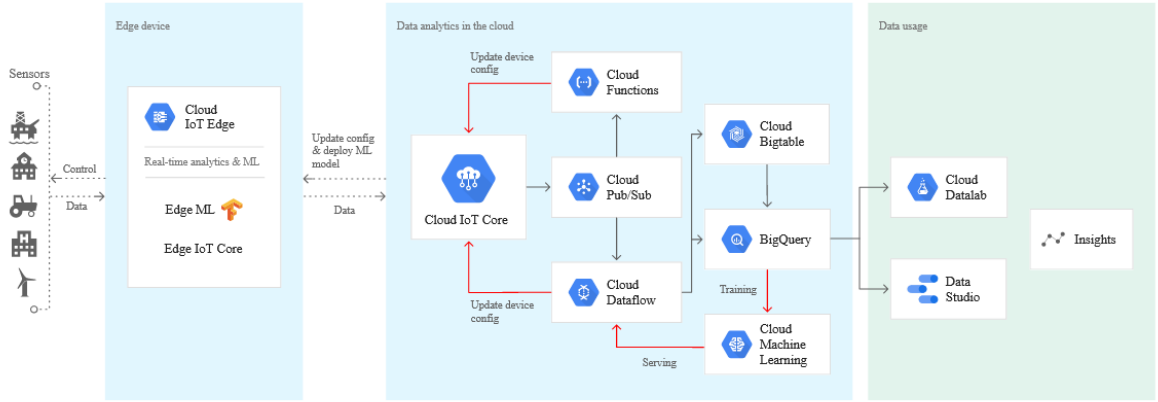

Figura 12. Ejemplo de flujo de servicios de Google Cloud IoT [42].

Las herramientas que proporciona incluyen mejoras en la seguridad, control de los dispositivos mediante el uso de puntos de acceso gestionados por sus sistemas, análisis, procesamiento y explotación de datos.

Google Cloud IoT posee una gran cantidad de documentación práctica muy bien definida en la que se va guiando al usuario paso a paso a través de todas las funcionalidades de la herramienta, indicando en todos los pasos de la documentación los recursos necesarios, el coste de los recursos, ejemplos y soporte en caso de que la información existente no sea suficiente.

1.2.3. Desarrollo por herramientas

El desarrollo por herramientas proporciona una serie de herramientas que deben de ser integradas haciendo uso de uno o varios lenguajes de programación para poder implementar soluciones IoT.

Este tipo de desarrollo se lleva a cabo mediante el uso de herramientas como las desarrolladas por Eclipse [43]: Kura [44] y Kapua [45].

Eclipse [43] proporciona un conjunto de herramientas gratuitas para el desarrollo de soluciones IoT [46], herramientas dirigidas a la gestión de los dispositivos y de cómo se obtiene y transmite la información, por ejemplo, Eclipse Paho [47] se utiliza para la comunicación entre máquinas, Eclipse Kura [44] para la construcción de IoT gateways, Eclipse Kapua [45] como middleware entre los dispositivos IoT y las aplicaciones, y Mosquitto [48] como bróker del sistema.

Este tipo de desarrollo no proporciona unas pautas para la implementación de soluciones IoT, sino más bien una serie de herramientas que, conectadas entre sí y con el uso de otras herramientas externas (véase Figura 13), permiten desarrollar este tipo de soluciones, pudiendo personalizarlas al disponer del control completo del código.

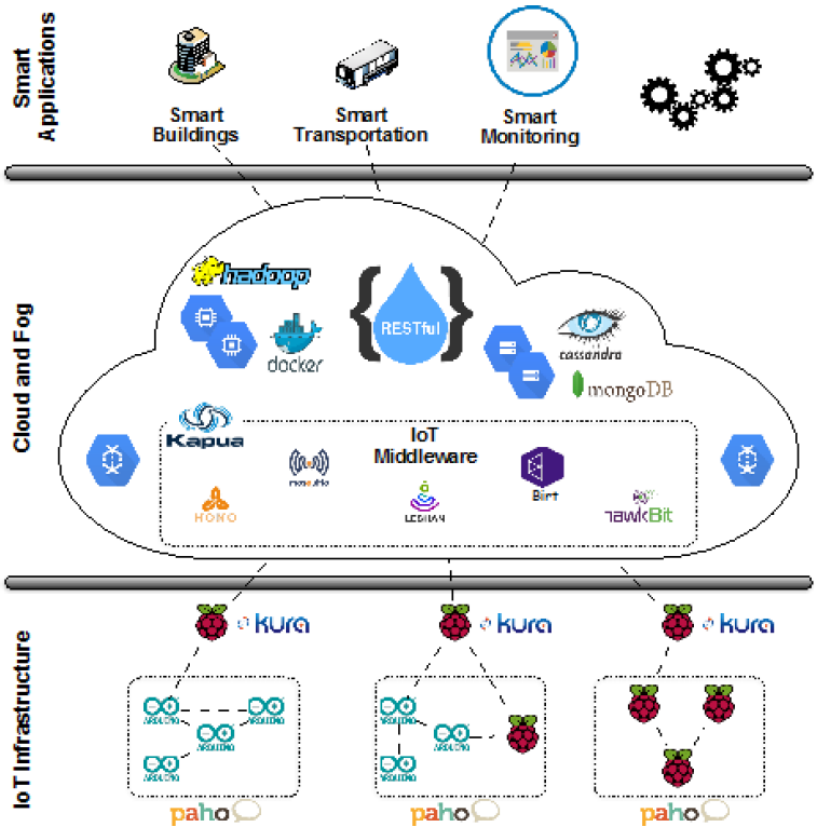

Figura 13. Ejemplo de arquitectura de consumo de datos IoT [49].

Algunos ejemplos de desarrollo utilizando las herramientas que nos proporciona Eclipse son: el desarrollo de un sistema para la obtención y visualización de los niveles de ruido en Coimbra, Portugal [49]; y la implementación de un sistema para la monitorización del pulso cardíaco, ejemplo en el que también han sido utilizados los servicios IoT de Amazon [50].

El uso de herramientas junto a plataformas es una práctica que suele llevarse a cabo, haciendo uso de las ventajas que proporciona la plataforma, como pueden ser la facilidad de implementación y la seguridad, y la libertad que proporciona el desarrollo por herramientas al tener el control total del código.

Como conclusión, Eclipse proporciona herramientas para el desarrollo de soluciones IoT, no da facilidades ni pautas para el desarrollo, por lo que depende mayormente del desarrollador. A cambio, el desarrollador tiene completa libertad para la implementación de soluciones IoT.

2. FIWARE

FIWARE es la plataforma sobre la que se centra este trabajo porque, no solo ofrece una serie de herramientas útiles para el desarrollo de soluciones IoT, sino que proporciona unas pautas de desarrollo que sirven tanto para ciudades inteligentes [51] como para la Industria 4.0 [52], aunque en este trabajo se profundiza en el entorno de ciudades inteligentes.

2.1. ¿Qué es FIWARE?

FIWARE [30] se trata de una plataforma desarrollada por el programa europeo para el desarrollo de internet del futuro (FI-PPP) [53] con el objetivo de facilitar el desarrollo de soluciones inteligentes en múltiples sectores para mejorar la calidad de vida tanto de ciudadanos como de empresas e instituciones [53].

FIWARE posee una serie de componentes llamados Generic Enablers (GEs) en su catálogo, fácilmente integrables entre sí y cuya función es la de ayudar en el desarrollo de soluciones inteligentes, provocando una reducción del tiempo de instalación y configuración del sistema [9], y permitiendo al desarrollador construir aplicaciones complejas, desde la comunicación y gestión de los dispositivos IoT, hasta el procesado y análisis de la información que se le mostrará al usuario final [54].

2.2. Estructura

Un ejemplo de diagrama de funcionamiento de la plataforma FIWARE sería el mostrado en la Figura 14, donde el agente IDAS (Internet of Things Backend Device Management) obtiene la información de los dispositivos disponibles y envía los datos al bróker de contexto Orion utilizando la API NGSI (Next Generation Service Interface), una API propuesta por FIWARE para gestionar la información de contexto: actualizaciones, consultas, registros y suscripciones [55].

A su vez, las distintas aplicaciones disponibles obtienen de Orion los datos a los que están suscritos y pueden publicar nuevos datos o actualizar los ya existentes.

A nivel del agente IDAS pueden existir más componentes que den apoyo al sistema; por ejemplo, para las localizaciones pueden enlazarse a Orion componentes como Cygnus, componente que conectado a una base de datos guarda un histórico de los datos recibidos por el bróker de contexto. En la capa de las aplicaciones se pueden enlazar componentes como Wirecloud, componente utilizado para procesar y presentar al usuario los datos recibidos del bróker de contexto.

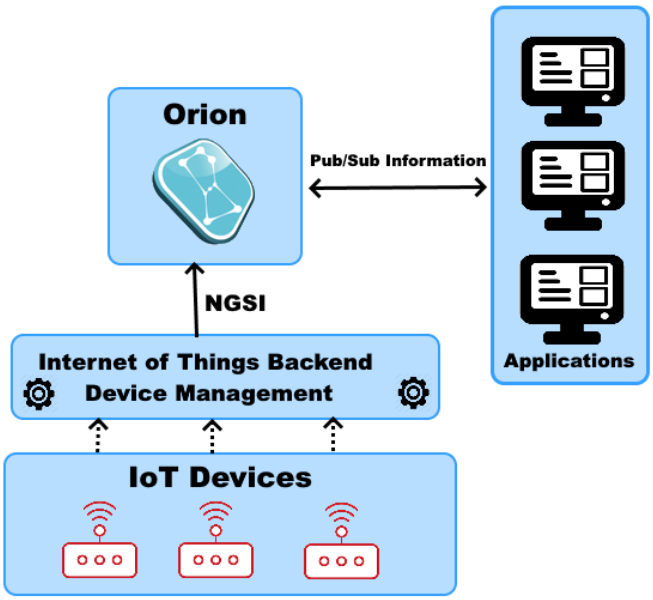

Figura 14. Diagrama del funcionamiento de FIWARE.

2.2.1. Orion

FIWARE ofrece un conjunto de componentes cuyo núcleo es Orion, su bróker de contexto, un componente obligatorio para todo sistema que quiera hacer uso de FIWARE [56]. En Orion se destaca la completa diferenciación entre los productores y los consumidores de contexto, los productores publican datos sin saber su uso final y los consumidores eligen de un conjunto de datos cuales quieren utilizar sin conocer su procedencia (véase Figura 15) [56].

Orion hace uso de la API NGSIv2, una API RESTful que permite la comunicación de los componentes del sistema con aplicaciones externas mediante un conjunto de operaciones de actualización, consulta y suscripción sobre la información de contexto.

Además, posee una característica que puede ser considerada un inconveniente en función del uso que se le aplique; solamente guarda la información del momento, es decir, si por ejemplo disponemos de un sensor de voltaje en las farolas nos muestra el valor que ese sensor está dando en ese instante [37].

Para solucionar este tipo de limitaciones, FIWARE proporciona una serie de componentes adicionales llamados Generic Enablers que aportan un valor adicional al sistema a desarrollar [57].

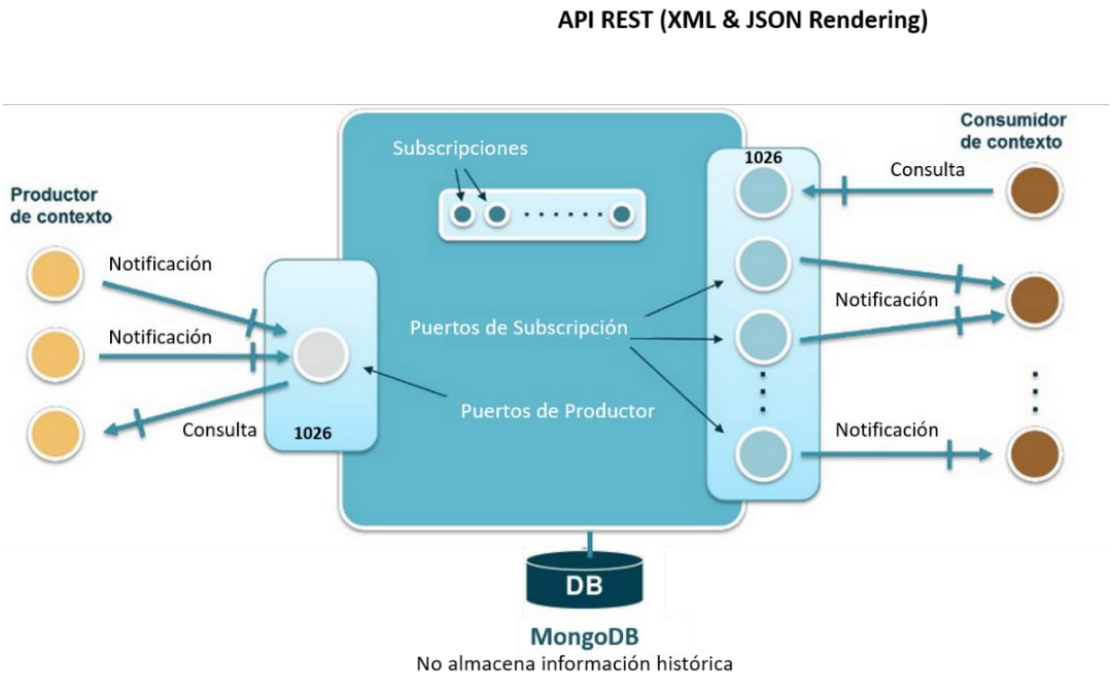

Figura 15. Funcionamiento de Orion [56].

2.2.2. Interacción con dispositivos IoT

Una vez establecidos los sistemas para tratar la información obtenida, es necesario el uso de componentes encargados de obtener la información de los sensores para que Orion pueda suscribirse y adquirir los datos generados en tiempo real.

Los Generic Enablers IDAS que proporciona FIWARE son los encargados de facilitar la comunicación con los dispositivos IoT para la obtención de información, proporcionando un conjunto de agentes cuya función es la de interactuar con la API NGSI mediante el uso de un protocolo específico.

Los agentes disponibles son los siguientes [56]:

- Agente JSON: Permite la conexión entre HTTP/MQTT (utilizando un payload JSON) y NGSI.
- Agente LWM2M: Permite la conexión entre Lightweight M2M y NGSI.
- Agente Ultralight: Permite la conexión entre HTTP/MQTT (con un payload UltraLight2.0) y NGSI.
- Agente LoRaWAN: Permite la conexión entre LoRaWAN y NGSI.
- Agente OPC-UA: Permite la conexión entre OPC Unified Architecture y NGSI.
- Agente Sigfox: Permite la conexión entre Sigfox y NGSI.
- Agente Library: Permite crear agentes propios.

2.2.3. Procesamiento y análisis de datos

FIWARE proporciona una serie de Generic Enablers que permiten procesar y analizar la información de contexto obtenida.

Algunos de estos Generic Enablers son los siguientes:

- Wirecloud: Es un Generic Enabler que utiliza la información de contexto junto a una serie de widgets, permite construir paneles con el objetivo de proporcionar información al usuario (véase Figura 16), por ejemplo, las localizaciones en un mapa de Google, o distintos tipos de gráficas a partir de los datos existentes. Este Generic Enabler puede ser útil para explotar la información de contexto de manera rápida y sencilla sin profundizar en el desarrollo de una aplicación.
- Cosmos: Se trata de un Generic Enabler para el análisis de BigData. Proporciona una serie de módulos para el almacenamiento y procesamiento de grandes volúmenes de datos.
- Knowage: Es un Generic Enabler diseñado para el análisis de datos de negocio, dispone de una serie de módulos para cubrir los requisitos de la inteligencia de negocio.
- Kurento: Simplifica la integración de elementos multimedia con los datos obtenidos con el objetivo de facilitar el trabajo a los desarrolladores.
- CKAN: Este Generic Enabler está centrado en la publicación y monetización de los datos, permitiendo establecer un acceso OAuth2, establecer permisos para distintos conjuntos de datos y añadir fácilmente vistas de Wirecloud asociadas a los datos.
- STH-Comet: Se trata de uno de los servicios de almacenamiento proporcionados por FIWARE, permite almacenar un histórico de datos. Se puede conectar mediante el uso de Cygnus.
- Cygnus: Permite conectar Orion con servicios de almacenamiento de FIWARE —CKAN, Cosmos (Hadoop) y SHT-Comet— o con servicios externos como MySQL o MongoDB para almacenar un histórico de datos de contexto.

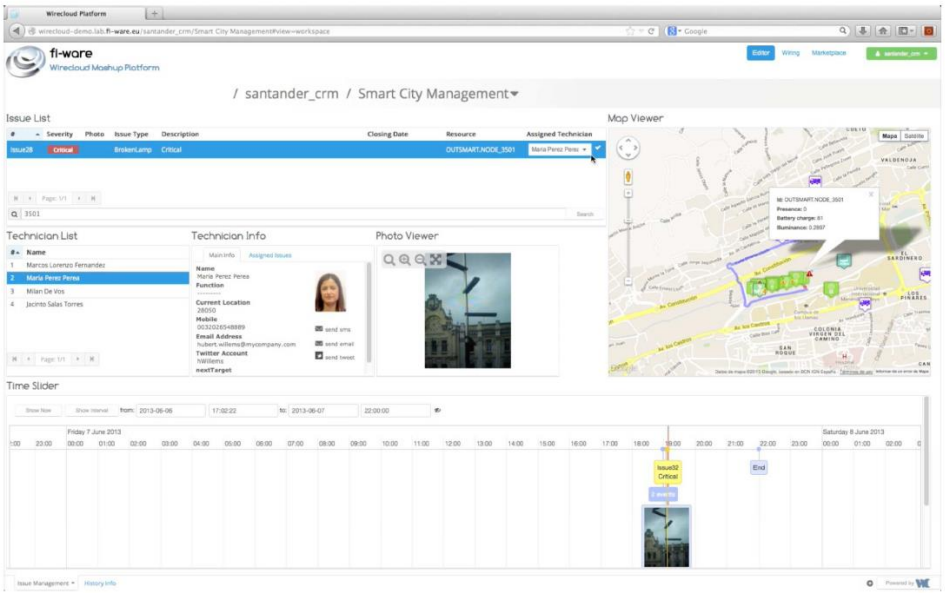

Figura 16. Aplicación Street Lamps Management [10].

2.3. Casos de estudio

FIWARE ha sido utilizado en varios casos reales como objeto de estudio para comprobar su utilidad dentro del desarrollo de soluciones para ciudades inteligentes, por ejemplo, SmartPort [9] y Street Lamps Management Application [10] son dos casos en los que se ha probado FIWARE en proyectos reales y que conllevan una mejora en la gestión de una parte de la ciudad, lo que se expone a continuación.

2.3.1. SmartPort: A Platform for Sensor Data Monitoring in a Seaport Based on FIWARE

SmartPort [9] es una herramienta diseñada con el objetivo de apoyar las distintas tareas realizadas en el puerto de Las Palmas de Gran Canaria y con el objetivo de ayudar en la toma de decisiones, por ejemplo, aportando información necesaria para la gestión de los contenedores e información en tiempo real del estado del mar.

Esta herramienta ha sido desarrollada haciendo uso del sistema FIWARE, sistema que obtiene la información de contexto de los sensores mediante el uso de un conjunto de agentes y del bróker de contexto Orion. Los datos obtenidos se van almacenando en una base de datos MongoDB utilizando Cosmos, componente que se conecta con Orion a través de Cygnus (véase Figura 17). También dispone de un sistema de alertas gestionadas por Orion y almacenadas en una base de datos MySQL. Por último, las aplicaciones acceden a los datos de Orion y a las alertas a través de una API creada por los desarrolladores.

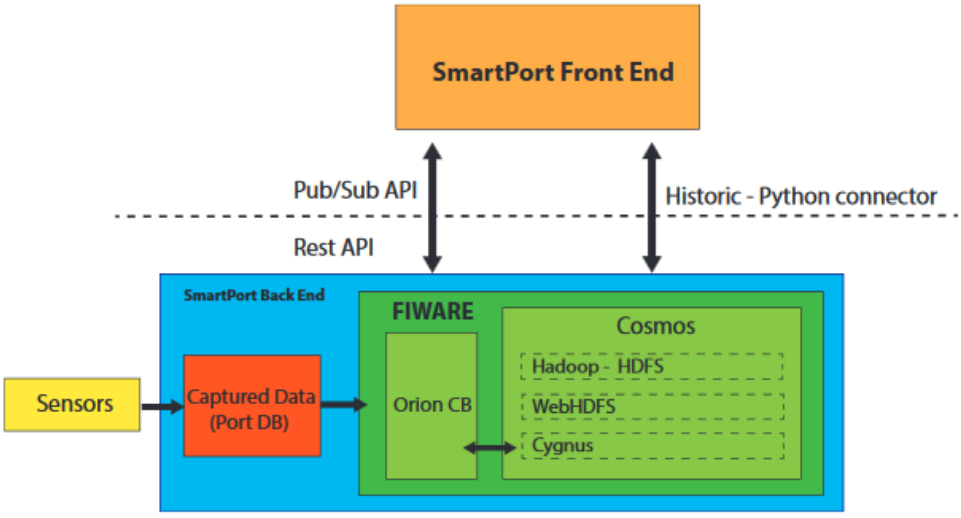

Figura 17. Arquitectura back-end de SmartPort [9].

2.3.2. Handling smart environments' devices, data and services at the semantic layer with FI-WARE

Este artículo trata sobre la solución de una serie de problemas acerca de la gestión de la información [10]:

- Integración de datos e interoperabilidad a nivel semántico: Cuando una aplicación y un servicio intercambian información sobre una habitación necesitan tener el mismo concepto de habitación.
- Agregaciones de información heterogénea: Tras obtener datos del tiempo, de la localización geográfica y de la lista de piscinas en el área se debe poder sacar una conclusión, por ejemplo, sugerir ir a una piscina cercana.

Estos problemas son resueltos mediante el desarrollo de una plataforma de gestión del sistema de iluminación de una ciudad, haciendo uso de una serie de sensores distribuidos a lo largo de ella, que miden parámetros de electricidad y medidas de presencia e iluminación de las farolas. En función de los datos obtenidos, el "Complex Event Processing" analiza la información y activa las alarmas. Mediante el módulo "Location Mgn" que se muestra en la Figura 18, se obtiene la posición de los móviles de los técnicos, así como la localización de las distintas farolas utilizando Google Maps.

La aplicación permite a los operadores ver las medidas de los distintos sensores, comprobar las alertas y su nivel de importancia, y asignar técnicos para que se dirijan a solucionar el problema. Además, cualquier técnico o ciudadano puede actualizar la información acerca de los problemas en las farolas adjuntando una imagen.

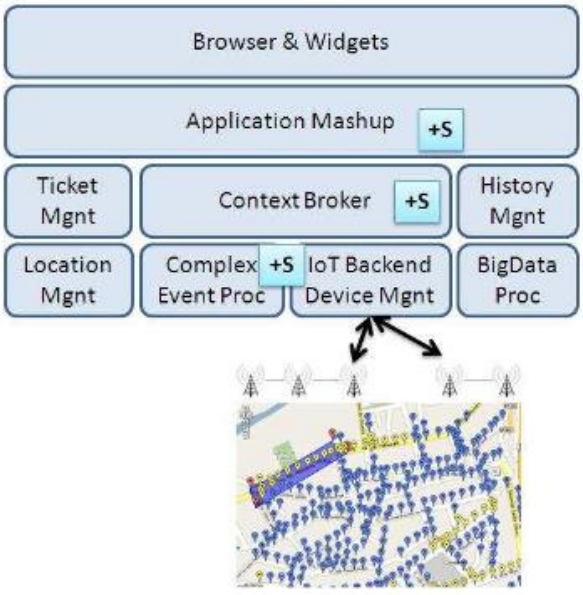

Figura 18. Arquitectura back-end de Street Lamps Management [10].

2.4. FIWARE frente al resto de desarrollos

FIWARE se presenta como una buena alternativa, pero existen otras opciones que en función de la situación del desarrollador pueden suponer una mejor elección. En la Figura 19 se muestra un listado de las ventajas y desventajas de cada uno de los tipos de desarrollo para ayudar a identificar con mayor facilidad las diferencias entre ellos.

Entre los distintos tipos de desarrollo existe una diferenciación, los gratuitos y los de pago. Los tipos de desarrollo gratuitos (FIWARE y desarrollo por herramientas) necesitan una mayor cualificación que los de pago, proporcionando a cambio una mayor libertad en el desarrollo. En cambio, los tipos de desarrollo de pago (por plataformas y por servicios) no necesitan de una alta cualificación, pero hay que pagar por sus servicios.

Características	FIWARE	Herramientas	Plataformas	Servicios
Documentación	✓	✓	✓	-
Ejemplos prácticos	✓	✓	✓	-
Precio	Gratis	Gratis	Pago	Pago
Código libre	✓	✓	✗	✗
Estándar	✓	✗	Pasos a seguir	✓
Apoyo e inversión	✓	-	✓	✓
Libertad en el desarrollo	✓	✓	Poca libertad	✗
Formación	Necesaria	Necesaria	No necesaria	Incluida
Cualificación necesaria	Alta	Alta	Media	Baja
Dificultad	Media	Alta	Baja	Muy Baja

Figura 19. Comparativa entre el desarrollo mediante FIWARE, herramientas, plataformas y servicios.

Si no se dispone de personal cualificado, los tipos de desarrollo que más encajan en este perfil son los de plataformas y los de servicios, siendo los de servicios una mejor opción en caso de disponer de un gran presupuesto y de conocer o tener buenas referencias de la empresa a la que se le va a contratar el servicio. En cualquier otro caso el servicio de plataformas supone una mejor opción.

El tipo de desarrollo por servicios no tiene marcadas las casillas de documentación y ejemplos prácticos debido a que dependen de la existencia de formación en el contrato establecido con la empresa que ofrece el servicio.

En el caso de que no incluyese formación, la cualificación necesaria y la dificultad serían nulas ya que la empresa contratada se encargaría del completo desarrollo y mantenimiento del producto desarrollado.

El hecho de que todo el desarrollo y mantenimiento pase a ser gestionado por la empresa externa no implica una ventaja frente al resto de tipos de desarrollo, debido a que en caso de desacuerdo entre las dos partes se pueden originar problemas graves en la empresa que contrató el servicio.

El tipo de desarrollo por herramientas puede no tener ninguna ventaja frente al resto de tipos, principalmente por la existencia de FIWARE, pero su capacidad de uso junto a otros tipos de desarrollo le otorga una gran ventaja ya que es capaz de adaptarse a cualquier situación proporcionando un valor adicional al resto de tipos de desarrollo.

En el caso de disponer de personal cualificado, FIWARE resulta la mejor opción, posee una gran cantidad de documentación y ejemplos prácticos, es de código libre, establece un estándar común para todos los desarrolladores, tiene apoyo e inversión constantes y proporciona completa libertad en el desarrollo.

3. Elementos básicos del desarrollo de soluciones IoT

En el desarrollo de soluciones IoT para ciudades inteligentes lo más destacado es el uso de módulos independientes o microservicios que, conectados entre sí, forman la arquitectura del sistema y permiten la explotación de la información de contexto con el objetivo de ofrecer la mejor solución según el contexto de la aplicación.

Estos dos elementos (los microservicios y la información de contexto) son el núcleo de toda arquitectura IoT en el desarrollo de sistemas para ciudades inteligentes. Los módulos o microservicios conforman el flujo de acciones del sistema con el objetivo de explotar de la manera más eficiente y útil la información obtenida de los sensores, que es conocida como información de contexto.

3.1. Microservicios

FIWARE está formado por una serie de módulos llamados Generic Enablers que, unidos entre sí, forman un ecosistema, existiendo módulos desde la parte de comunicación entre el bróker de contexto y los dispositivos, hasta módulos para guardar un histórico de datos. Esto es una práctica habitual dentro del desarrollo de soluciones IoT, los servicios por plataformas están formados por un conjunto de herramientas independientes que unidas crean un sistema completo.

Esta estructura es similar a la de los microservicios, un conjunto de pequeños servicios independientes que pueden conectarse entre sí para dar un determinado servicio o cumplir una funcionalidad [58].

La arquitectura por microservicios ha ido en aumento en los últimos años a causa de la simplificación del desarrollo y puesta en producción de aplicaciones que van escalando indefinidamente, reduciendo así la complejidad, aumentando la velocidad y eficiencia de los desarrollos, y facilitando el mantenimiento de los productos software producidos. Al estar dividido el software en una serie de microservicios las funcionalidades afectadas por la modificación de uno de ellos son mínimas, así como el hecho de sustituir partes de la aplicación o añadir nuevas [59]. También aseguran una mayor velocidad de procesamiento, al ser pequeños servicios enfocados en un único objetivo permiten una mejor depuración de su funcionamiento, ocasionando que actúen a mayor velocidad que las aplicaciones monolíticas.

Este tipo de desarrollo permite al desarrollador realizar de manera sencilla procesos como el escalado del sistema y la unión entre diferentes servicios (interoperabilidad), características necesarias en un desarrollo IoT por su gran flujo de datos y por la necesidad de utilizar diferentes servicios con el objetivo de proporcionar un mejor resultado.

Las ventajas que este tipo de desarrollo tiene frente a las aplicaciones monolíticas son las siguientes [60]:

- **Mantenimiento:** Resulta más sencillo puesto que al estar muy limitado el servicio las dependencias son menores.
- **Modularidad:** Al ser servicios separados pueden ser desarrollados por grupos diferentes, proporcionando una mayor diversidad de posibles soluciones (característica fundamental de FIWARE).
- **Rendimiento:** Pueden desplegarse en máquinas distintas, mejorando el rendimiento de la solución desarrollada (a cambio de un coste mayor), Figura 20.

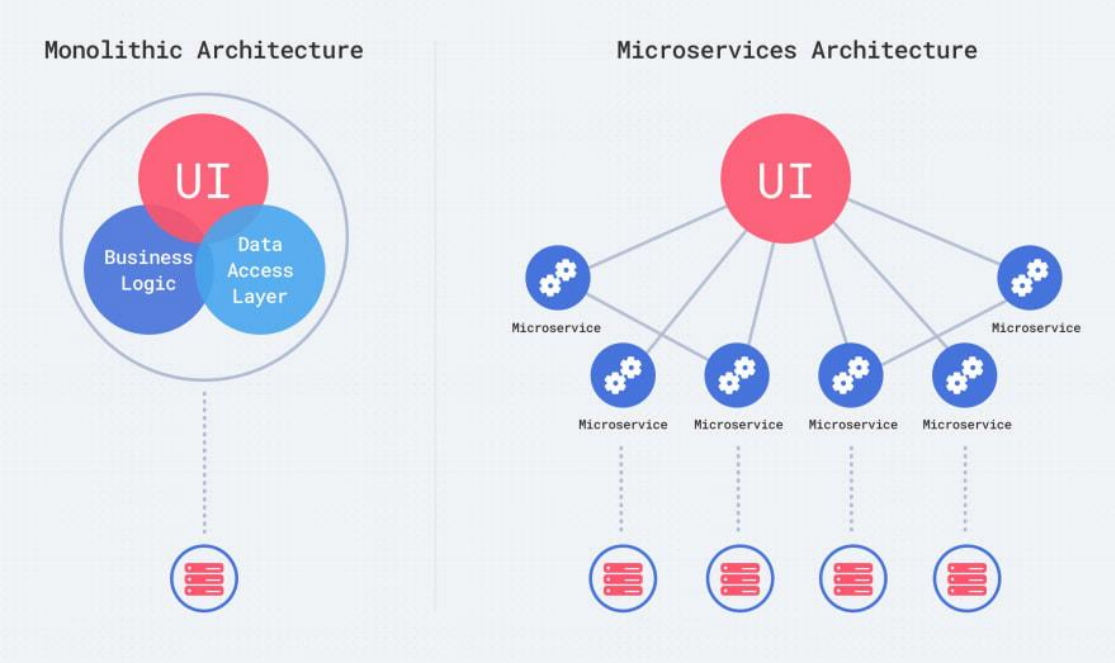

Figura 20. Arquitectura monolítica frente arquitectura mediante microservicios [61].

FIWARE presenta todas estas ventajas en su estructura, todos los servicios que lo conforman son servicios independientes, servicios que no tienen ningún tipo de relación con el resto de componentes, pero que a su vez permiten la conexión con el resto de los servicios para formar un único sistema.

Además, FIWARE permite a los desarrolladores crear y publicar sus propios servicios, así como utilizar servicios externos, por ejemplo, las herramientas de Eclipse, o servicios de plataformas como Azure IoT Suite.

3.2. Información de contexto

Los desarrollos IoT para ciudades inteligentes están basados en el tratamiento de la información de contexto, información que se intenta gestionar y procesar adecuadamente con el objetivo de adaptarla a las necesidades del usuario [9].

Muchas investigaciones tratan sobre la gestión de la información de contexto con el objetivo de darle el mejor uso posible, ya que los datos obtenidos son muy diferentes según el contexto en el que se encuentren, por ejemplo, una tarjeta en el contexto bancario se tratará de una tarjeta de crédito, pero en el contexto de la informática puede ser una tarjeta de video o una tarjeta de red. Esta diferencia supone un gran reto dentro del sector de IoT [62].

Pero, ¿qué significa realmente contexto?, ¿por qué es algo sobre lo que se habla tanto? Según la RAE contexto es: “Entorno lingüístico del cual depende el sentido y el valor de una palabra, frase o fragmento considerados” [63]. Según Anind K. Dey: “Contexto es cualquier de información que pueda ser usada para describir la situación de una entidad. Una entidad es una persona, lugar u objeto que se considera relevante para la interacción entre un usuario y una aplicación, incluyendo al usuario y a las propias aplicaciones” [64].

Es decir, contexto es conocer la situación en la que se encuentra una entidad para proporcionarle aquello que necesita o necesitará, de tal manera que el conocer el contexto permite proporcionar al usuario final una experiencia personalizada a sus necesidades, darle la información que quiere cuando la quiere, así como ofrecerle los servicios que necesita cuando los necesita y como los necesita [65].

Esto ocasiona que se desarrollen propuestas o soluciones para el tratamiento de la información de contexto y para la adaptación del servicio a las necesidades del usuario final, por ejemplo, con el desarrollo de GeoNat [66] y el de LISA [67].

La solución a este problema supondría un gran paso en el desarrollo de soluciones IoT, permitiendo la completa explotación de cualquier tipo de información captada por los dispositivos y la comunicación de sistemas que capten distintos tipos de información con el objetivo de obtener una serie de conclusiones, por ejemplo, a partir de distintos dispositivos ubicados por la ciudad: un tipo de dispositivo que obtenga los datos meteorológicos, un tipo de dispositivo ubicado en cada una de las piscinas públicas que recopile información del agua y los móviles de los ciudadanos, ser capaz de sugerir a un ciudadano una piscina cercana para bañarse.

Todas estas soluciones son pensando en el usuario final, pero ¿y el desarrollador?

Para ayudar al desarrollador se propone la creación de un conjunto de microservicios, MI-FIWARE, que libere al desarrollador de la carga de trabajo del back-end y del tratamiento de la información de contexto, permitiéndole centrarse en el desarrollo front-end.

4. MI-FIWARE

Supongamos que somos dueños de una empresa y, por no disponer de las condiciones adecuadas en el edificio y el cambio de temperatura entre estaciones, nuestros empleados están enfermando. Algunos empleados ponen el aire a muy baja temperatura y otros a muy alta, por lo que se va a automatizar la temperatura y humedad de cada pasillo y habitación del edificio.

Se instalará una serie de sensores en habitaciones y pasillos que medirán temperatura y humedad. Esta información debe obtenerse, procesarse y, a partir de ella, permitir establecer la temperatura y humedad a unas condiciones ideales, así como poder monitorizarlas y cambiarlas manualmente.

Para llevar a cabo este desarrollo se ha decidido utilizar FIWARE, desplegándose como se muestra en la Figura 21 la estructura necesaria entorno a su bróker de contexto. Para el desarrollo de la herramienta de monitorización y gestión se ha propuesto utilizar para el front-end Angular y para el back-end, Node.js y Express.js, donde se desarrollará una API que se alimente del bróker de contexto.

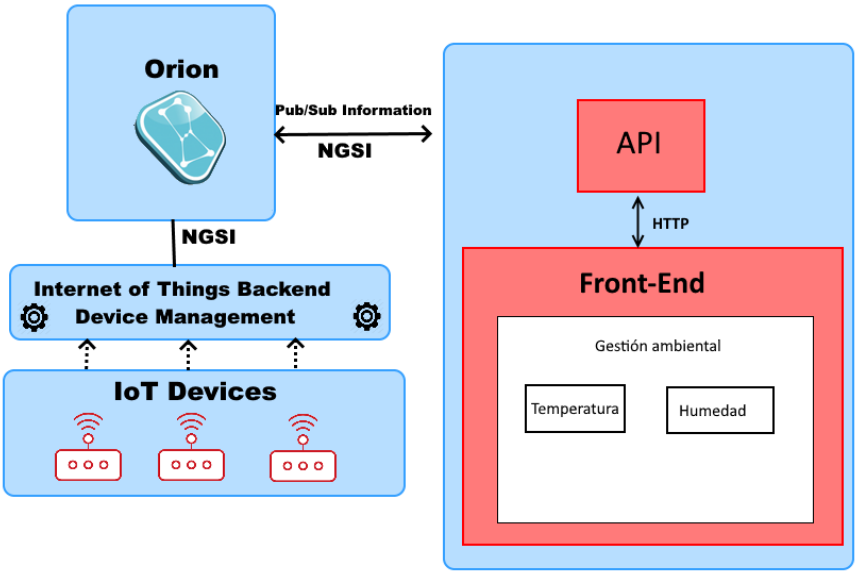

Figura 21. Arquitectura del ejemplo propuesto.

Pero se presenta un problema, no disponemos de personal cualificado para el desarrollo del back-end y la aplicación debe estar terminada lo más pronto posible con el objetivo de evitar más bajas en el personal.

Utilizando FIWARE se podría optar por utilizar módulos como Wirecloud, pero a cambio no permitirían la completa personalización de la aplicación a las necesidades de la empresa.

Por lo que para ayudar a los desarrolladores de front-end y que no tengan que hacer el desarrollo del back-end para procesar la información del bróker de contexto se propone el desarrollo de un conjunto de microservicios, con el cual el desarrollador pueda adquirir información de contexto solo preocupándose de la dirección del microservicio y de la información que se quiere adquirir. De esta manera el desarrollador no deberá de preocuparse del back-end, solo del front-end, facilitando de esta manera el desarrollo de la solución.

Este microservicio, a partir de una cadena de texto que proporcionará el desarrollador de front-end en su componente web, le transmitirá los datos de contexto que se ajusten a ese texto introducido, permitiéndole suscribirse a ellos. De esta manera el desarrollador no tendrá que realizar un desarrollo de back-end, pudiendo centrarse en el desarrollo de front-end.

Para que esto sea posible, el desarrollador de front-end deberá seguir una estructura específica a la hora de desarrollar su componente web con el objetivo de poder gestionar esa información de contexto dinámicamente.

Además, se abre la posibilidad de integrar un microservicio de autenticación y autorización que según la configuración que se le aplique obligue al usuario a proporcionar unas credenciales para poder acceder a cierta información de contexto [68], sirviendo de apoyo al microservicio propuesto.

Las llamadas a datos de contexto se harán directamente sobre el bróker de contexto, pudiendo integrarse un microservicio que permita elegir si obtener los datos del bróker de contexto o de la base de datos de históricos, minimizando la complejidad y carga de trabajo [69].

Un ejemplo sería el mostrado en la Figura 22, Orion obtiene datos de una serie de dispositivos IoT, estos datos se van almacenando en una base de datos utilizando Cygnus para la gestión del histórico, y estos datos se le proporcionan a la aplicación a través de un middleware.

El middleware tiene un servicio fundamental, Microservice Management, que se encargará de proporcionar la información necesaria a la aplicación ya procesada y lista para mostrar al usuario final. Adicionalmente, se puede hacer uso de dos microservicios, uno para el control de acceso, de tal manera que se restrinja la obtención de ciertos datos; y uno para la gestión de la fuente de datos, permitiendo llamar adicionalmente a datos del histórico.

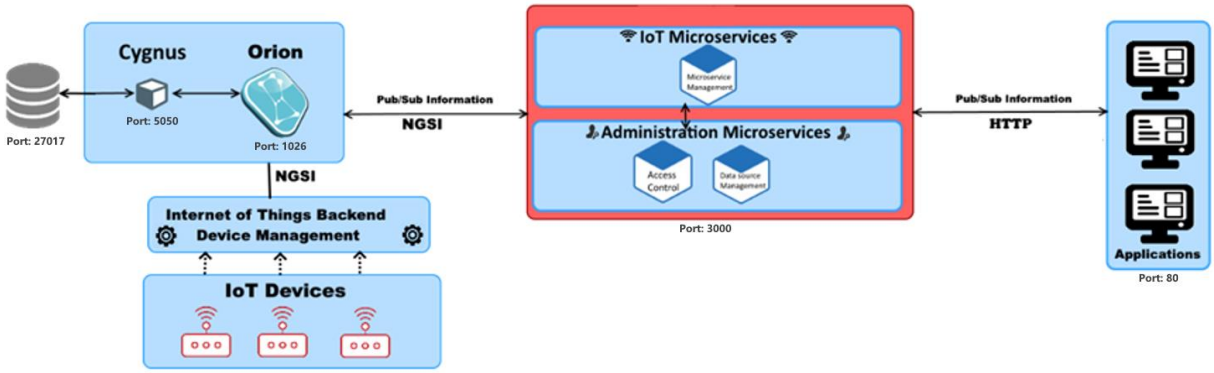

Figura 22. Ejemplo de arquitectura del sistema propuesto.

Con respecto al ejemplo anterior, el desarrollador solamente deberá indicar al microservicio la palabra clave “habitación” en el caso de querer obtener la información de cada una de las habitaciones del edificio. Si quisiera la información de los pasillos debería indicar “pasillo”.

Siempre se le proporciona la información desde la más genérica a la más específica, dando “habitación” toda la información de las habitaciones y “salón” la información de la habitación identificada como salón.

Una vez el microservicio sabe qué información quiere el desarrollador, le proporciona toda la información disponible en el bróker de contexto que el desarrollador le ha solicitado.

En la Figura 23 se muestra un ejemplo de tratamiento y comunicación de datos entre el bróker, el microservicio y la aplicación. El cliente pregunta por el dato “Room”, al ser un dato genérico el microservicio pregunta por todos los datos “Room” existentes en el bróker. Una vez obtenidos los datos procesa la información y se la devuelve al cliente siguiendo la estructura de datos definida entre las dos partes.

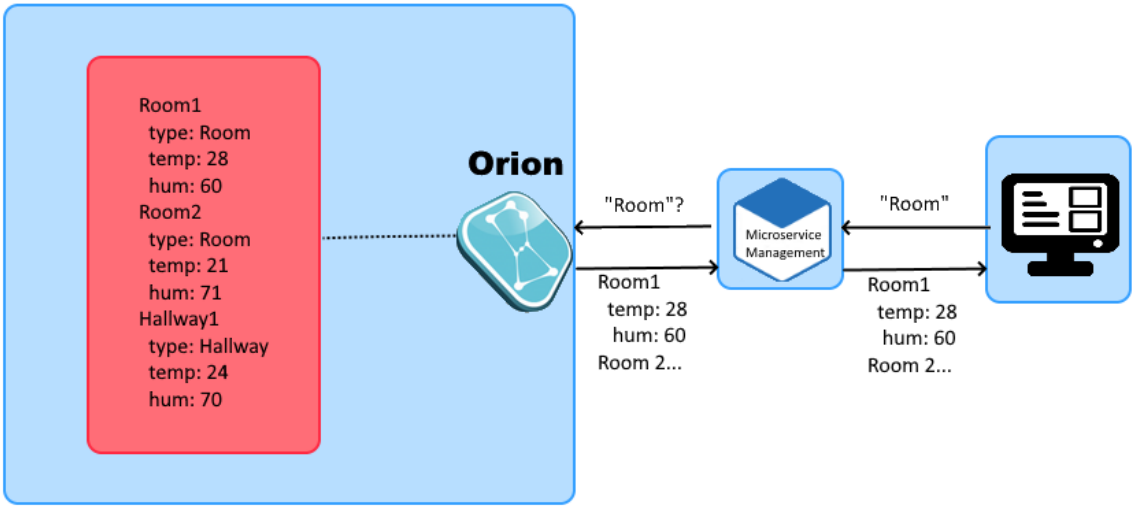

Figura 23. Ejemplo de tratamiento y comunicación de datos del microservicio.

En este trabajo se realiza el desarrollo del Microservice Management[1], microservicio que junto a la estructura de componentes propuesta[2] libera al desarrollador de la carga del back-end. A esta propuesta de microservicios entre el bróker de FIWARE y la aplicación se le dado el nombre de MI-FIWARE [70] [71].

4.1. Microservicio

El microservicio ha sido desarrollado haciendo uso de las tecnologías Node.js, Express.js y SocketIO. Su función será la de liberar al desarrollador de la carga de trabajo del back-end.

El microservicio es quien obtiene y procesa los datos del bróker de contexto, y quien se ocupa de gestionar las suscripciones con el bróker y emitir la información que recibe. La información que emite es publicada en un canal de comunicación entre el microservicio y los componentes, pasando a estar disponible entre todos los componentes que quieran hacer uso de ella.

Dentro del desarrollo del microservicio destacan tres tecnologías: Node.js [72] para la gestión del servidor, Express.js [73] para la gestión de las peticiones y SocketIO [74] para la gestión de la comunicación con los componentes.

[1] https://github.com/Jalbladewing/microservicioFiware
[2] https://github.com/Jalbladewing/stenciljsFiware

Node.js es un entorno de ejecución de JavaScript que actúa desde la capa del servidor y que posee una peculiaridad: al contrario que los entornos tradicionales que trabajan con múltiples hilos, Node.js trabaja con un solo hilo, no bloqueando de esta manera la E/S de las conexiones entrantes, lo que le permite dar un mayor número de respuestas frente a los entornos tradicionales.

Por ejemplo, Nginx funciona de la misma forma que Node.js, posee un solo hilo para dar respuesta a las peticiones de los clientes (véase Figura 24) [75].

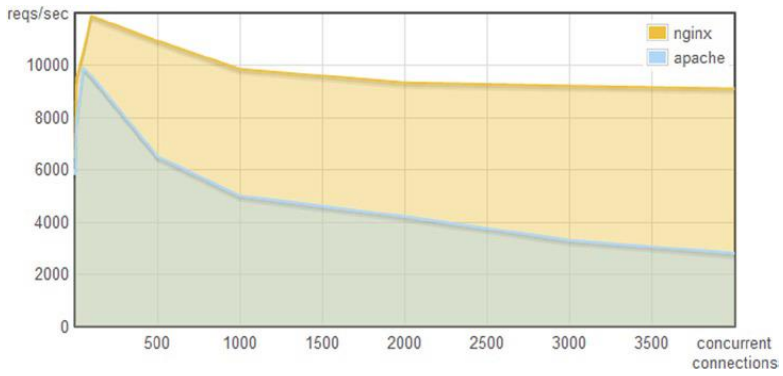

Figura 24. Gráfica comparativa de conexiones concurrentes entre Nginx y Apache [75].

A cambio de ser capaz de dar más respuesta a los clientes es muy sensible a las operaciones que tengan un alto consumo de CPU, ya que bloquearían el único hilo que utiliza. Node.js resulta ideal para aquellos servidores con un alto número de peticiones pero con un bajo uso de la CPU, como es el caso de este microservicio [76]. Por esta razón se ha elegido Node.js frente a sus competidores, nuestro microservicio se va a encargar de responder a una gran cantidad de componentes, pero no va a hacer uso de CPU, obtendrá la información, la procesará tratándola como una cadena de texto y se la enviará al componente. En el caso de que ese procesamiento de texto llegará a consumir mucha CPU se plantearía el utilizar otro tipo de servidor que haga uso de múltiples hilos. Una vez elegido el entorno de ejecución donde va a funcionar el microservicio faltan por definir la manera y herramientas necesarias para comunicarse con los componentes y el bróker de contexto. Node.js por defecto permite realizar esta funcionalidad, pero existe un framework llamado Express.js que facilita el desarrollo de los puntos de acceso [75]. Por ejemplo, el utilizar Express.js permite utilizar un middleware para la gestión de estructuras JSON llamada body-parser, algo necesario para el microservicio [77]. Teniendo preparado el entorno donde se va a levantar el microservicio y una manera de gestionar las peticiones, es necesario una forma de poder establecer una comunicación con los componentes, ya que no deben de estar siempre abiertos a comunicaciones, sino que cuando haya un cambio deben de detectar y obtener los datos asociados al cambio realizado. Esta necesidad ya está resuelta por una tecnología existente y utilizada en los servicios de mensajería, los WebSockets [78]. La tecnología de WebSockets establece una línea de comunicación bidireccional entre el cliente y el servidor, provocando que al haber un cambio en los datos el servidor pueda enviarle la información directamente al cliente [79].

Esta tecnología se ha implementado utilizando la librería SocketIO, ya que facilita el uso de WebSockets en Express.js, permitiendo crear eventos dentro de la comunicación que se establece entre el cliente y el servidor. Esta característica permite filtrar la información transmitida a los componentes [80].

Con respecto a los datos transmitidos, se ha especificado una forma de estructurar estos datos con el objetivo de que todos los componentes sean capaces de entender la información que el microservicio les proporcione, así como establecer qué comunicaciones se van a realizar y qué información se va a tratar en dichas comunicaciones.

La estructura, como se muestra en la Figura 25 consiste en un vector de entidades donde cada entidad está formada por un nombre y un vector de valores. Esta estructura es del tipo JSON para facilitar la comunicación y explotación de los datos por parte de los componentes.

```
{
    "entities":
    [{
        "name":
        "values":
    [
            {"name":"variable1","value":"241"},
            {"name":"variable2","value":"Juan"}
        ]
    },
    {
    "name":
        "values":
    [
            {"name":"variable1","value":"300"},
            {"name":"variable2","value":"Pedro"}
        ]
    }]
}
```

Figura 25. Estructura de datos propuesta.

Se ha elegido esta estructura de datos porque permite la manipulación de cualquier tipo de información que el bróker de contexto pueda contener, generalizando todos los datos como entidades que contienen valores. Bajo esta estructura no se ha tenido en cuenta la información de metadatos adicional que tiene cada atributo o valor del bróker de contexto para reducir la complejidad del problema.

La comunicación se divide en dos tipos de comunicación, las de inicialización y las de suscripción. La comunicación de inicialización (véase Figura 26) ocurre cuando los componentes se inician por primera vez. En el momento en el que los componentes se inicializan realizan una llamada GET al microservicio para obtener los datos iniciales, en esta llamada indican el tipo de entidad que desean obtener, por ejemplo, Room, y si desean alguna entidad en específico, por ejemplo, Room1.

El microservicio una vez recibe esta llamada realiza una llamada al bróker de contexto para obtener los datos del tipo y entidad (si se ha especificado) indicados. Una vez recibidos los datos, los procesa para devolvérselos al componente siguiendo la estructura explicada anteriormente.

El componente, tras recibir los datos iniciales, establece una suscripción a esos datos con el microservicio y muestra al usuario la interfaz desarrollada.

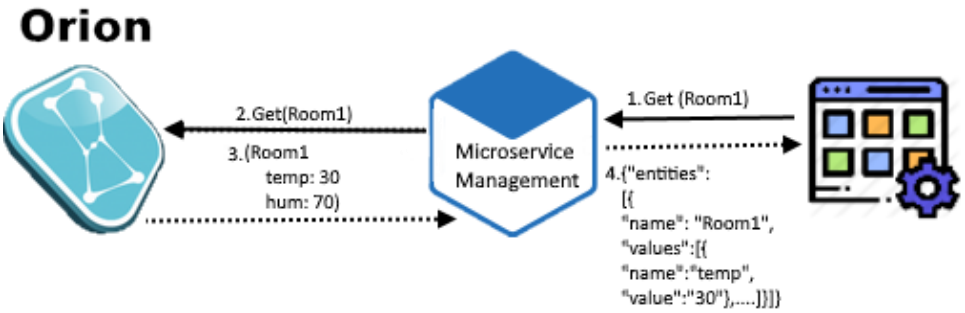

Figura 26. Comunicación de datos entre Orion, el microservicio y un componente.

La comunicación de suscripción (Figura 27) se realiza en el momento en el que los dispositivos actualizan su información, por ejemplo, por un cambio de temperatura.

Cuando ocurre este tipo de comunicación el bróker de contexto avisa al microservicio, y éste realiza una llamada al bróker de contexto para obtener todos los datos de la información que el bróker le proporcionó, ya que el bróker solo proporciona el dato modificado. Por ejemplo, en el caso de la temperatura de una habitación solo devolverá la habitación afectada y la temperatura actual.

Tras obtener todos los datos el microservicio procesa los datos y los publica bajo un identificador para que todos los componentes interesados puedan adquirir esa información y actualizar sus datos.

Este identificador es un parámetro del POST al microservicio y se le indicará al bróker de contexto en el momento de crear la suscripción, por lo que en el momento en el que el bróker avise al microservicio de que se han realizado cambios en los datos el microservicio ya conocerá el identificador.

El identificador sirve para que los componentes solo se alimenten de los datos que les interesen, por ejemplo, a un componente de datos de una habitación no le interesan los datos de las farolas de la calle, ni los datos del resto de habitaciones.

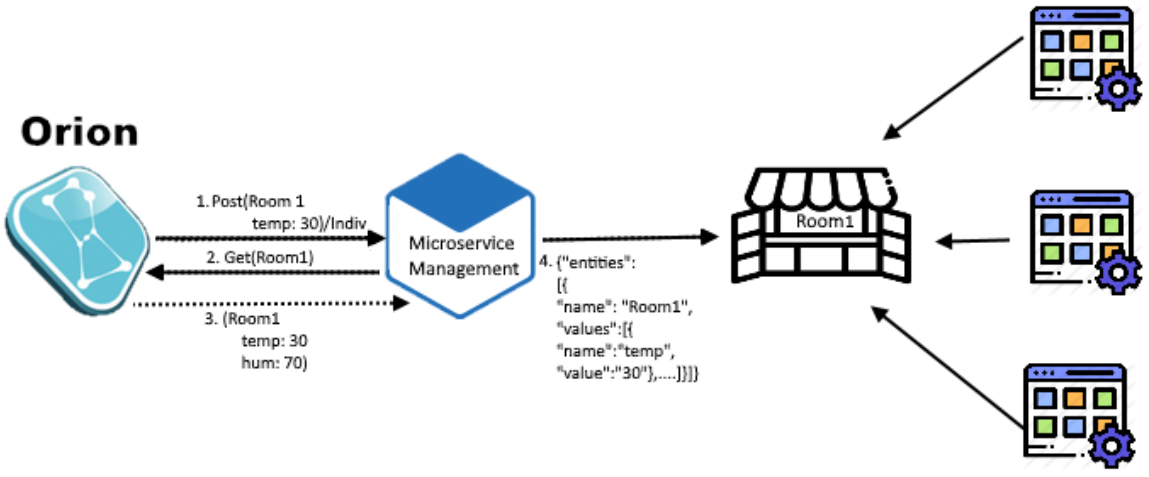

Figura 27. Flujo de datos de suscripciones entre Orion, el microservicio y componentes.

El microservicio está formado por tres partes:

- **Dominio:** Define las funciones que ayudarán a formar los objetos existentes (entity y value).
- **Controlador:** Contiene las acciones o funciones que se ejecutan (deserialización del JSON) y se encarga del envío mediante SocketIO.
- **Rutas:** Encargadas de gestionar los GET y POST recibidos.

El desarrollo de la estructura se ha realizado en dos versiones.

La primera versión ha sido enfocada en tener un microservicio funcional con unas características básicas que lo hagan útil a la hora de facilitar el desarrollo de componentes.

La segunda versión se ha enfocado en permitir que la gestión de los componentes sea accesible desde la interfaz, permitiendo a los usuarios parametrizarlos sin tener que acceder al código de la aplicación. También, se ha añadido una funcionalidad extra de filtrado y suscripción por filtro para poder ver en tiempo real los datos de los dispositivos que más interesen. Adicionalmente, se ha creado una función para obtener todos los atributos de un tipo de entidad y los de toda la aplicación. Por último, se han modificado las funciones para permitir la suscripción a todas las entidades existentes en el bróker de contexto.

4.1.1. Dominio

El microservicio está formado por dos funciones (véase Figuras 28 y 29) que ayudarán a definir la estructura explicada anteriormente, donde una entidad o entity está formada por una serie de valores o values, similar al modelo de datos definido por NGSI (véase Figura 30).

```
domain > JS Entity.js > ...
1  function Entity(name, values) {
2    // custom type checking here...
3    this.name = name;
4    this.values =  values;
5  }
6
7  module.exports = Entity;
```

Figura 28. Función Entity del dominio.

```
domain > JS Value.js > Value
1  function Value(name, value)
2  {
3    this.name = name;
4    this.value =  value;
5  }
6
7  module.exports = Value;
```

Figura 29. Función Value del dominio.

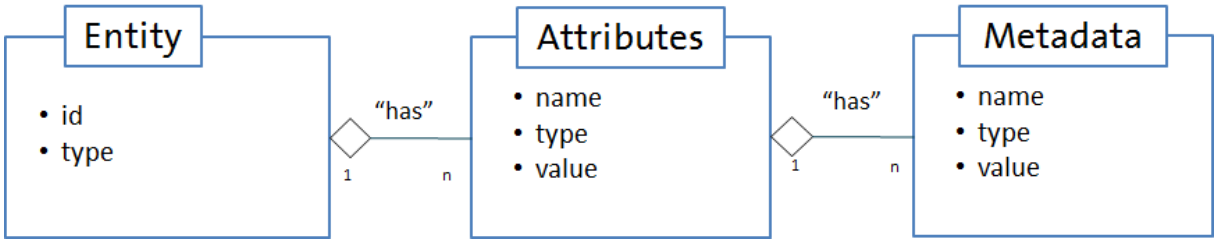

Figura 30. Modelo de datos de NGSI [81].

4.1.2. Controlador

El controlador se encarga de ejecutar las funciones necesarias para el procesamiento de los datos y su transmisión a los distintos componentes mediante SocketIO.

La función monitor (véase Figura 31) es la encargada de llamar a la función de deserialización y de transmitir los datos publicándolos bajo un identificador.

En una segunda versión el controlador se ha modificado para que añada a la identificación del canal el filtro utilizado (véase Figura 32), así, como se muestra en la Figura 33, ese canal solo será escuchado por aquellos componentes interesados en el filtro que se utilizó para una entidad de datos específica y un tipo de dato específico.

```
7   function monitor(type, id, payload) {
8     if (payload && Object.keys(payload).length !== 0) {
9       var dataPreProcessed = payload.split(",")
10      entities = deserializeJson(dataPreProcessed);
11
12      var entitiesJson = "{" + '"entities": ' + JSON.stringify(entities) + "}";
13      SOCKET_IO.emit(type + "-" + id, JSON.parse(entitiesJson));
14    }
15  }
```

Figura 31. Primera versión de la función monitor del controlador.

```
7   function monitor(type, id, filter, payload) {
8     if (payload && Object.keys(payload).length !== 0) {
9       var dataPreProcessed = payload.split(",")
10      entities = deserializeJson(dataPreProcessed);
11      var entitiesJson = "{" + '"entities": ' + JSON.stringify(entities) + "}";
12      SOCKET_IO.emit(type + "-" + id + "-" + filter, JSON.parse(entitiesJson));
13    }
14  }
```

Figura 32. Cambios en la función monitor de la segunda versión.

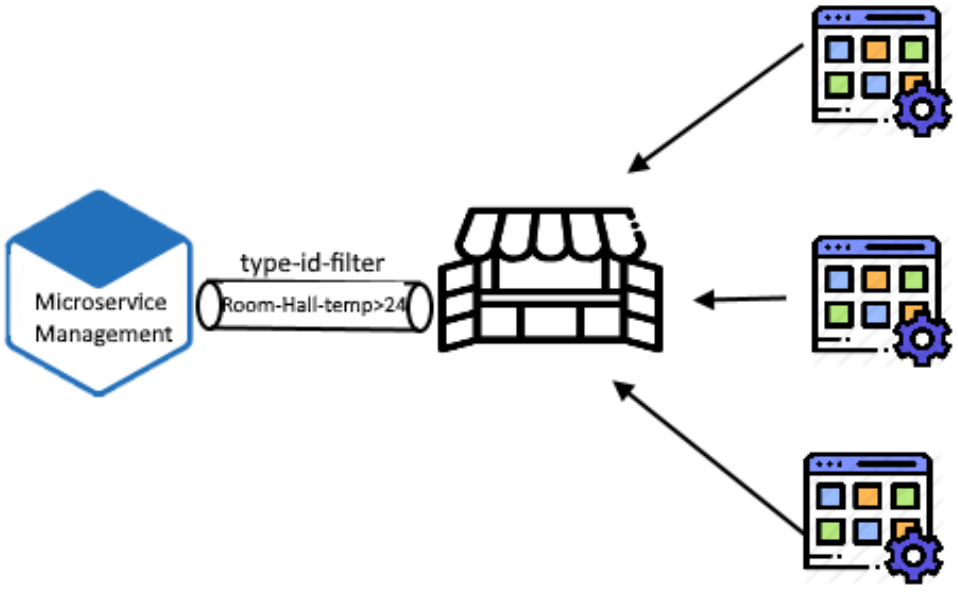

Figura 33. Nuevo flujo de datos entre el microservicio y los componentes utilizando filtros.

La deserialización (véase Figura 34) se hará mediante la gestión del JSON que proporciona el bróker de contexto, tratándolo como un string, ya que por la estructura del JSON no es posible deserializarlo directamente a un objeto, sino que se debe realizar un tratamiento de cadenas de texto.

```
56  function deserializeJson(reqBody)
57  {
58    var entities = [];
59    var values = [];
60    var entityName = "";
61
62    for(var i = 0; i < reqBody.length; i++)
63    {
64      if(reqBody[i].includes('"id":'))
65      {
66        if(i!= 0)
67        {
68          entities.push(new Entity(entityName, values));
69          values = [];
70        }
71        entityName = reqBody[i].split(":")[1].replace(new RegExp('"', 'g'), "");
72
73      }else if(!reqBody[i].includes('"type":'))
74      {
75        var processedData = reqBody[i].replace("}]", "").replace("}", "").split(":");
76        var value = new Value(processedData[0].replace(new RegExp('"', 'g'), ""), processedData[1]);
77        values.push(value);
78      }
79    }
80    entities.push(new Entity(entityName, values));
81    return entities;
82  }
83
```

Figura 34. Función de deserialización (deserializeJson) del controlador.

4.1.3. Rutas

En las rutas se gestionan las llamadas explicadas en la sección anterior, usando las cláusulas GET y POST.

El GET a partir de los datos proporcionados por el componente obtiene la información requerida del bróker de contexto, deserializa la información obtenida, se suscribe al bróker por parte del componente y envía los datos que solicitó el componente (véase Figura 35).

```
91  router.get('/subscription', (req, res) => {
92    var paramName = req.query.entity
93    var paramId = req.query.id
94    var entities = [];
95    var extraUrl = "";
96
97    if(paramId != "") extraUrl = "&id=" + paramId;
98
99    request(orionUrl + ":1026/v2/entities?type=" + paramName + extraUrl +"&options=keyValues", function (error, response, body) {
100     if (!error && response.statusCode == 200)
101     {
102       var reqBody = body.split(",")
103       entities = monitor.deserializeJson(reqBody);
104
105       var entitiesJson = "{" + '"entities": ' + JSON.stringify(entities) + "}";
106
107       createSubscription(paramName, paramId);
108       res.json(JSON.parse(entitiesJson));
109     } else {
110       console.log("There was an error: ") + response.statusCode;
111       console.log(body);
112       res.status(response.statusCode).send();
113     }
114   });
115   });
```

Figura 35. Primera versión del punto de acceso GET del microservicio.

En la segunda versión, el GET, en caso de que la petición no sea para una entidad en específico, comprueba si se le ha introducido un filtro. En el caso de haberse introducido un filtro le proporciona al componente las entidades que cumplen ese filtro y comunica al creador de suscripciones el filtro que se le ha introducido para que cree una suscripción acorde a esos datos. (véase Figura 36).

```
117  router.get('/subscription', (req, res) => {
118    var paramName = req.query.entity;
119    var paramId = req.query.id;
120    var paramQuery = '';
121    var entities = [];
122    var extraUrl = "";
123    var extraUrlQuery = '';
124
125    if(paramId != "")
126    {
127      extraUrl = "&id=" + paramId;
128    }else
129    {
130      if(req.query.queryFilter != 'undefined' && req.query.queryFilter != "" )
131      {
132        paramQuery = decodeURIComponent(req.query.queryFilter);
133        extraUrlQuery = '&q=' + decodeURIComponent(req.query.queryFilter);
134      }
135    }
136
137    request(orionUrl + ":1026/v2/entities?type=" + paramName + extraUrl + extraUrlQuery +"&options=keyValues", function (error, response, body) {
138      if (!error && response.statusCode == 200)
139      {
140          var reqBody = body.split(",")
141          entities = monitor.deserializeJson(reqBody);
142
143          var entitiesJson = "{" + '"entities": ' + JSON.stringify(entities) + "}";
144
145          createSubscription(paramName, paramId, paramQuery);
146          res.json(JSON.parse(entitiesJson));
```

Figura 36. Cambios en el punto de acceso GET del microservicio.

Para la suscripción existe un problema, se pueden crear tantas suscripciones como se quiera, pero aun existiendo permite crear otras suscripciones iguales, lo que puede llevar a un flujo infinito de datos mediante suscripciones.

Para solucionar este problema, antes de crear una suscripción se comprueba si existe otra suscripción igual, y en el caso de que no exista la suscripción, se crea una nueva (véase Figura 37 y Figura 38).

```
24  function createSubscription(paramName, paramId)
25  {
26    request(orionUrl + ":1026/v2/subscriptions", function (error, response, body) {
27      if (!error && response.statusCode == 200)
28      {
29        var reqData = JSON.parse(body);
30        var exists = false;
31        var name = paramName;
32        var queryId = '"idPattern": ".*", ';
33
34        if(paramId != "")
35        {
36          name = paramId;
37          queryId = '"id": "' + paramId + '", ';
38        }
39
40        for(var i = 0; i < reqData.length; i++)
41        {
42          if(reqData[i].description == 'Notify me of all ' + name + ' changes') exists = true;
43        }
44        if(!exists)
45        {
46          request({
47            headers: {
48              'Content-Type': 'application/json'
```

Figura 37. Primera versión de la inicialización de datos de la función createSubscription de la ruta.

```
        },
        uri: orionUrl + ':1026/v2/subscriptions?options=skipInitialNotification',
        body: '{ "description": "Notify me of all ' + name + ' changes", "subject": { "entities": [{' +
        queryId + '"type": "' + paramName + '"}],'+
        +'"condition": { "attrs": [ ] } },  "notification": {"http": { "url": "http://192.168.2.114:3000/subscription/'+ paramName +
        '&' + paramId + '" } } }',
        method: 'POST'
      }, function (err, res, body) {
        //it works!
      });
    }

  } else {
      console.log("There was an error: ") + response.statusCode;
      console.log(body);
      res.status(response.statusCode).send();
  }
 });
}
```

Figura 38. Primera versión de la petición de suscripción de la función createSubscription de la ruta.

En la segunda versión de la función de crear suscripciones se ha incluido un filtro en el enlace al que llama el bróker de contexto cuando se produce un cambio en el dato suscrito (véase Figura 39).

```
request({
  headers: {
    'Content-Type': 'application/json'
  },
  uri: orionUrl + ':1026/v2/subscriptions?options=skipInitialNotification',
  body: '{ "description": "Notify me of all ' + name + ' changes", "subject": '
  +'{ "entities": [{' + queryId + '"type": "' + paramName + '"}],  "condition": '
  +'{ "attrs": [ ]' + '} },  '
  +'"notification": {"http": { "url": "' + microserviceUrl + ':3000/subscription/'+ paramName + '&' + paramId + '&' + notifyFilter + '" } } }',
  method: 'POST'
```

Figura 39. Cambios en la función createSubscription al crear la suscripción.

También, se han realizado cambios para permitir hacer suscripciones condicionales, es decir, que el bróker de contexto avise al microservicio cuando ocurra algo, de manera similar a una alarma, por ejemplo, cuando la temperatura suba de 4º en una cámara frigorífica (véase Figura 40).

```
34        var queryFilter = '';
35        var notifyFilter = '';
36
37        if(paramId != "")
38        {
39          name = paramId;
40          queryId = '"id": "' + paramId + '", ';
41        }else
42        {
43          if(paramQuery!='undefined' && paramQuery != "")
44          {
45            notifyFilter = encodeURIComponent(paramQuery);
46            name = paramName + " with conditions " + encodeURIComponent(paramQuery);
47            queryFilter = ',"expression": {"q": "'+ paramQuery +'"}'; //Notify when on change this condition is true (not used yet)
48          }
49        }
```

Figura 40. Cambios de filtros en la función createSubscription.

El POST, como se ha explicado en la sección anterior, obtendrá los datos de suscripción del bróker de contexto (véase Figura 41).

Una vez obtenidos los datos que proporciona el bróker, el microservicio llamará de vuelta al bróker para obtener todos los datos, ya que éste solo envía los datos solicitados en lugar de toda la información.

Después de obtener todos los datos del bróker, los deserializará y se los transmitirá a los componentes para que puedan utilizarlos.

```
68  // Whenever a subscription is received, display it on the monitor
69  // and notify any interested parties using Socket.io
70  router.post('/subscription/:type', (req, res) => {
71    var params = req.params.type.split("&");
72    var extraUrl = "";
73    if(params[1] != "") extraUrl = "&id=" + params[1];
74
75    request(orionUrl + ":1026/v2/entities?type=" + params[0] + extraUrl +"&options=keyValues", function (error, response, body) {
76      if (!error && response.statusCode == 200)
77      {
78          monitor.monitor(params[0], params[1], body);
79
80      } else {
81          console.log("There was an error: ") + response.statusCode;
82          console.log(body);
83          res.status(response.statusCode).send();
84      }
85  });
86
87
88    res.status(204).send();
89  });
```

Figura 41. Primera versión del punto de acceso POST del microservicio.

En la segunda versión del POST se añadió la gestión del filtro para que en el caso de que exista un filtro se le comunique al controlador, y de esta manera poder avisar a los componentes interesados en esa información sin afectar al resto de componentes (véase Figura 42).

Debido a que las condiciones poseen caracteres especiales como <, > y =; se ha codificado la condición al enviársela al bróker de contexto y se ha decodificado al recibirla.

```
84  router.post('/subscription/:type', (req, res) => {
85    var params = req.params.type.split("&");
86    var extraUrl = "";
87    var extraUrlQuery = '';
88    var filterData = "";
89    if(params[1] != "")
90    {
91      extraUrl = "&id=" + params[1];
92    }else
93    {
94      if(params[2] != "" && params[2] != 'undefined')
95      {
96        filterData = params[2];
97        extraUrlQuery = '&q=' + decodeURIComponent(params[2]);
98      }
99    }
100
101   request(orionUrl + ":1026/v2/entities?type=" + params[0] + extraUrl + extraUrlQuery +"&options=keyValues"
102     if (!error && response.statusCode == 200)
103     {
104       monitor.monitor(params[0], params[1], filterData, body);
105
```

Figura 42. Cambios en el POST del microservicio.

También, se ha habilitado un punto de acceso para obtener un listado de todos los tipos de dispositivos disponibles (véase Figura 43), permitiendo así la creación de componentes de listado de tipos de dispositivos para poder filtrar entre ellos.

```
155   router.get('/typeList', (req, res) => {
156     request(orionUrl + ":1026/v2/types?options=values", function (error, response, body) {
157       if (!error && response.statusCode == 200)
158       {
159           res.json(JSON.parse(body));
160       } else {
161           console.log("There was an error: ") + response.statusCode;
162           console.log(body);
163           res.status(response.statusCode).send();
164       }
165   });
166   });
```

Figura 43. Punto de acceso GET de listado de tipos de entidades.

Por último, se ha habilitado un punto de acceso para obtener la lista de todos los atributos de un tipo de entidad (véase Figura 44 y Figura 45), en el caso de que no se le pase ningún dato devuelve todos los atributos que existen en el bróker de contexto.

```
200   router.get('/attributeList', (req, res) => {
201     var paramName = req.query.type;
202     var attributeList = [];
203
204     if(paramName != "" && paramName != "undefined")
205     {
206       request(orionUrl + ":1026/v2/entities?type="+ paramName +"&options=keyValues", function (error, response, body) {
207         if (!error && response.statusCode == 200)
208         {
209             try
210             {
211               request(orionUrl + ":1026/v2/entities/" + JSON.parse(body)[0].id + "/attrs?options=keyValues", function (error, response, body) {
212                 if (!error && response.statusCode == 200)
213                 {
214                   for(var i = 0; i < Object.keys(JSON.parse(body)).length; i++)
215                   {
216                     attributeList.push(Object.keys(JSON.parse(body))[i])
217                   }
218                   res.json(attributeList);
219                 } else {
220                   console.log("There was an error: ") + response.statusCode;
221                   console.log(body);
222                   res.status(response.statusCode).send();
223                 }
224               });
225               }catch(error)
226               {
227                 console.log("There was an error: ") + response.statusCode;
228                 console.log(body);
229                 res.status(204).send();
230               }
```

Figura 44. Punto de acceso GET de listado de atributos de un tipo de entidad.

```
238     }else
239     {
240       request(orionUrl + ":1026/v2/types?options=noAttrDetail", function (error, response, body) {
241         if (!error && response.statusCode == 200)
242         {
243           for(var i = 0; i < JSON.parse(body).length; i++)
244           {
245             for(var j = 0; j < Object.keys(JSON.parse(body)[i].attrs).length; j++)
246             {
247               if(!attributeList.includes(Object.keys(JSON.parse(body)[i].attrs)[j])) attributeList.push(Object.keys(JSON.parse(body)[i].attrs)[j]);
248             }
249           }
250           res.json(attributeList);
251         } else {
252           console.log("There was an error: ") + response.statusCode;
253           console.log(body);
254           res.status(response.statusCode).send();
255         }
256       });
257     }
258   });
```

Figura 45. Punto de acceso GET de listado de atributos de todas las entidades.

4.2. Componente

La estructura del componente ha sido realizada utilizando Stencil.js, y su objetivo es mostrar los datos de los dispositivos en tiempo real mediante el uso de SocketIO.

Siguiendo la estructura del componente el desarrollador, sólo tiene que implementar qué datos adicionales se mostrarán y el estilo visual que utilizarán.

Una vez desarrollado el front-end se debe compilar e incluir el componente en el proyecto, indicando tres parámetros: el tipo de entidad, el tipo de dato (individual o lista) y la dirección del microservicio.

El componente puede ser utilizado en cualquier aplicación que utilice HTML, incluyendo aplicaciones que no hagan uso de ningún tipo de librería, ya que el componente se encargará de la gestión y uso de todas las librerías necesarias.

4.2.1. Stencil.js

Stencil.js [82] es un compilador de componentes web desarrollado utilizando JavaScript Vanilla por el equipo de Ionic.

Con el desarrollo de Stencil.js se busca poder crear componentes reutilizables de manera sencilla, que funcionen en el mayor número de navegadores y proyectos web.

Stencil.js genera componentes web que funcionan en cualquier proyecto web, haga uso de algún framework o no (véase Figura 46), es decir, funciona tanto en proyectos de Angular o React como en proyectos HTML sin ninguna librería [83].

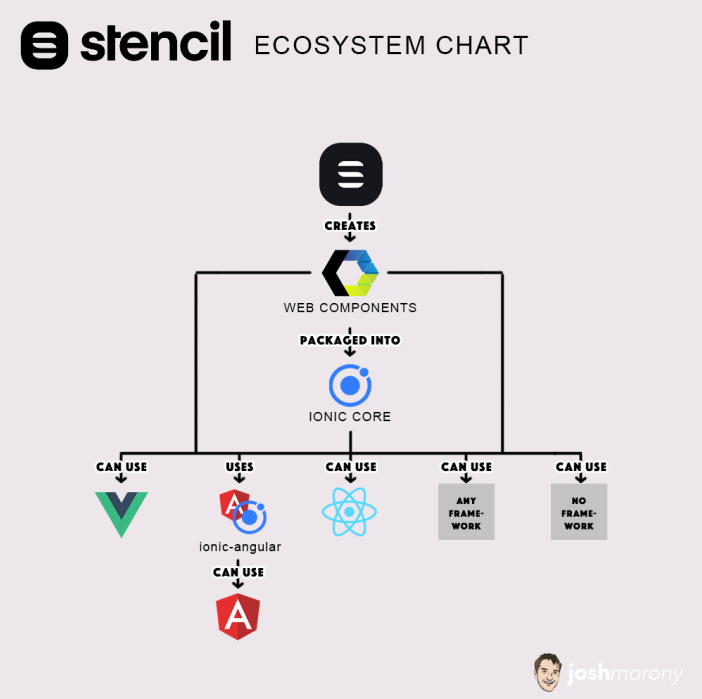

Figura 46. Frameworks web y su relación con Stencil.js [84].

Además, Stencil.js posee las siguientes características [85]:

- Virtual DOM
- Async rendering
- Reactive data-binding
- TypeScript
- JSX

Estas características aportan no solo una mejora de rendimiento como es Virtual DOM, que evita el redibujado de la interfaz tras el cambio de un dato [86], sino que también son una gran ayuda al desarrollador a la hora de crear componentes utilizando Stencil.js con características como JSX para la escritura de HTML desde JavaScript [87].

Todas las características descritas son la razón de que se haya elegido Stencil.js para el desarrollo de los componentes web. Se trata de una tecnología innovadora, que funciona en gran parte de los proyectos web, hagan uso de algún framework o no; posee tecnología de React y Angular que facilita el desarrollo de los componentes y su rendimiento, aunque aún tiene poca comunidad ya que su primera versión estable fue lanzada el 1 de junio de 2019 [88].

4.2.2. Estructura del componente

El componente para funcionar debe disponer de una serie de llamadas y funciones que le permitan comunicarse con el microservicio y almacenar los datos.

Como se muestra en la Figura 47 todo componente deberá de hacer uso de Component, State y Prop del core de Stencil.js, y de SocketIOService del archivo llamado *app-io*, que deberá incluirse en todo proyecto de Stencil.js.

El archivo app-io se encarga de realizar una importación manual de la librería SocketIO ya que la librería no funciona correctamente si se instala mediante *npm*.

```
1  import { Component, State, Prop } from '@stencil/core';
2  import { SocketIoService } from './app-io';
3
4  @Component({
5    tag: 'app-root',
6    styleUrl: 'app-root.css',
7    shadow: true
8  })
9  export class AppRoot {
```

Figura 47. Dependencias de la estructura del componente.

La clase debe instanciar el servicio de SocketIO, una variable (list) para almacenar los datos, y tres parámetros del componente:

- **type:** El tipo de entidad que se va a obtener del bróker de contexto, por ejemplo, Room.
- **entityid:** El id de una entidad en específico en el caso de que solo se quiera obtener una entidad y no una lista de entidades, por ejemplo, Kitchen.
- **filter:** El filtro que va a ser aplicado al obtener las distintas entidades, por ejemplo, temperatura>3.
- **service_url:** La dirección del microservicio, por ejemplo, localhost.

Y en el constructor (véase Figura 48) se debe instanciar el servicio de SocketIO y la lista que contendrá los datos.

```
12  /**
13   * socket io instance
14   */
15  _socketService: SocketIoService = SocketIoService.getInstance();
16
17  @State() list: any[];
18  @Prop() type: string;
19  @Prop() entityid: string;
20  @Prop() filter: string;
21  @Prop() service_url: string;
22
23  constructor() {
24    this._socketService;
25    this.list = [];
26    this.filter = "";
27  }
```

Figura 48. Propiedades, estados y constructor de la estructura del componente.

En el *componentWillLoad* se debe hacer un GET al microservicio para inicializar la lista de datos con entidades y para que el microservicio suscriba el componente a los datos del bróker de contexto (véase Figura 49).

```
29  componentWillLoad() {
30    fetch('http://'+ this.service_url +':3000/subscription?entity=' + this.type + '&id=' + this.entityid + '&queryFilter=' + this.filter)
31      .then((response: Response) => response.json())
32      .then(response => {
33        this.list = JSON.parse(JSON.stringify(response)).entities
34      });
35  }
```

Figura 49. Función componentWillLoad de la estructura del componente.

En el *componentDidLoad* se indica al servicio de SocketIO que escuche el canal de comunicación, teniendo en cuenta el tipo de entidad, el id y el filtro que se han establecido para ser capaz de actualizar los datos cuando el microservicio transmita los nuevos datos por el canal de comunicación (véase Figura 50).

```
37  /**
38   * inital socket usage
39   */
40  componentDidLoad() {
41    this._socketService.onSocketReady(() => {
42      this._socketService.onSocket(this.type + "-" + this.entityid + "-" + this.filter, (msg: string) => {
43        this.list = JSON.parse(JSON.stringify(msg)).entities
44      });
45    });
46  }
```

Figura 50. Función componentDidLoad de la estructura del componente.

En las Figuras 51 y 52 se muestra un ejemplo de uso de la información disponible en la lista de datos, donde se itera sobre cada entidad y sobre cada valor de la entidad para mostrar el nombre de cada entidad y el nombre y valor de sus variables.

```
render() {

  return (
    <div>
      <header>
        <h1>Stencil App Starter</h1>
      </header>

      <main>
        <ul class="list-group">
          {this.list.map((entity) =>
            <li class="list-group-item active">
            <div class="md-v-line"></div><i class="fas fa-laptop mr-4 pr-3"></i>
              {entity.name} - (
              {entity.values.map((entityValue) =>
              <span>{entityValue.name}: {entityValue.value}, </span>
              )}
              )
            </li>
          )}
        </ul>
      </main>

    </div>
  );
}
}
```

Figura 51. Ejemplo de render utilizando la estructura de componentes propuesta.

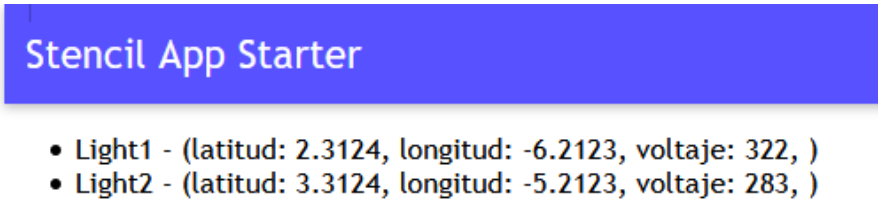

Figura 52. Resultado del render de la estructura del componente propuesta.

4.2.3. Uso del componente

Para poder hacer uso del componente se debe indicar en el archivo stencil.config.ts que la salida es de tipo **dist**, y ejecutar el comando **npm run build** (véase Figura 53).

De esta manera se compilará el proyecto Stencil.js con el componente desarrollado bajo una carpeta llamada dist.

```
TS stencil.config.ts > config
import { Config } from '@stencil/core';

// https://stenciljs.com/docs/config

export const config: Config = {
  globalStyle: 'src/global/app.css',
  globalScript: 'src/global/app.ts',
  outputTargets: [
    {
      type: 'dist',
      // uncomment the following line to disable service workers in production
      // serviceWorker: null
    }
  ]
};

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\Users\Juan Alberto Llopis\Desktop\stenciljsFiware-master> npm run build
```

Figura 53. Archivo stencil.config.ts del componente.

Una vez compilado el componente se debe copiar la carpeta dist en la carpeta proyecto, en el caso de la Figura 54 se trata de un HTML simple sin ningún framework instalado.

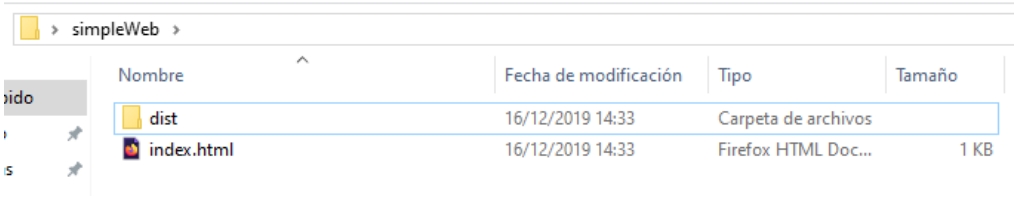

Figura 54. Carpeta raíz de la página web de ejemplo.

Por último, se debe incluir en la aplicación el script **app.js**, la hoja de estilos **app.css** y llamar al componente con los parámetros necesarios.

En la Figura 55 el componente se llama **app-root** y se le ha indicado por parámetro que obtenga las entidades de tipo Light, que obtenga todas las existentes y que el microservicio se encuentra en localhost.

Tras estos pasos el componente ya sería utilizable por la aplicación y cambiándole los parámetros mostrará otros datos, ya sea la información de un conjunto de entidades o la información de una sola entidad (véase Figura 56).

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script src="./dist/app.js"></script>
    <link href="./dist/app.css" rel="stylesheet">
    <style type="text/css">
      body {
        font-family: "Trebuchet MS", Helvetica, sans-serif;
      }

      div {
        position: relative;
        margin: auto;
        width: 50%;
        padding: 40px;
      }

      p {
        font-size: 3.6em;
      }

    </style>
  </head>
  <body>

    <div>
      <app-root type="Light" id="" service_url="localhost"></app-root>
    </div>
  </body>
</html>
```

Figura 55. Ejemplo de importación y uso del componente.

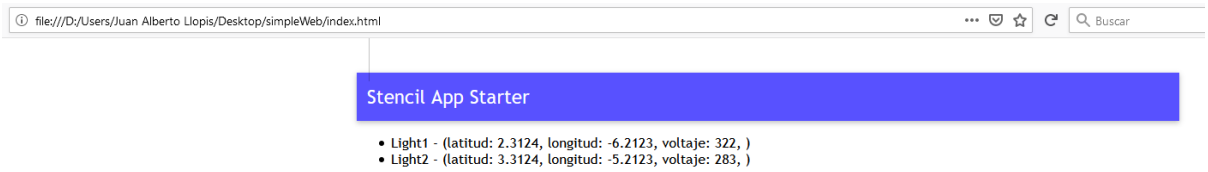

Figura 56. Ejemplo de la web utilizando la estructura de componentes propuesta.

5. Caso práctico

Para mostrar el uso y la utilidad de la propuesta descrita se ha realizado el desarrollo de una aplicación que muestre mediante gráficas y tablas los datos de una serie de sensores ubicados en distintas habitaciones para medir la temperatura y humedad de cada habitación.

Esta aplicación estará formada por componentes que, gracias a la estructura establecida y al microservicio creado, serán capaces de mostrar en todo momento los datos reales de los dispositivos, así como permitir cambiar fácilmente la fuente de datos de la que se alimentará cada uno de los componentes.

Posteriormente se ha realizado una segunda versión que, además de lo descrito, posee un filtro, característica que permitirá elegir con exactitud qué entidades se quieren monitorizar.

También, se ha añadido la posibilidad de cambiar la fuente de datos de la que se alimentan los componentes de la propia interfaz, evitando tener que modificar el código. En la Figura 57 y la Figura 58 se muestran los prototipos de las dos ventanas que conforman la aplicación desarrollada.

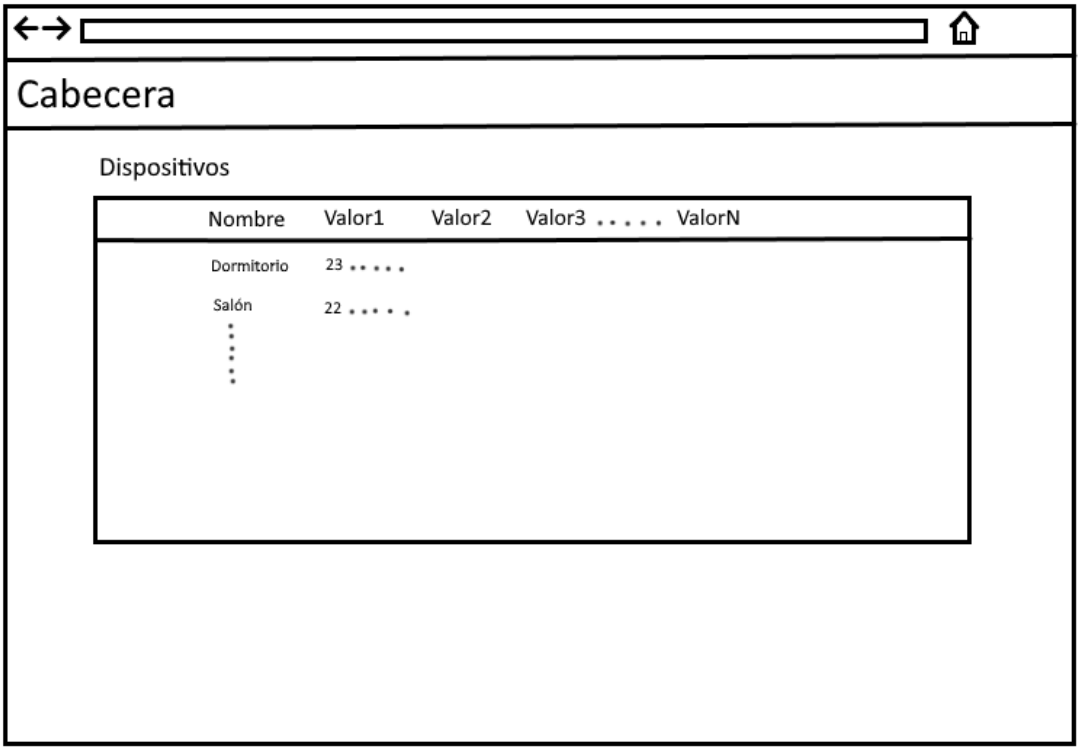

Figura 57. Prototipo de la primera ventana del caso práctico.

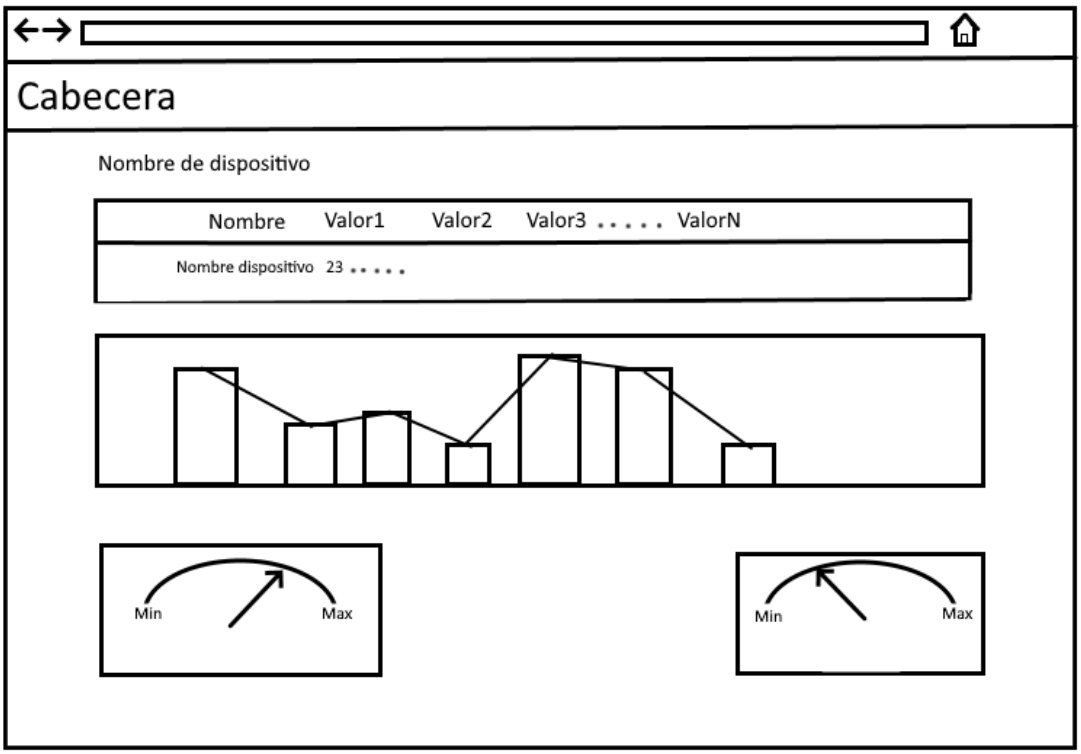

Figura 58. Prototipo de la segunda ventana del caso práctico.

La primera versión de la aplicación tiene como objetivo construir las bases de la aplicación para mostrar el potencial del microservicio y la estructura de los componentes propuesta, esta primera versión no presenta en su estructura el uso de filtros.

La segunda versión se basa en ampliar la aplicación añadiéndole nuevas funcionalidades y mejoras a los componentes con el objetivo de construir una serie de componentes que sirvan de apoyo para futuros desarrollos.

5.1. Arquitectura del sistema

Como se muestra en la Figura 59, el sistema funciona sobre una máquina virtual de OpenStack [89] de la Universidad de Almería bajo el sistema operativo Ubuntu 18.04, la última versión estable LTS, donde se han levantado cuatro contenedores Docker [90]:

- El primer contenedor contiene la base de datos que simula los dispositivos haciendo uso de la tecnología de MongoDB.
- El segundo contenedor contiene el bróker de contexto de FIWARE, encargado de la comunicación entre la base de datos de MongoDB y el microservicio.
- El tercer contenedor tiene como función almacenar el microservicio propuesto que estará en comunicación constante con el bróker de contexto.
- Por último, el cuarto contenedor contendrá la aplicación web sobre un servidor Apache para poder acceder a ella desde cualquier ordenador de la universidad, y utilizando un servicio VPN desde cualquier ordenador fuera de la red.

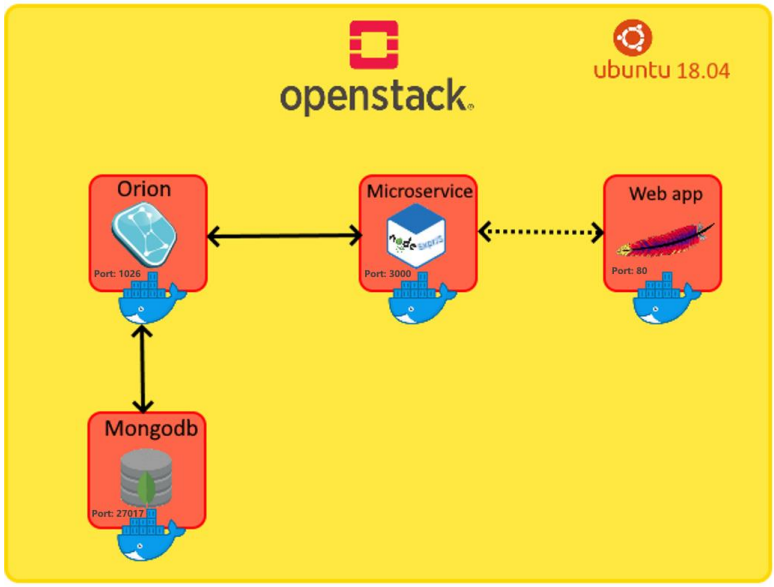

Figura 59. Arquitectura del sistema de la primera versión del caso práctico.

Como se muestra en la Figura 60, en la arquitectura del sistema de la segunda versión se ha realizado una pequeña modificación, ahora los datos de los dispositivos se obtienen de dispositivos reales.

Para lograr esto sin tener que desplegar dispositivos se ha hecho uso de la web de ThingSpeak, una web utilizada para el análisis de datos de dispositivos IoT que permite a los usuarios subir los datos que obtienen sus dispositivos y realizar operaciones y gráficas sobre esos datos obtenidos.

Se ha creado un script que comprueba los datos de temperatura, humedad y presión de una serie de dispositivos y actualiza la información que posee el bróker de contexto con estos datos para que el bróker pueda comunicarlos una vez el microservicio se lo pida, proporcionando los datos de ese instante de los dispositivos de ThingSpeak.

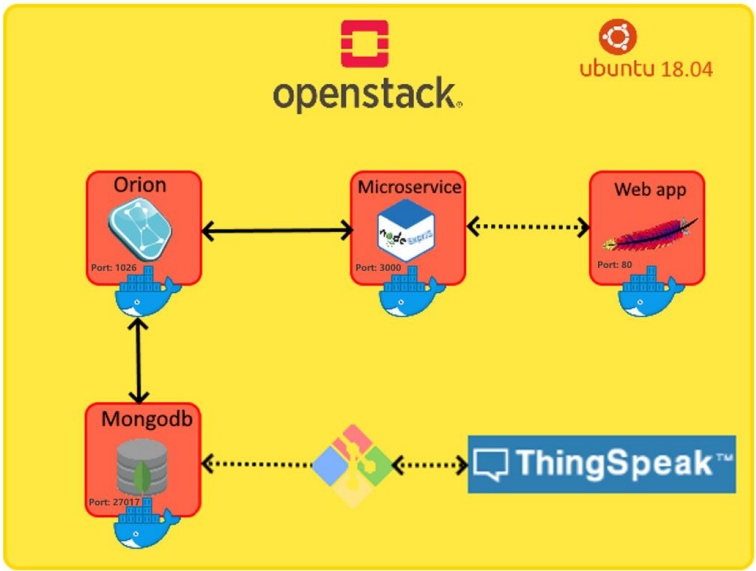

Figura 60. Arquitectura del sistema de la segunda versión del caso práctico.

5.2. Componentes

La primera versión de la aplicación está formada por tres componentes, un componente de tabla y dos componentes de gráficas. Los tres componentes por la naturaleza del sistema muestran datos en tiempo real, pero esta característica se explotará principalmente en el componente de gráfica de líneas, que mostrará los datos en función del tiempo. En la segunda versión, todos los componentes han sido modificados para que hagan uso de los filtros y para adaptarlos a mostrar datos de cualquier tipo de entidad. Adicionalmente, se han hecho cambios para permitir hacer modificaciones que antes se hacían desde el código de la propia interfaz. Además, se han añadido dos componentes nuevos, uno de filtro y otro de lista desplegable.

5.2.1. Componente de tabla

El componente de tabla (véase Figura 61) es puramente CSS, debido a que la estructura de datos se da preparada para su utilización.

Fuera del renderizado la única diferencia con la estructura propuesta es el uso de una nueva propiedad para redirigir al usuario en función del elemento de la tabla que sea pulsado (véase Figura 62).

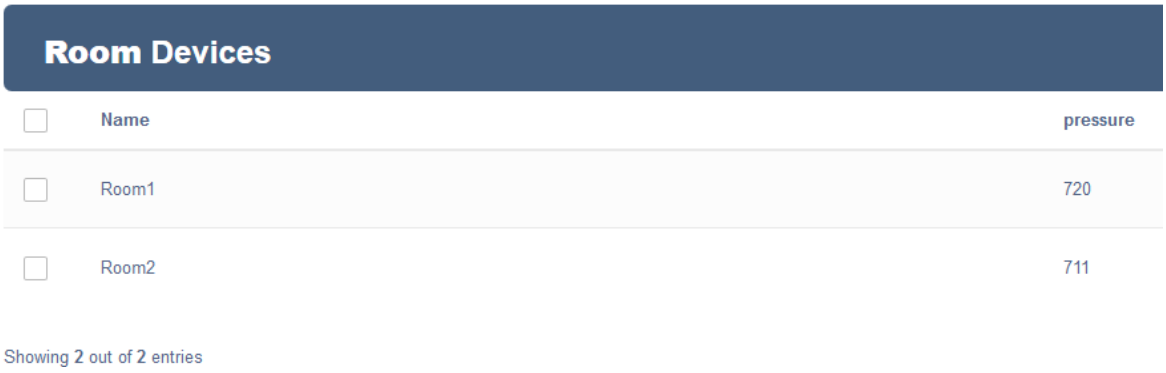

Figura 61. Componente tabla de la primera versión del caso práctico.

```
@State() list: any[];
@Prop() type: string;
@Prop() entityid: string;
@Prop() service_url: string;
@Prop() page_url: string;
```

Figura 62. Variables de la primera versión del componente tabla.

En el renderizado se construye la tabla haciendo uso de las distintas clases y de la estructura de datos del componente, no teniendo que hacer ningún desarrollo de procesamiento de datos (véase Figura 63).

```
return (
    <div class="container body">
        <div class="table-wrapper">
            <div class="table-title">
                <div class="row">
                    <div class="col-sm-6">
                        <h2><b>{this.type}</b> Devices</h2>
                    </div>
                </div>
            </div>
            <table class="table table-striped table-hover">
                <thead>
                    <tr>
                        <th>
                            <span class="custom-checkbox">
                                <input type="checkbox" id="selectAll"/>
                                <label htmlFor="selectAll"></label>
                            </span>
                        </th>
                        <th>
                            Name
                        </th>
                        {this.list.length > 0 ? this.list[0].values.map((entityValue) =>
                            <th>{entityValue.name}</th>
                        ):<th>No items found</th>}
                    </tr>
                </thead>
                <tbody>
                    {this.list.map((entity) =>
                        <tr>
                            <td>
                                <span class="custom-checkbox">
                                    <input type="checkbox" id="checkbox2" name="options[]" value="1"/>
                                    <label htmlFor="checkbox2"></label>
                                </span>
                            </td>
                            <td>{entity.name}</td>
                            {entity.values.map((entityValue) =>
                                <td>{entityValue.value}</td>
                            )}
                            {this.entityid=="" ? <td class="button-td"><a href={this.page_url+"?id="+entit
                        </tr>
```

Figura 63. Render de la primera versión del componente tabla.

En la segunda versión el componente de tabla se ha modificado para que haga uso de los filtros de la segunda versión del microservicio y para que pueda funcionar con el nuevo componente de filtrado. También, se ha modificado su estructura para que acepte entidades de distintos tipos, es decir, entidades con atributos diferentes, lo que permite suscribirse a entidades con ciertos atributos en lugar de a entidades de un tipo en específico. Para ello se ha creado una función que actualiza una lista de atributos, función usada para construir las distintas filas y columnas (véase Figura 64). Las filas se construyen utilizando la lista de atributos, y las columnas haciendo uso de la función *writeRows*. Para cada columna, si la entidad presenta ese atributo colocará su valor, en caso negativo coloca el valor *None* (véase Figura 65).

```
updateAttributeList()
{
    if(this.entityid != "" && this.entityid != 'undefined' || this.type != "" && this.type)
    {
      if(this.list.length > 0)
      {
        this.attributeList = [];
        for(var i = 0; i < this.list[0].values.length; i++)
        {
          this.attributeList.push(this.list[0].values[i].name);
        }
      }else
      {
        fetch('http://'+ this.service_url +':3000/attributeList?type=' + this.entityid)
        .then((response: Response) => response.json())
        .then(response => {
          this.attributeList = response;
        });
      }
}

writeRows(entityAttributes, attribute)
{
  for(var i = 0; i < entityAttributes.length; i++)
  {
      if(entityAttributes[i].name == attribute)
      {
          return <td>{entityAttributes[i].value}</td>;
      }
  }
  return <td>None</td>;
```

Figura 64. Funciones updateAttributeList y writeRows de la segunda versión del componente tabla.

```
        </th>
        {this.attributeList.length > 0 ? this.attributeList.map((attribute) =>
            <th>{attribute}</th>
        ):<th>No items found</th>}
    </tr>
</thead>
<tbody>
    {this.list.map((entity) =>
        <tr>
            <td>
                <span class="custom-checkbox">
                    <input type="checkbox" id="checkbox2" name="options[]" value="1"/>
                    <label htmlFor="checkbox2"></label>
                </span>
            </td>
            <td>{entity.name}</td>
            {this.attributeList.map((attribute) =>
                this.writeRows(entity.values, attribute)
            )}
            {this.entityid=="" ? <td class="button-td"><a href={this.page_url+"?type=" +
        </tr>
```

Figura 65. Escritura de los atributos de las entidades y en la segunda versión del componente tabla.

5.2.2. Componente de gráfica de líneas

Las gráficas han sido desarrolladas utilizando la librería ApexCharts [91], una librería de código abierto que permite crear gráficas para JavaScript, React y Vue. Se ha hecho uso de una adaptación de ApexCharts para Stencil.js que se encuentra disponible en GitHub, y que permite hacer uso de esta librería en la tecnología Stencil.js, aunque con funcionalidades limitadas [92].

En la segunda versión, la gráfica de líneas se ha modificado para poder elegir desde la propia gráfica los datos del atributo que se quieren mostrar (Figura 66 y 67).

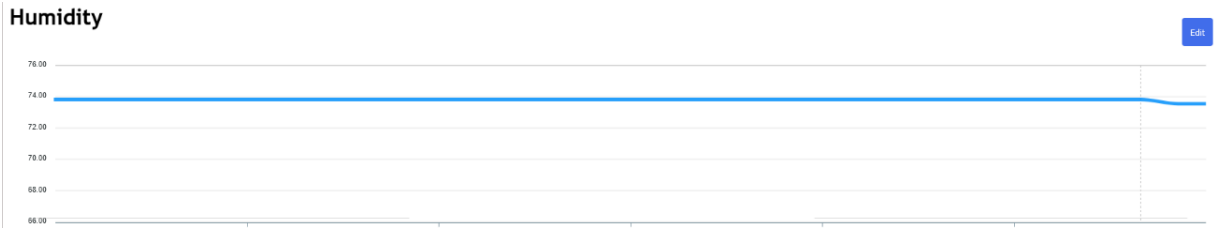

Figura 66. Componente de gráfica de líneas de la segunda versión del caso práctico.

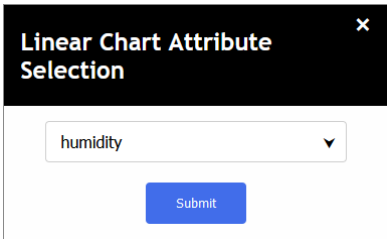

Figura 67. Selección de atributos del componente de gráfica de líneas del caso práctico.

Este componente hace uso de tres nuevas variables, dos locales del componente y una propiedad (véase Figura 68).

```
@State() list: any[];
@State() valueList: any[];
@State() dateList: any[];
@Prop() type: string;
@Prop() entityid: string;
@Prop() service_url: string;
@Prop() data_to_compare: string;

constructor() {
  this._socketService;
  this.list = [];
  this.valueList = [];
  this.dateList = [];
```

Figura 68. Variables y constructor de la primera versión del componente de gráfica de líneas.

Ambas variables locales son para guardar los valores del eje “x” y del eje “y” de la gráfica, donde el eje “x” contendrá los datos del tiempo y el eje “y” los datos a medir.

La propiedad sirve para que el desarrollador indique qué dato del dispositivo quiere medir en la gráfica, por ejemplo, temperatura.

Para esta gráfica se han modificado las funciones de componentWillLoad y de componentDidLoad de la estructura básica propuesta (véase Figura 69), de esta manera cada vez que hay un cambio en el valor que estamos comparando nos guarda ese nuevo dato y la fecha en la que se obtuvo para poder mostrar esos datos en la gráfica.

En el renderizado se crea la gráfica de ApexChart siguiendo la nomenclatura indicada en la documentación e introduciéndole los datos del eje “x” y del eje “y” para que pueda renderizar la gráfica correctamente (véase Figura 70).

```
componentWillLoad() {
  fetch('http://'+ this.service_url +':3000/subscription?entity=' + this.type + '&id=' + this.entityid)
    .then((response: Response) => response.json())
    .then(response => {
      this.list = JSON.parse(JSON.stringify(response)).entities
      this.list[0].values.map((entityValue) =>{
          console.log(entityValue.name)
          console.log(entityValue.value)
       if(entityValue.name == this.data_to_compare)
       {
          this.valueList.push(parseInt(entityValue.value))
          console.log(this.valueList)
          this.dateList.push(new Date().getTime())
       }
       })

    });
}

/**
 * inital socket usage
 */
componentDidLoad() {
  this._socketService.onSocketReady(() => {
    this._socketService.onSocket(this.type + "-" + this.entityid, (msg: string) => {
      this.list = JSON.parse(JSON.stringify(msg)).entities
      this.list[0].values.map((entityValue) =>{
          if(entityValue.name == this.data_to_compare)
          {
              this.valueList.push(entityValue.value)
              this.dateList.push(new Date().getTime())
          }
          })
```

Figura 69. Funciones de componentWillLoad y componentDidLoad de la primera versión del componente de gráfica de líneas del caso práctico.

```
return (
    <div>
      <apex-chart
        type="line"
        width="100%"
        height="300px"
        series={[{
            name: this.data_to_compare,
            data: this.valueList
        }]}
        options={{
            xaxis: {
                categories: this.dateList,
                type: "datetime",
                range: 3000,
                labels: {
                  datetimeFormatter: {
                    year: 'yyyy',
                    month: 'MMM \'yy',
                    day: 'dd MMM',
                    hour: 'HH:mm'
                  }
                }
            },
            chart: {
                animations: {
                  enabled: true,
                  easing: 'linear',
                  dynamicAnimation: {
                    enabled: true,
                    speed: 500
                  },
```

Figura 70. Render de la primera versión del componente de gráfica de líneas del caso práctico.

En la segunda versión, se han creado nuevas variables para poder almacenar la lista de atributos de la entidad seleccionada, qué atributo ha sido seleccionado y la referencia de la ventana modal (véase Figura 71).

En el constructor se han ligado las funciones que hacen uso de la interfaz para poder utilizar la referencia de la ventana modal. Si no se hace este paso, no es posible utilizar la referencia en las funciones.

```
@State() list: any[];
@State() attributeList: string[];
@State() attributeSelected: string;
@State() auxAttributeSelected: string;
@State() valueList: any[];
@State() dateList: any[];
@Prop() type: string;
@Prop() entityid: string;
@Prop() filter: string;
@Prop() service_url: string;

private modalDialog?: HTMLDivElement;

constructor() {
  this._socketService;
  this.list = [];
  this.attributeList = [];
  this.attributeSelected = "";
  this.auxAttributeSelected = "";
  this.filter = "";
  this.valueList = [];
  this.dateList = [];
  this.modalButtonClick = this.modalButtonClick.bind(this);
  this.closeModal = this.closeModal.bind(this);
  this.submitNewAttribute = this.submitNewAttribute.bind(this);
}
```

Figura 71. Variables y constructor de la segunda versión del componente de gráfica de líneas.

Las operaciones realizadas en *componentWillLoad* y *componentDidLoad* se han almacenado en una nueva función llamada updateGraphValues (véase Figura 72).

También, al cargar por primera vez el componente se obtiene la lista de atributos de la entidad sobre la que se están obteniendo los datos.

```
componentWillLoad() {
  fetch('http://'+ this.service_url +':3000/subscription?entity=' + this.type + '&id=' + this.entityid + '&queryFilter=' + this.filter)
    .then((response: Response) => response.json())
    .then(response => {
      this.list = JSON.parse(JSON.stringify(response)).entities
      this.updateAttributeList();
      this.updateGraphValues();
    });
}

/**
 * inital socket usage
 */
componentDidLoad() {
  this._socketService.onSocketReady(() => {
    this._socketService.onSocket(this.type + "-" + this.entityid + "-" + this.filter, (msg: string) => {
      this.list = JSON.parse(JSON.stringify(msg)).entities;
      this.updateGraphValues();
    });
  });
}

updateGraphValues()
{
  if(this.list.length > 0)
  {
    this.list[0].values.map((entityValue) =>{
      if(entityValue.name == this.attributeSelected)
      {
        this.valueList.push(entityValue.value)
        this.dateList.push(new Date().getTime())
      }
    })
  }
}

updateAttributeList()
{
  if(this.list.length > 0)
  {
    this.attributeList = [];
    for(var i = 0; i < this.list[0].values.length; i++)
    {
      if(i == 0) this.attributeSelected = this.list[0].values[i].name
      this.attributeList.push(this.list[0].values[i].name);
    }
  }
}
```

Figura 72. Funciones componentWillLoad, componentDidLoad, updateGraphValues y updateAttributeList de la segunda versión del componente de gráfica de líneas.

Para la ventana modal se han creado funciones para mostrarla y ocultarla, además de una función para confirmar los cambios (véase Figura 73). Además, se ha creado una función que escucha los cambios realizados sobre el componente de la lista desplegable para saber qué elemento de la lista se ha seleccionado y almacenarlo.

```
94    modalButtonClick()
95    {
96        this.modalDialog.style.display = "block";
97    }
98
99    closeModal()
100   {
101     this.modalDialog.style.display = "none";
102   }
103
104   submitNewAttribute()
105   {
106     this.attributeSelected = this.auxAttributeSelected;
107     this.closeModal();
108     this.updateGraphValues();
109     this.valueList = [];
110     this.dateList = [];
111   }
112
113   @Listen('entitySelected')
114   entitySelected(event: CustomEvent) {
115     this.auxAttributeSelected = event.detail;
116   }
```

Figura 73. Funciones modalButtonClick, closeModal, submitNewAttribute y Listener de la segunda versión del componente de gráfica de líneas.

```
123   render() {
124
125     return (
126       <div class="wrapContent">
127
128         <div id="myModal" class="modal" ref={el => this.modalDialog = el as HTMLDivElement}>
129           <div class="modal-content">
130             <div class="modal-header">
131               <span class="close" onClick={this.closeModal}>&times;</span>
132               <h2>Linear Chart Attribute Selection</h2>
133             </div>
134             <div class="modal-body">
135               <app-comboBox combodata={this.attributeList}></app-comboBox>
136               <input class='button -blue center' type="submit" value="Submit" onClick={this.submitNewAttribute}/>
137             </div>
138           </div>
139         </div>
140
141       <h1>{this.getAttributeName()}</h1>
142         <button class='button -blue center editBtn' onClick={this.modalButtonClick}>Edit</button>
143           <apex-chart
```

Figura 74. Render de la segunda versión del componente de gráfica de líneas.

5.2.3. Componente de gráfica radial

El componente de gráfica radial (Figura 75) hace uso de la librería ApexChart. El objetivo de esta gráfica es el de comparar el valor de la variable de una entidad con el máximo valor de esa variable del resto de entidades.

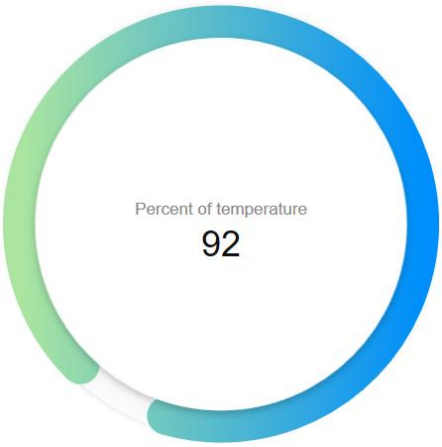

Figura 75. Primera versión del componente de gráfica radial.

La gráfica radial, al igual que la gráfica de líneas, se ha modificado en la segunda versión para poder elegir desde la propia gráfica los datos del atributo que se quieren mostrar (véase Figura 76).

Figura 76. Segunda versión del componente de gráfica radial.

Para este componente se han creado cuatro variables nuevas, dos locales y dos propiedades (véase Figura 77). Las variables locales sirven para almacenar el valor de la variable de la entidad que se va a comparar y para almacenar el máximo valor de esa variable del resto de entidades. Las dos propiedades creadas sirven para saber qué entidad se va a comparar y qué variable nos interesa.

Adicionalmente al renderizado del componente, se ha creado una función para obtener el máximo valor de entre las entidades y para pasar el valor obtenido a porcentaje sobre 100 (véase Figura 78).

```
@State() list: any[];
@State() roomData: any;
@State() maxData: any;
@Prop() type: string;
@Prop() entityid: string;
@Prop() service_url: string;
@Prop() entity_to_compare: string;
@Prop() data_to_compare: string;

constructor() {
  this._socketService;
  this.list = [];
  this.roomData = 0;
  this.maxData = 0;
}

componentWillLoad() {
  fetch('http://'+ this.service_url +':3000/subscription?entity=' + this.type + '&id=' + this.entityid)
    .then((response: Response) => response.json())
    .then(response => {
      this.list = JSON.parse(JSON.stringify(response)).entities
      this.updateGraphValues();
    });
}

/**
 * inital socket usage
 */
componentDidLoad() {
  this._socketService.onSocketReady(() => {
    this._socketService.onSocket(this.type + "-" + this.entityid, (msg: string) => {
      this.list = JSON.parse(JSON.stringify(msg)).entities
      this.updateGraphValues();
    });
  });
}
```

Figura 77. Variables, constructor y funciones componentWillLoad y componentDidLoad de la primera versión del componente de gráfica radial.

```
updateGraphValues()
{
  this.maxData = 0;
  this.list.map((entity) =>
  {
        entity.values.map((entityValue) =>{
            if(entityValue.name == this.data_to_compare)
            {
                if(entity.name == this.entity_to_compare) this.roomData = entityValue.value;
                if(this.maxData < entityValue.value) this.maxData = entityValue.value;
            }
        });
  }
  );
}

valueToPercent (value) {
  return (value * 100) / this.maxData
}

render() {

  return (
      <div>
        <apex-chart
          type="radialBar"
          width="600px"
          options={{
              plotOptions: {
                  radialBar: {
                    startAngle: -135,
                    endAngle: 225,
                     hollow: {
                      margin: 0,
                      size: '70%',
                      background: '#fff',
                      image: undefined,
```

Figura 78. Funciones updateGraphValues, valueToPercent y Render de la primera versión del componente de gráfica radial.

En la segunda versión, se han creado una serie de variables que permiten almacenar la lista de atributos referentes a la entidad seleccionada, el atributo seleccionado y una referencia a la ventana modal (véase Figura 79). También se ligan las funciones que harán uso de la referencia de la ventana modal para poder hacer uso de esa referencia en otras funciones.

```
17    @State() list: any[];
18    @State() attributeList: string[];
19    @State() attributeSelected: string;
20    @State() auxAttributeSelected: string;
21    @State() roomData: any;
22    @State() maxData: any;
23    @Prop() type: string;
24    @Prop() entityid: string;
25    @Prop() filter: string;
26    @Prop() service_url: string;
27    @Prop() entity_to_compare: string;
28
29    private modalDialog?: HTMLDivElement;
30
31    constructor() {
32      this._socketService;
33      this.list = [];
34      this.attributeList = [];
35      this.attributeSelected = "";
36      this.auxAttributeSelected = "";
37      this.filter = "";
38      this.roomData = 0;
39      this.maxData = 0;
40      this.modalButtonClick = this.modalButtonClick.bind(this);
41      this.closeModal = this.closeModal.bind(this);
42      this.submitNewAttribute = this.submitNewAttribute.bind(this);
43    }
```

Figura 79. Variables y constructor de la segunda versión del componente de gráfica radial.

Se ha añadido una función para que al cargar el componente por primera vez cargue los atributos de la entidad seleccionada (véase Figura 80).

```
45  componentWillLoad() {
46    fetch('http://'+ this.service_url +':3000/subscription?entity=' + this.type + '&id=' + this.entityid + '&queryFilter=' + this.filter)
47      .then((response: Response) => response.json())
48      .then(response => {
49        this.list = JSON.parse(JSON.stringify(response)).entities
50        this.updateAttributeList();
51        this.updateGraphValues();
52      });
53  }
54
55  updateAttributeList()
56  {
57    if(this.list.length > 0)
58    {
59      this.attributeList = [];
60      for(var i = 0; i < this.list.length; i++)
61      {
62        if(this.list[i].name == this.entity_to_compare)
63        {
64          for(var j = 0; j < this.list[i].values.length; j++)
65          {
66            if(j == 0) this.attributeSelected = this.list[i].values[j].name
67            this.attributeList.push(this.list[i].values[j].name);
68          }
69          break;
70        }
71      }
72    }
73  }
74
```

Figura 80. Función updateAttributeList de la segunda versión del componente de gráfica radial.

Se han creado funciones para abrir y cerrar la ventana modal, confirmar el atributo seleccionado y una función para conocer el elemento seleccionado en el componente de lista desplegable (véase Figura 81).

```
108   modalButtonClick()
109   {
110       this.modalDialog.style.display = "block";
111   }
112
113   closeModal()
114   {
115     this.modalDialog.style.display = "none";
116   }
117
118   submitNewAttribute()
119   {
120     this.attributeSelected = this.auxAttributeSelected;
121     this.closeModal();
122     this.updateGraphValues();
123   }
124
125   @Listen('entitySelected')
126   entitySelected(event: CustomEvent) {
127     switch(event.detail.split(":")[0])
128     {
129       case "attributeCombo":
130         this.auxAttributeSelected = event.detail.split(":")[1];
131         return;
132     }
133   }
```

Figura 81. Funciones modalButtonClick, closeModal, submitNewAttribute y Listener de la segunda versión del componente de gráfica radial.

Por último, se ha añadido al render la ventana modal y el botón de editar que abre la ventana modal (véase Figura 82).

```
135   render() {
136
137     return (
138       <div class="wrapContent">
139
140         <div id="myModal" class="modal" ref={el => this.modalDialog = el as HTMLDivElement}>
141           <div class="modal-content">
142             <div class="modal-header">
143               <span class="close" onClick={this.closeModal}>&times;</span>
144               <h2>Radial Chart Attribute Selection</h2>
145             </div>
146             <div class="modal-body">
147               <app-comboBox combodata={this.attributeList} comboid="attributeCombo"></app-comboBox>
148               <input class='button -blue center' type="submit" value="Submit" onClick={this.submitNewAttribute}/>
149             </div>
150           </div>
151         </div>
152
153         <button class='button -blue center editBtn' onClick={this.modalButtonClick}>Edit</button>
154         <apex-chart
155           type="radialBar"
```

Figura 82. Render de la segunda versión del componente de gráfica radial.

5.2.4. Componente de filtro

El componente de filtro (véase Figura 83) permite construir cualquier componente, pudiendo seleccionar para ese componente el tipo de entidad que va a cargar y el filtro sobre el que se va a ejecutar.

Además, si se selecciona la opción mostrar todas las entidades permite filtrar sobre ellas para que muestre solo aquellas que contengan cierto atributo, por ejemplo, todas las entidades que tengan el atributo temperatura.

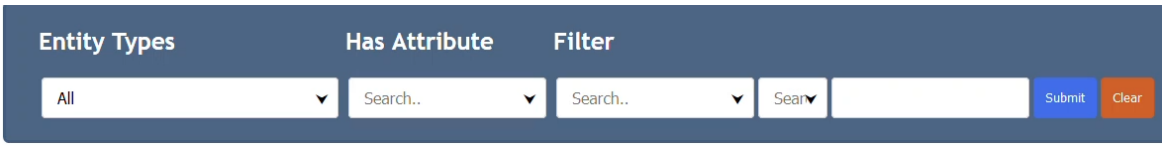

Figura 83. Componente filtro.

Al componente se le deben pasar como propiedades el código HTML del componente a construir y la dirección del microservicio, y tiene como variable el filtro que se va a ejecutar, el listado de tipos de entidades, el listado de atributos, el listado de operaciones posibles, el código HTML del componente a construir y el tipo de entidad seleccionado (véase Figura 84).

```
11    @State() filter: string;
12    @State() typelist: string[];
13    @State() selectedType: string;
14    @State() attributeList: string[];
15    @State() attributeSelected: string;
16    @State() attributeHasSelected: string;
17    @State() operationList: string[];
18    @State() operationSelected: string;
19    @State() htmlRender: string;
20    @Prop() page_url: string;
21    @Prop() service_url: string;
22
23    constructor() {
24      this.htmlRender = this.page_url
25      this.typelist = [];
26      this.attributeList = [];
27      this.operationList = ["==", "!=", "<", ">", "<=", ">="];
28      this.attributeHasSelected = "";
29      this.attributeSelected = "";
30      this.operationSelected = "";
31    }
32
```

Figura 84. Variables y constructor del componente filtro.

Al pulsar el botón Submit sustituye en el código HTML del componente la propiedad type y la propiedad filter, introduciendo el tipo de entidad seleccionado y el filtro introducido (véase Figura 85).

```
33    handleSubmit(e) {
34      e.preventDefault()
35      var selected =  this.selectedType;
36      if(this.selectedType == "All")selected = "";
37      if(this.filter == undefined) this.filter = "";
38      var fullFilter = this.attributeHasSelected + ";" + this.attributeSelected + this.operationSelected + this.filter;
39
40      if(fullFilter == ";") fullFilter = "";
41
42      var renderType = this.page_url.split('type="');
43      var afterType = renderType[1].substring(renderType[1].indexOf('"') + 1, renderType[1].length);
44      this.htmlRender = renderType[0] + 'type="' + selected + '"' + afterType;
45
46      var renderFilter = this.htmlRender.split('filter="');
47      var afterFilter = renderFilter[1].substring(renderFilter[1].indexOf('"') + 1, renderFilter[1].length);
48      this.htmlRender = renderFilter[0] + 'filter="' + fullFilter + '"' + afterFilter;
49      // send data to our backend
50    }
51
52    handleChange(event) {
53      this.filter = event.target.value;
54    }
```

Figura 85. Funciones handleSubmit y handleChange del componente filtro.

Además, la primera vez que se cargue el componente se obtendrán del microservicio el listado de todos los tipos de entidades y el listado de todos los atributos y se mostrarán por defecto todas las entidades existentes (véase Figura 86).

```
componentWillLoad() {
  this.htmlRender = this.page_url

  fetch('http://'+ this.service_url +':3000/typeList')
  .then((response: Response) => response.json())
  .then(response => {
    this.typelist = [];
    this.typelist.push("All");
    response.forEach(element => {
      this.typelist.push(element);
    });
    if(this.typelist.length > 0)
    {
      this.selectedType = this.typelist[0];
      var selected =  this.selectedType;
      if(this.selectedType == "All") selected = "";

      var renderType = this.page_url.split('type="');
      var afterType = renderType[1].substring(renderType[1].indexOf('"') + 1, renderType[1].length);
      this.htmlRender = renderType[0] + 'type="' + selected + '"' + afterType;
      this.updateAttributeList();
    }

  });
}
```

Figura 86. Función componentWillLoad del componente filtro.

La función destinada a obtener la lista de atributos, carga del microservicio todos los atributos existentes en el caso de seleccionar la opción que muestra todas las entidades sin importar su tipo, si no se selecciona esta opción, entonces carga los atributos del tipo de entidad seleccionado (véase Figura 87).

```
updateAttributeList()
{
  var selected =  this.selectedType;
  if(this.selectedType == "All") selected = "";
  fetch('http://'+ this.service_url +':3000/attributeList?type=' + selected)
  .then((response: Response) => response.json())
  .then(response => {
    this.attributeList = response;
  });
}
```

Figura 87. Función updateAttributeList del componente filtro.

También se ha creado una función para limpiar todos los filtros y una función para que el componente escuche al componente de la lista desplegable, con el objetivo de saber cuándo y qué elemento se selecciona de qué lista (véase Figura 88).

En el render se construye la cabecera del filtro, donde se encuentran todas las listas desplegables (véase Figura 89).

Debajo de la cabecera del filtro se construye el componente que se quiere mostrar haciendo uso de los filtros (véase Figura 90).

```
93    clearFields()
94    {
95      this.attributeHasSelected = "";
96      this.attributeSelected = "";
97      this.operationSelected = "";
98      this.filter = "";
99    }
100
101   @Listen('entitySelected')
102   entitySelected(event: CustomEvent) {
103     switch(event.detail.split(":")[0])
104     {
105       case "typeCombo":
106         this.selectedType = event.detail.split(":")[1];
107         this.updateAttributeList();
108         this.clearFields();
109         return;
110       case "attributeHasCombo":
111         this.attributeHasSelected = event.detail.split(":")[1];
112         return;
113       case "attributeCombo":
114         this.attributeSelected = event.detail.split(":")[1];
115         return;
116       case "operationCombo":
117         this.operationSelected = event.detail.split(":")[1];
118         return;
119     }
120   }
```

Figura 88. Funciones clearFields y Listener del componente filtro.

```
render() {

  return (
      <div>
          <form class="filterHeader" onSubmit={(e) => this.handleSubmit(e)}>
            <table>
              <tr>
                <th>
                  <div>
                    <h2 class="inputLabel">Entity Types</h2>
                      <table>
                        <tr>
                          <th>
                            <app-comboBox combodata={this.typelist} comboid="typeCombo" selectedData={this.selectedType}></app-comboBox>
                          </th>
                        </tr>
                      </table>
                  </div>
                </th>
```

Figura 89. Listado de tipos de entidades, filtro de Has Attribute y primer filtro de atributos del render del componente filtro.

```
95
96
97                </form>
98                <h1 innerHTML={this.htmlRender} />
99              </div>
100       );
101     }
102   }
```

Figura 90. Sección de escritura de componentes del render del componente filtro.

5.2.5. Componente de lista desplegable

El componente de lista desplegable (véase Figura 91) se ha creado debido a su utilidad con la nueva versión de la aplicación. Se trata de un componente que se utiliza prácticamente en todos los demás componentes, permitiendo crear cualquier tipo de lista desplegable simplemente pasándole una lista de cadenas de texto.

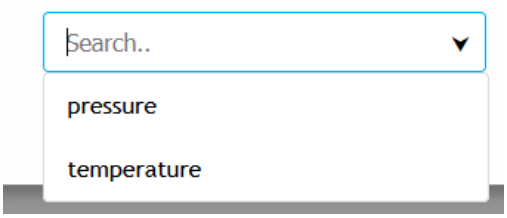

Figura 91. Componente lista desplegable.

El componente de lista desplegable tiene las siguientes propiedades: la lista de elementos a mostrar, el identificador del componente para poder diferenciarlo de otras listas desplegables, el ancho del componente y el dato seleccionado por el componente para que desde fuera del componente se pueda modificar. Como variable tiene el elemento seleccionado junto a una referencia al elemento HTML de la lista y al elemento HTML de búsqueda (véase Figura 92).

```
11  @Prop() combodata: string[];
12  @Prop() comboid: string;
13  @Prop() componentWidth: string;
14  @Prop({ mutable: true }) selectedData: string;
15
16  private dropDownParent?: HTMLDivElement;
17  private dropDown?: HTMLDivElement;
18  private dropDownSearch?: HTMLInputElement;
19
20  constructor() {
21    this.dropDownClick = this.dropDownClick.bind(this);
22    this.filterFunction = this.filterFunction.bind(this);
23    this.selectedData = ""
24  }
```

Figura 92. Variables y constructor del componente lista desplegable.

Posee una función para mostrar la lista desplegable al pulsar en el buscador, una función para el filtrado del buscador y una función que detecta, almacena y transmite el elemento seleccionado de la lista (véase Figura 93 y Figura 94).

```
28  filterFunction() {
29    var filter = this.dropDownSearch.value.toUpperCase();
30    var a = this.dropDown.getElementsByTagName("a");
31    for (var i = 0; i < a.length; i++) {
32      var txtValue = a[i].textContent || a[i].innerText;
33      if (txtValue.toUpperCase().indexOf(filter) > -1) {
34        a[i].style.display = "";
35      } else {
36        a[i].style.display = "none";
37      }
38    }
39  }
```

Figura 93. Función filterFunction del componente lista desplegable.

```
53    clickElement(event)
54    {
55      this.selectedData = event.target.innerText;
56      this.entitySelected.emit(this.comboid + ":" + this.selectedData);
57    }
58
59    @Event({
60      eventName: 'entitySelected',
61      composed: true,
62      cancelable: true,
63      bubbles: true,
64    }) entitySelected: EventEmitter;
```

Figura 94. Funciones clickElement y lanzador de eventos del componente lista desplegable.

Por último, en el render se crea la lista desplegable junto al buscador que incorpora (véase Figura 95).

```
return (
    <div>
        <div class="dropdown" ref={el => this.dropDownParent = el as HTMLDivElement}>
          <div>
            <input ref={el => this.dropDownSearch = el as HTMLInputElement} onFocus={this.dropDownClick} onBlur={this.dropDownClick}
            <span class="input-icon">&#11167;</span>
            </div>
            <div ref={el => this.dropDown = el as HTMLDivElement} id="myDropdown" class="dropdown-content">
                {this.combodata.length > 0 ? this.combodata.map((elementName) =>
                        <a onMouseDown={(event) => this.clickElement(event)}>{elementName}</a>
                    ):<a class="no-display"></a>}
            </div>
        </div>
    </div>
);
```

Figura 95. Render del componente lista desplegable.

5.3. Aplicación

La aplicación ha sido desarrollada usando HTML, sin utilizar ningún framework para poder mostrar el potencial de los componentes de Stencil.js

Dispone de dos ventanas, la primera ventana muestra los datos de todos los dispositivos (véase Figura 96).

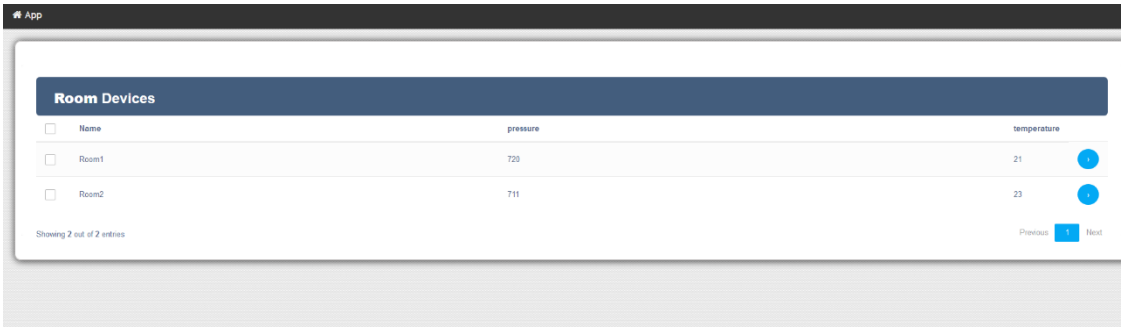

Figura 96. Página principal de la primera versión de la aplicación.

La segunda ventana muestra datos específicos del dispositivo seleccionado y los compara con el resto de dispositivos (véase Figura 97).

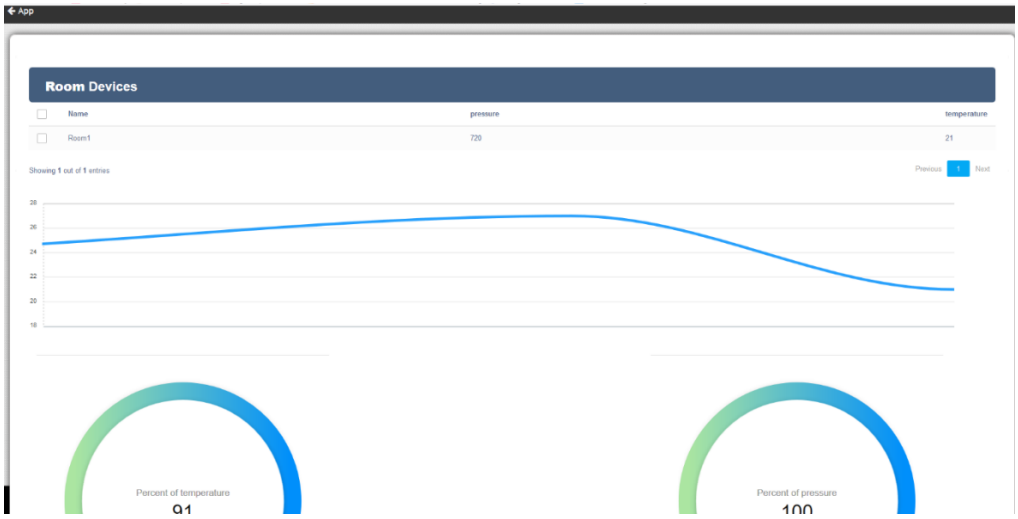

Figura 97. Página de datos de un dispositivo de la primera versión de la aplicación.

La aplicación contiene una carpeta llamada dist con los componentes desarrollados para poder hacer uso de ellos desde el HTML (véase Figura 98). En el HTML de cada página se incluye el script y el CSS de los componentes.

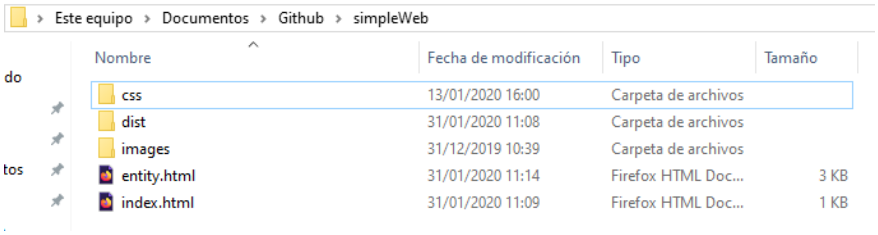

Figura 98. Carpeta raíz de la aplicación.

Cada componente es llamado introduciendo las propiedades del componente mediante etiqueta. Por ejemplo, en la página principal se llama al componente tabla y se le indica el tipo de entidad que interesa mostrar en la tabla, en este caso la entidad Room, la dirección del microservicio, que es localhost y la redirección de la tabla, que es la página entity.html (véase Figura 99).

El entityid está vacío ya que la tabla muestra datos de todas las entidades de tipo Room y no los datos de una entidad en específico.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <script src="./dist/app.js"></script>
    <link href="./dist/app.css" rel="stylesheet">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
    <link rel="stylesheet" type="text/css" href="./css/ownStyle.css">
    <style type="text/css">
     div {
       position: relative;
       margin: auto;
       width: 95%;
       padding: 40px;
     }
    </style>
  </head>
  <body>
   <div class="topnav">
    <a href="index.html"><i class="fa fa-home"></i> App</a>
  </div>
    <div class="parentDiv">
      <app-table type="Room" entityid="" service_url="localhost" page_url="entity.html"></app-table>
    </div>
  </body>
</html>
```

Figura 99. Importación y uso del componente tabla de la página principal de la primera versión de la aplicación.

La página de información individual genera el HTML una vez conocidos los parámetros de entrada.

Los componentes se cargan utilizando los datos de la entidad seleccionada.

En esta página se crea una tabla, una gráfica de líneas y dos gráficas comparativas de la temperatura y la presión (véase Figura 100).

```
document.write('<div class="topnav">');
document.write('<a href="index.html"><i class="fa fa-arrow-left"></i> App</a></div>');
document.write('<div class="parentDiv">');
document.write('<app-table type="Room" entityid="' + getURLParameter("id") + '" service_url="localhost"></app-table>');
document.write('<div style="height: 300px;width: 95%;">');
document.write('<app-realChart id="realChart" type="Room" entityid="' + getURLParameter("id") + '" service_url="localhost" data_to_compare="temperature"></app
document.write('</div>');
document.write('<div class="row"><div class="column">');
document.write('<app-compareChart type="Room" entityid="" service_url="localhost" data_to_compare="temperature" entity_to_compare="' + getURLParameter("id") +
  '"></app-compareChart>');
document.write('</div>');
document.write('<div class="column" style="flex: 20%;">');
document.write('<app-compareChart type="Room" entityid="" service_url="localhost" data_to_compare="pressure" entity_to_compare="' + getURLParameter("id") + '
  "></app-compareChart>');
document.write('</div>');
document.write('</div>');
```

Figura 100. Código de la página de datos de un dispositivo de la primera versión de la aplicación.

Para modificar los datos de los que se alimentan los componentes se deben de cambiar las propiedades type y las propiedades date_to_compare (véase Figura 101).

En el caso de hacer uso de componentes que utilicen entidades específicas también se debe modificar la propiedad entityid.

```
document.write('<div class="topnav">');
document.write('<a href="index.html"><i class="fa fa-arrow-left"></i> App</a></div>');
document.write('<div class="parentDiv">');
document.write('<app-table type="Light" entityid="' + getURLParameter("id") + '" service_url="localhost"></app-table>');
document.write('<div style="height: 300px;width: 95%;">');
document.write('<app-realChart id="realChart" type="Light" entityid="' + getURLParameter("id") + '" service_url="localhost" data_to_compare="voltaje"></app-realChart>');
document.write('</div>');
document.write('<div class="row"><div class="column">');
document.write('<app-compareChart type="Light" entityid="" service_url="localhost" data_to_compare="voltaje" entity_to_compare="' + getURLParameter("id") + '
  "></app-compareChart>');
document.write('</div>');
document.write('<div class="column" style="flex: 20%;">');
document.write('<app-compareChart type="Light" entityid="" service_url="localhost" data_to_compare="latitud" entity_to_compare="' + getURLParameter("id") + '
  "></app-compareChart>');
document.write('</div>');
document.write('</div>');
document.write('</div>');
```

Figura 101. Cambios sobre el código de la página de un dispositivo específico para cambiar los datos fuente de la primera versión de la aplicación.

Una vez realizadas las modificaciones, la aplicación mostrará los datos del nuevo dispositivo en tiempo real (véase Figura 102), facilitando el desarrollo de aplicaciones de ciudades inteligentes, ya que un componente funcionará en cualquier tipo de aplicación web y con cualquier tipo de dispositivo desplegado por la ciudad.

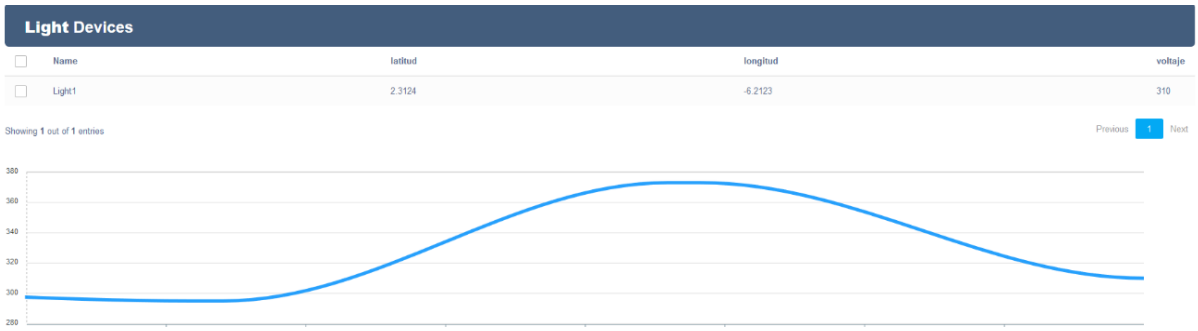

Figura 102. Página de un dispositivo modificada de la primera versión de la aplicación.

En la segunda versión, la página principal permite realizar filtros sobre la tabla, mostrar todas las entidades, solo las entidades de un tipo, las entidades que tengan cierto atributo y las entidades que cumplan cierta condición; no teniendo que modificar el código para cambiar los datos de la tabla (véase Figura 103 y Figura 104).

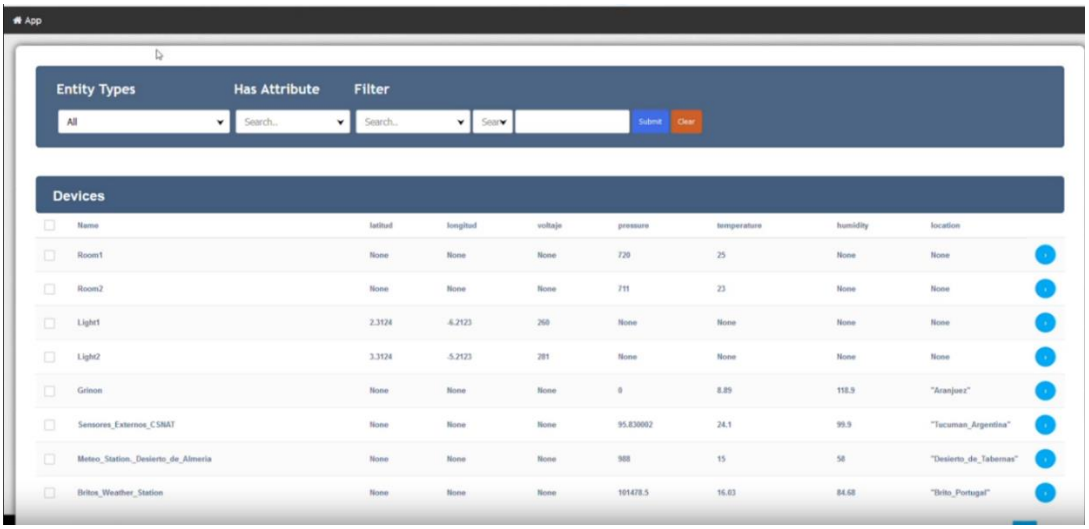

Figura 103. Página principal de la segunda versión de la aplicación.

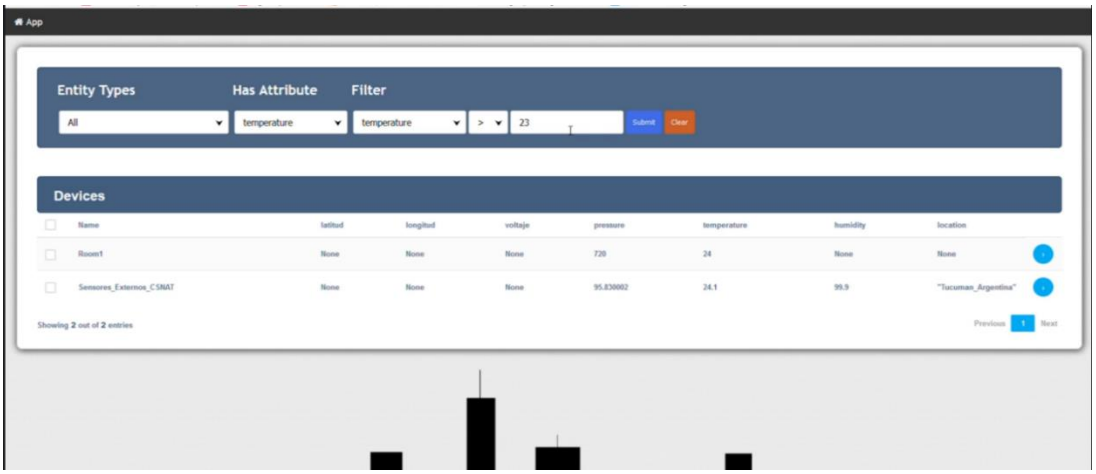

Figura 104. Filtros sobre la página principal de la segunda versión de la aplicación.

Además, los componentes de gráfica también tienen la opción de modificar los atributos sobre los que funcionan (véase Figura 105). Esta característica está disponible pulsando el botón editar que se encuentra en la esquina superior derecha de las gráficas.

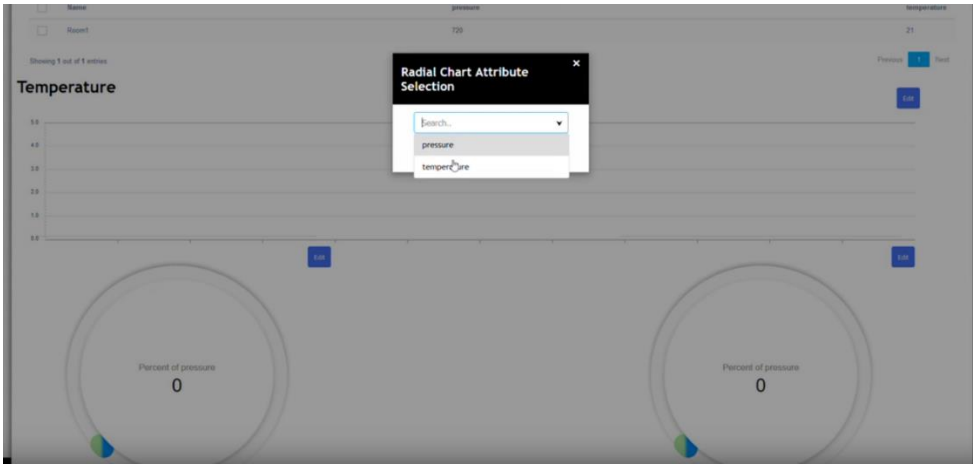

Figura 105. Edición del atributo de la gráfica radial de la página de datos de un dispositivo de la segunda versión de la aplicación.

En el código se ha sustituido el componente tabla por el componente *writeComponent*, indicándole que construya una tabla (véase Figuras 106 y 107).

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <script src="./dist/app.js"></script>
    <link href="./dist/app.css" rel="stylesheet">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
  <link rel="stylesheet" type="text/css" href="./css/ownStyle.css">
    <style type="text/css">
     div {
       position: relative;
       margin: auto;
       width: 95%;
       padding: 40px;
     }
    </style>
  </head>
  <body>
   <div class="topnav">
    <a href="index.html"><i class="fa fa-home"></i> App</a>
  </div>
    <div class="parentDiv">
      <app-table type="Room" entityid="" service_url="localhost" page_url="entity.html"></app-table>
    </div>
  </body>
</html>
```

Figura 106. Cambio en el código de la página principal de la primera versión de la aplicación.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <script src="./dist/app.js"></script>
    <link href="./dist/app.css" rel="stylesheet">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
  <link rel="stylesheet" type="text/css" href="./css/ownStyle.css">
    <style type="text/css">
     div {
       position: relative;
       margin: auto;
       width: 95%;
       padding: 40px;
     }
    </style>
  </head>
  <body>
   <div class="topnav">
    <a href="index.html"><i class="fa fa-home"></i> App</a>
  </div>
    <div class="parentDiv">
      <app-writeComponent service_url="192.168.68.49" page_url='<app-table type="Room" entityid="" filter=""
      service_url="192.168.68.49" page_url="entity.html"></app-table>'></app-writeComponent>
    </div>
  </body>
</html>
```

Figura 107. Código de la página principal de la segunda versión de la aplicación.

6. Conclusiones

En este trabajo se ha realizado la creación de un microservicio, MI-FIWARE, que libera al desarrollador de la carga de trabajo del back-end y del tratamiento de la información de contexto, para ello:

- Se ha identificado un crecimiento en la vertiente de ciudades inteligentes, un crecimiento que ha provocado el aumento de proyectos en la industria mediante el uso de servicios, plataformas y herramientas.
- Se han estudiado y comparado todas las opciones de desarrollo para identificar la opción más adecuada a utilizar.
- Una vez identificada la mejor opción, FIWARE, se ha estudiado para conocer todas sus características con el objetivo de hacer el mejor uso posible, estudio que ha terminado con la propuesta y desarrollo de un microservicio y de una estructura de componente web utilizando Stencil.js.

Este microservicio evita que el desarrollador tenga que realizar el back-end de la aplicación y acompañado de la estructura propuesta permite a los desarrolladores hacer componentes reutilizables solamente implementando front-end.

Estos componentes funcionan en cualquier página web, haga uso de frameworks o no, y por su estructura, desde el momento en el que se crean, son capaces de mostrar datos en tiempo real sin necesidad de realizar desarrollos adicionales.

Tras desarrollar la propuesta se han implementado una serie de componentes y se ha desplegado una página web, que, utilizando los componentes desarrollados, permite la monitorización de los datos de una serie de dispositivos.

Este desarrollo se ha realizado con el objetivo de mostrar la potencia de la propuesta implementada, donde el propio usuario puede parametrizar la aplicación al completo, permitiendo monitorizar cualquier tipo de dato de la manera que le resulte más útil.

El desarrollo realizado permite al desarrollador hacer una implementación rápida y sin necesidad de realizar back-end. Además, le permite crear con facilidad componentes parametrizables desde la parte del usuario y utilizables bajo cualquier aplicación y con cualquier tipo de dato.

Se ha conseguido el objetivo de mejorar la parte destinada al desarrollador y un objetivo que no estaba propuesto: facilitar la personalización de la aplicación desde la propia interfaz, ocasionando que el usuario final pueda seleccionar los datos que se mostrarán.

Como trabajo futuro se propone realizar una estructura de desarrollo de componentes no solo para Stencil.js, sino para otros frameworks como Angular y Vue.js. También, se propone modificar el microservicio para que funcione con otros bróker de contexto y no solo con FIWARE, dando soporte a un mayor número de desarrolladores y no obligándoles a utilizar una tecnología en específico.

Como se ha visto con el tipo de desarrollo por herramientas, el hecho de poder ser utilizado en cualquier tipo de desarrollo supone una gran ayuda a los desarrolladores, porque soluciona las limitaciones que presentan ese tipo de desarrollos. En el caso de MI-FIWARE, el hecho de hacerlo utilizable en otros bróker de contexto puede suponer el hecho de que un bróker pase a ser mejor que otro.

Además, se propone el desarrollo de microservicios de apoyo que permitan funcionalidades como el uso de histórico de datos y el control de acceso. El hecho de tener acceso a un histórico de datos sin necesidad de implementar back-end supondría una gran ayuda a los desarrolladores, el histórico permite crear aplicaciones de ayuda en la toma de decisiones, aplicaciones que hagan uso de inteligencia artificial, etc. Actualmente las gráficas de datos se muestran inicialmente vacías, con este microservicio las gráficas al iniciarse contendrían datos y permitirían ver valores antiguos.

El control de acceso permitiría desarrollar aplicaciones más generales, aplicaciones donde puedan interactuar distintos tipos de usuario. Sin el control de acceso se debe realizar una implementación adicional para restringir la lectura de datos o se debe separar en aplicaciones diferentes. Este microservicio funcionaría como un middleware entre los componentes y el resto de microservicios. Si el componente no tiene acceso a la información que solicita o que el resto de microservicios publica, el microservicio de control de acceso no le dará acceso a los datos. La información de acceso se almacenará en una base de datos a la que solo pueda acceder este microservicio y en ella se almacenarán los distintos roles y los tipos de entidades que pueden consultar.

Estos son solo dos ejemplos de los microservicios de apoyo que se pueden construir, pero el número de microservicios que se pueden incluir en el ecosistema no está limitado debido a la propia definición de microservicios.

Por último, se añadirá a los microservicios compatibilidad con los metadatos del modelo de datos de NGSI. Actualmente se tiene acceso a las entidades y a sus atributos, pero no a la información adicional de los atributos, los metadatos. Inicialmente no se proporcionó soporte a estos elementos ya que no son esenciales en el desarrollo de soluciones IoT, pero le dan un valor adicional permitiendo añadir información que los atributos no pueden proporcionar, por ejemplo, la precisión de un sensor de temperatura o el nombre asociado a las coordenadas de un dispositivo.

7. Bibliografía

1. Telefónica. [Online].; 2017 [visitado 2019 Septiembre]. URL: https://bit.ly/2S8ruGc.
2. Lueth KL. IoT Analytics. [Online].; 2018. URL: https://bit.ly/38kAv4g.
3. United States Census Bureau. [Online]. [cited 2019 Agosto. URL: https://www.census.gov/.
4. International Data Corporation. [Online]. [cited 2019 Agosto. URL: https://www.idc.com/getdoc.jsp?containerId=prUS45197719.
5. MacGillivray C. I-SCOOP. [Online]. [cited 2019 Agosto. URL: https://www.i-scoop.eu/iot-2019-spending/.
6. Lee J, Kao HA, Yang S. Service Innovation and Smart Analytics for Industry 4.0 and Big Data Environment. Procedia CIRP. 2014;: p. 3-8.
7. Lom , Pribyl O, Svitek M. Industry 4.0 as a Part of Smart Cities. In Smart Cities Symposium Prague (SCSP); 2016; Praga. p. 1-6.
8. Scully P. IoT Analytics. [Online].; 2018. URL: https://bit.ly/31BTeWH.
9. Fernández Moniz P, Santana Núñez J, Ortega S, Trujillo-Pino A, Suarez JP, Dominguez Trujillo C, et al. SmartPort: A Platform for Sensor Data Monitoring in a Seaport Based on FIWARE. Sensors. 2016 Mar; 16.
10. Ramparany F, Galan Marquez F, Soriano J, Elsaleh T. Handling smart environment devices, data and services at the semantic level. In IEEE International Conference on Big Data; 2014; Washington, DC. p. 14-20.
11. Da Silva W, Alvaro , G, Afonso R, Dias K, Garcia V. Smart cities software architectures: a survey. In Proceedings of the ACM Symposium on Applied Computing; 2013; Coimbra, Portugal. p. 1722-1727.
12. Caragliu A, Del Bo C, Nijkamp P. Smart cities in Europe. Journal of Urban Technology. 2011 Agosto; 18.
13. Goerlich Gisbert FJ, Cantarino Martí I. Estimaciones de la población rural y urbana a nivel municipal. Estadística española. 2015 Enero; 57(186).
14. Banco Mundial. [Online].; 2018 [cited 2019 Agosto. URL: https://bit.ly/2SvBqsh.
15. Schaffers H, Komninos N, Pallot M, Trousse B, Nilsson M, Oliveira A. Smart Cities and the Future Internet: Towards Cooperation Frameworks for Open Innovation. In Future Internet Assembly 2011: Achievements and Technological Promises.: Springer, Berlin, Heidelberg; 2011. p. 431-446.

16. Javidroozi V, Shah H, Feldman G. Urban Computing and Smart Cities: Towards Changing. IEEE Access. 2019 Agosto; 7.

17. Weiss M. Posicionamento da indústria de TIC para a construção das cidades inteligentes no Brasil: resultados de um levantamento com sete gigantes do setor. Revista Tecnologia e Sociedade. 2019 Abril; 35.

18. Meraki CISCO. [Online]. [visitado 2019 Septiembre]. URL: https://bit.ly/39gs0aD.

19. IBM. [Online]. [visitado 2019 Septiembre]. URL: https://www.ibm.com.

20. CISCO. [Online]. [visitado 2019 Septiembre]. URL: https://www.cisco.com.

21. SAP. [Online]. [visitado 2019 Septiembre]. URL: https://www.sap.com.

22. IBM - Madrid Smart City. [Online].; 2014 [visitado 2019 Septiembre]. URL: https://ibm.co/2S5pRc5.

23. CISCO - Barcelona Smart City. [Online]. [visitado 2019 Septiembre]. URL: https://bit.ly/373xy6N.

24. Gurung A, Prater E. A Research Framework for the Impact of Cultural Differences on IT Outsourcing. Journal of Global Information Technology Management. 2006 Enero; 9.

25. ThingSpeak. [Online]. [visitado 2019 Diciembre]. URL: https://thingspeak.com/.

26. Pratim Ray P. A survey of IoT cloud platforms. Future Computing and Informatics Journal. 2016 Diciembre; 1.

27. Azure - Microsoft. [Online]. [visitado 2019 Septiembre]. URL: azure.microsoft.com.

28. AWS - IoT. [Online]. [visitado 2019 Septiembre]. URL: https://aws.amazon.com/es/iot/.

29. Google Cloud IoT Solutions. [Online]. [visitado 2019 Septiembre]. URL: https://cloud.google.com/solutions/iot/?hl=es.

30. FIWARE. [Online].; 2019 [visitado 2019 Noviembre] 04. URL: www.fiware.org.

31. Singh Gill N. Xenonstack. [Online].; 2019 [visitado 2019 Septiembre]. URL: https://www.xenonstack.com/blog/iot-analytics-platform/.

32. AWS - Connected Home. [Online]. [visitado 2019 Septiembre]. URL: https://aws.amazon.com/es/iot/solutions/connected-home/.

33. AWS - Smart Product Solution. [Online]. [visitado 2019 Septiembre]. URL: https://aws.amazon.com/es/solutions/smart-product-solution/.

34. Ranjan Sahay M, Kumar Sukumaran M, Amarnath S, Palani TND. Environmental Monitoring System Using IoT and Cloud Service at Real-Time. EasyChair. 2019 Mayo;(968).

35. Tärneberg W, Chandrasekaran V, Humphrey M. Experiences Creating a Framework for Smart Traffic Control Using AWS IOT. In IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC); 2016; Shanghai. p. 63-69.

36. Microsoft Azure IoT Suite – Connecting Your Things to the Cloud. [Online].; 2015 [visitado 2019 Septiembre]. URL: https://bit.ly/2UypcS8.

37. Azure - Steam Analytics. [Online]. [visitado 2019 Septiembre]. URL: https://azure.microsoft.com/en-us/services/stream-analytics/.

38. Azure - Logic Apps. [Online]. [visitado 2019 Septiembre]. URL: https://azure.microsoft.com/en-us/services/logic-apps/.

39. Azure - Logic Apps Overview. [Online].; 2018 [visitado 2019 Septiembre]. URL: https://bit.ly/2Sv1q6K.

40. Cirillo F, Wu FJ, Solmaz G, Kovacs E. Embracing the Future Internet of Things. Sensors. 2019 Enero; 19.

41. Popa CL, Carutasu G, Id E, C, Carutasu NL, Dobrescu T. Smart City Platform Development for an Automated. Sustainability. 2017 Noviembre; 9.

42. Google Cloud - IoT Overview. [Online].; 2019 [visitado 2019 Septiembre]. URL: https://bit.ly/2vcOLNZ.

43. Eclipse. [Online].; 2019 [visitado 2019 Noviembre] 04. URL: https://www.eclipse.org.

44. Eclipse - Kura. [Online]. [visitado 2019 Septiembre]. URL: www.eclipse.org/kura/.

45. Eclipse - Kapua. [Online]. [visitado 2019 Septiembre]. URL: www.eclipse.org/kapua.

46. IoT Eclipse. [Online]. [visitado 2019 Septiembre]. URL: https://iot.eclipse.org.

47. Eclipse - Paho. [Online]. [visitado 2019 Septiembre]. URL: https://www.eclipse.org/paho/.

48. Mosquitto. [Online]. [visitado 2019 Septiembre]. URL: https://mosquitto.org/.

49. T. Fernandes J, Perez Abreu D, Velasquez K, Mateus M, Dias Carrilho JA, Gameiro da Silva MC, et al. Building a smart city iot platform: the suscity approach. In Congreso Español de Acústica ; Encuentro Ibérico de Acústica; 2017; La Coruña. p. 557-566.

50. FIWARE Training - Plataforma FIWARE. [Online]. [cited 2019 Agosto. URL: https://bit.ly/2OCJJBk.

51. FIWARE - Smart Cities. [Online]. [visitado 2019 Septiembre]. URL: https://www.fiware.org/community/smart-cities/.

52. FIWARE - Smart Industry. [Online]. [visitado 2019 Septiembre]. URL: https://www.fiware.org/community/smart-industry/.

53. FIWARE - Foundation. [Online]. [visitado 2019 Septiembre]. URL: https://www.fiware.org/foundation.

54. Zahariadis , Papadakis A, Alvarez F, Gonzalez J, Lopez F, Facca F, et al. FIWARE Lab: Managing Resources and Services in a Cloud Federation Supporting Future Internet Applications. In IEEE/ACM 7th International Conference on Utility and Cloud Computing; 2014 Diciembre; Londes. p. 792-799.

55. Martínez A, Onofre H, Estrada H, Torres D, Maquinay O. Diseño y desarrollo de una arquitectura IoT en contexto con la plataforma FIWARE. Research in Computing Science. 2018 Mayo; 147(8).

56. FIWARE Training. [Online]. [visitado 2019 Septiembre]. URL: https://fiware-orion.readthedocs.io/en/master/.

57. FIWARE - Catalogue. [Online]. [visitado 2019 Septiembre]. URL: https://www.fiware.org/developers/catalogue/.

58. P. Kukade P, Kale G. Auto-Scaling of Micro-Services Using Containerization. International Journal of Science and Research. 2015 Septiembre; 4.

59. Thönes J. Microservices. IEEE Software. 2015 Enero; 32(1).

60. Namiot D, Sneps-Sneppe M. On Micro-services Architecture. International Journal of Open Information Technologies. 2014 Agosto; 2(9).

61. Barashkov A. DEV.TO. [Online].; 2018 [visitado 2019 Octubre]. URL: https://bit.ly/31L0nEl.

62. Jara AJ, Serrano M, Gómez A, Fernández D, Molina G, Bocchi Y, et al. Smart Cities Semantics and Data Models. In Proceedings of the International Conference on Information Technology & Systems (ICITS 2018); 2018; Península de Santa Elena, Ecuador. p. 77-85.

63. RAE. [Online]. [visitado 2019 Octubre]. URL: http://lema.rae.es/drae2001/srv/search?id=CSdcCqiYSDXX2mo8Hj4K.

64. K. Dey A. Understanding and Using Context. Personal and Ubiquitous Computing. 2001 Febrero; 5.

65. Garcia de Prado A, Ortiz G, Boubeta Puig J. CARED-SOA: a Context-AwaRe Event-Driven Service Oriented Architecture. IEEE Access. 2017 Marzo; 5.

66. Mena , Corral A, Iribarne L, Criado J. A Progressive Web Application Based on Microservices Combining Geospatial Data and the Internet of Things. IEEE Access. 2019 Julio; 7.

67. Prasad Gochhayat S, Kaliyar P, Conti , Tiwari P, Prasath S, Gupta D, et al. LISA: Lightweight context-aware IoT service architecture. Journal of Cleaner Production. 2018 Diciembre; 212.

68. Butzin B, Golatowski F, Timmermann D. Microservices approach for the internet of things. In Emerging Technologies and Factory Automation (ETFA); 2016 Septiembre; Berlin. p. 1-6.

69. Krylovskiy A, Jahn M, Patti E. Designing a Smart City Internet of Things Platform with Microservice Architecture. In 3rd International Conference on Future Internet of Things and Cloud; 2015; Roma. p. 25-30.

70. Llopis Expósito JA. GitHub-microservicioFiware. [Online]. [visitado 2019 Diciembre]. URL: https://github.com/Jalbladewing/microservicioFiware.

71. Llopis Expósito JA. GitHub-stenciljsFiware. [Online]. [visitado 2019 Diciembre]. URL: https://github.com/Jalbladewing/stenciljsFiware.

72. Node.js. [Online]. [visitado 2019 Noviembre]. URL: https://nodejs.org/es/.

73. Express.js. [Online]. [visitado 2019 Noviembre]. URL: https://expressjs.com/es/.

74. Socket.IO. [Online]. [visitado 2019 Noviembre]. URL: https://socket.io/.

75. Ali Syed B. Beginning Node.js. 1st ed. Apress , editor.; 2014.

76. Capan T. Toptal. [Online].; 2017 [visitado 2019 Noviembre]. URL: https://bit.ly/2tEhj2B.

77. Express.js - Resources. [Online]. [visitado 2019 Noviembre]. URL: https://expressjs.com/en/resources/middleware/body-parser.html.

78. Lombardi A. WebSocket: Lightweight Client-Server Communications. 1st ed. Media O, editor.; 2015.

79. Azaustre C. carlosazaustre. [Online].; 2015 [visitado 2019 Noviembre]. URL: https://bit.ly/2So1aXk.

80. Oreilly. [Online]. [visitado 2019 Noviembre]. URL: https://bit.ly/2UvZSw9.

81. FIWARE - NGSI. [Online]. [visitado 2019 Octubre]. URL: https://fiware.github.io/specifications/ngsiv2/stable/.

82. Stencil.js. [Online]. [visitado 2019 Noviembre]. URL: https://stenciljs.com/.

83. Díaz E. Paradigmadigital. [Online].; 2019 [visitado 2019 Noviembre]. URL: https://bit.ly/31zdM20.

84. Morony J. joshmorony. [Online].; 2019 [visitado 2019 Noviembre]. URL: https://bit.ly/2Sl1EgI.

85. Stencil.js - Docs. [Online]. [visitado 2019 Noviembre]. URL: https://stenciljs.com/docs/introduction.

86. Álvarez Caules C. arquitecturajava. [Online].; 2016 [visitado 2019 Noviembre]. URL: https://bit.ly/2S7MZqr.

87. Fedosejev A. React.js Essentials Publishing P, editor.; 2015.

88. GitHub-Stencil. [Online]. [visitado 2019 Noviembre]. URL: https://github.com/ionic-team/stencil/releases.

89. OpenStack. [Online].; 2019 [visitado 2019 Noviembre] 04. URL: https://www.openstack.org.

90. Docker. [Online].; 2019 [visitado 2019 Noviembre] 04. URL: https://www.docker.com/.

91. ApexCharts. [Online]. [visitado 2019 Diciembre]. URL: https://apexcharts.com/.

92. Karon M. GitHub-Stencil_ApexCharts. [Online]. [visitado 2019 Diciembre]. URL: https://github.com/apexcharts/stencil-apexcharts.

Resumen/Abstract

Dentro de la vertiente de Smart (industria inteligente, hogar inteligente, ciudad inteligente, etc.) el concepto de ciudades inteligentes es el que mayor crecimiento está presentando, ocasionando que haya un mayor número de desarrolladores enfocados en llevar a cabo esta idea y un incremento en las empresas que contribuyen con herramientas, servicios y plataformas. Para apoyar el auge de las ciudades inteligentes se propone la creación de un conjunto de microservicios, MI-FIWARE (referente al uso de microservicios, MI, utilizando FIWARE), que libere al desarrollador de la carga de trabajo del back-end y del tratamiento de la información de contexto.

Palabras clave: Ciudad inteligente, Microservicio, Componente Web, FIWARE.

Within the Smart solutions (Smart Industry, Smart Home and Smart City), the concept of Smart Cities is the one having the greatest growth, inducing an increment in the number of developers focused on carrying out this idea, and a growth in the number of companies that contributes with tools, services and platforms. To support the boom of smart cities, the creation of a set of microservices, MI-FIWARE (referring to the use of microservices, MI, using FIWARE), is proposed to free the developer from the back-end workload and from the processing of context information.

Keywords: Smart City, Microservice, Web Componente, FIWARE.

UNIVERSIDAD DE ALMERÍA