# Diseño del componente software backend orientado a una plataforma IoT diseñada para Smart Campus

Kevin David Arias Figueroa, Juan Felipe Estupiñan Soto

Trabajo de Grado para Optar el título de Ingeniero de Sistemas

Director

Gabriel Rodrigo Pedraza Ferreira

PhD. Ciencias de la Computación

Universidad Industrial de Santander

Facultad de Ingenierías Físico Mecánicas

Escuela de Ingeniería de Sistemas e Informática

Bucaramanga

2019

## Dedicatoria

**“**A mi familia, que es lo que queda cuando todo lo demás falla.

A mis padres, por los incontables esfuerzos que han hecho para formarme como persona, por brindarme su amor y enseñarme que la educación es el camino para un mejor futuro.

A mi hermana, por escucharme, aconsejarme y apoyarme en todos los momentos de mi vida.”

**Juan Felipe Estupiñan Soto**

## Dedicatoria

**“**Primero que todo a Dios, que es quien ha permitido todo esto.

A mi familia, que siempre ha estado ahí para apoyarme en los momentos difíciles y cuyo apoyo me motiva cada día a perseguir mis metas.

A mis padres, a quienes les debo todo, han sido mi ejemplo a seguir, son quienes me han enseñado a ser la persona que soy y su apoyo siempre sin importar la condición me ha llevado a cumplir los objetivos que me propongo.

A mis hermanos por ser mis amigos incondicionales a quienes siempre puedo acudir en busca de consejos.

A mis amigos con quienes he compartido este camino y he vivido momentos inolvidables.”

**Kevin David Arias Figueroa**

## Agradecimientos

A nuestros amigos Camilo, Federico y Jose, por todas las horas invertidas en discutir la mejor manera de desarrollar esta idea, por el enorme apoyo brindado durante la implementación de la misma porque sin ellos, no habría sido posible, y más allá de eso por todos los proyectos que hemos emprendido como equipo y por todos los gratos momentos que compartimos durante nuestra carrera.

A nuestros profesores, por contribuir a nuestra formación como profesionales, y en particular a nuestro profesor Gabriel Pedraza por orientarnos a lo largo del desarrollo del proyecto.

**Tabla de Contenido**

Pág.

**Lista de Tablas**

## Lista de Figuras

## Lista de Apéndices

**(Ver apéndices adjuntos en el CD y pueden visualizarlos en la Base de Datos de la Biblioteca UIS)**

Apéndice A. Javadoc.

Apéndice B. Instalación del proyecto.

Apéndice C. Pruebas del API de servicios.

Apéndice D. Vídeo del caso de uso.

Apéndice E. Código fuente.

## Lista de Siglas

| | |
|---|---|
| **IoT:** | Internet of Things |
| **AWS:** | Amazon Web Services |
| **HTTP:** | Hypertext Transfer Protocol |
| **AMQP:** | Advanced Message Queuing Protocol |
| **JSON:** | JavaScript Object Notation |
| **MQTT:** | Message Queuing Telemetry Transport |
| **REST:** | Representational State Transfer |

## Resumen

**TITULO:** DISEÑO DEL COMPONENTE SOFTWARE BACKEND ORIENTADO A UNA PLATAFORMA IOT DISEÑADA PARA SMART CAMPUS*

**AUTOR:** KEVIN DAVID ARIAS FIGUEROA, JUAN FELIPE ESTUPIÑAN SOTO**

**PALABRAS CLAVE:** IOT, BACKEND, SMART CAMPUS, API.

**DESCRIPCIÓN:**

Los campus universitarios son entornos donde día tras día ocurre una enorme cantidad de sucesos, que se pueden interpretar como una constante producción de cientos de datos, los cuales utilizados inteligentemente pueden traer beneficios para la comunidad educativa. Un sistema de datos centralizado, por ejemplo, podría permitir un acceso rápido al estado de las diferentes zonas de un campus, que traería ventajas en materia de seguridad al poder conocer zonas afectadas por alguna situación de riesgo, o en materia de gestión de recursos al conocer qué zonas de campus están utilizando energía innecesariamente, entre otros.

En este trabajo se presenta una solución Backend para una plataforma IoT, este software se desarrolló con una arquitectura de microservicios de alta disponibilidad elaborada en Java usando el framework Spring Boot, permite la integración de dispositivos y gateways a través de un broker y gracias a su escalabilidad puede manejar grandes volúmenes de datos que se generen a partir de estos, almacenarlos y a través de un API REST ser consultados para el uso que se les quiera dar, además, cuenta con una unidad de persistencia para almacenar la información de los elementos que estén presentes en la plataforma (dispositivos y gateways).

Al final de este proyecto se probó la integración de este software con dispositivos reales instalados en gateways y una plataforma para la administración de los mismos, en un entorno real de producción (esto gracias a otros tres proyectos que se desarrollaron en paralelo a este), comprobando así, que la solución cumple con lo esperado, y funciona como un componente cohesivo en una plataforma IoT para Smart Campus.

---

* Trabajo de grado

**Facultad de Ingenierías Físicomecánicas. Escuela de Ingeniería de Sistemas e Informática

Director: Gabriel Rodrigo Pedraza Ferreira, PhD. Ciencias de la Computación

## Abstract

**TITULO:** DESIGN OF A BACKEND SOFTWARE COMPONENT ORIENTED TO AN IOT PLATFORM DESIGNED FOR SMART CAMPU

**AUTOR:** KEVIN DAVID ARIAS FIGUEROA, JUAN FELIPE ESTUPIÑAN SOTO**

**PALABRAS CLAVE:** IOT, BACKEND, SMART CAMPUS, API.

**DESCRIPCIÓN:**

The campus in the university is an environment where day after day an enormous amount of events occur. These can be interpreted as a constant production of data that, if used correctly and in an intelligent manner, could bring benefits for the educative community. A centralized system of data, for instance, could allow fast access to the condition of different areas inside the campus, bringing benefits in terms of security by knowing the affected area is under a dangerous situation, or in matter of resource management, by knowing which areas of the campus are using unnecessary energy, among others.

This paper presents a Backend solution for an loT infrastructure. The software was developed as a high availability microservices architecture in Java using the framework Spring Boot, allowing the integration of devices and gateways through a broker and, thanks to its scalability, the management of large data volumes generated from said devices, besides the ability to store them and query them through a REST API for ad hoc applications. It also carries a unit of persistence to store the information from the available elements in the infrastructure (devices and gateways).

This project at the end, proved the incorporation of this software with real devices installed on gateways, and a platform for the administration of these, in a real production environment (thanks to the other three projects that were developed in parallel to this one), thus proving the solution meets the expectations, and works as a cohesive component in a loT platform for Smart Campus.


---

*Bachelor Thesis

**Facultad de Ingenierías Físicomecánicas. Escuela de Ingeniería de Sistemas e Informática

Director: Gabriel Rodrigo Pedraza Ferreira, PhD. Computer Science

# Introducción

El internet de las cosas, también llamado IoT es un concepto que surge a finales de la década de los 90, usado por primera vez por Kevin Ashton, profesor del MIT, en 1999, para describir sistemas en los cuales el internet se conecta con el mundo real a través de sensores que captan y envían datos. Desde ese momento, el internet de las cosas ha sido un campo de intenso estudio, que ha traído importantes avances tecnológicos en diferentes sectores de la industria, como la telefonía con los smartphones, la salud con los relojes activity trackers, entre otros.

Ha sido tanto el impacto del internet de las cosas, que incluso han surgido nuevos campos de estudio basado en este concepto, tales como las Smart Cities, que son ciudades llenas de dispositivos que captan datos (sensores) y los utilizan para gestionar otros dispositivos (actuadores) o sistemas, todo esto con el fin de aprovechar la mayor cantidad de los datos generados y mejorar la calidad de vida de sus habitantes. Del mismo modo, surgieron también los Smart Campus, cuyo objetivo es similar al de las Smart Cities, pero aplicado a los Campus Universitarios.

Esto hace útil y hasta cierto punto necesario aprovechar y hacer uso de las herramientas con las que se dispone hoy en día, y no solo para usar datos generados en los procesos de la academia sino también los que se podrían recibir a través de los distintos dispositivos y los que se podrían enviar a los mismos para automatizar desde procesos sencillos como abrir puertas, ascensores, etc., hasta casos de uso más complejo que contribuyan a mejorar la calidad de vida de

los estudiantes, tales como, control de la calidad del aire, monitoreo de pacientes aislados (en el caso de los hospitales universitarios), smart parking, entre otros.

Con el objetivo de implementar una plataforma IoT para Smart Campus, es necesaria una infraestructura que soporte la misma, esta infraestructura debe contar con ciertos elementos cruciales que se mencionaran más adelante en el presente documento, además de características como escalabilidad, alta disponibilidad y extensibilidad.

En este proyecto se presenta la solución backend para una plataforma IoT, la cual se encarga de persistir esta cantidad masiva de datos y de soportar e integrar los diferentes elementos de la arquitectura (dispositivos y gateways), permite también, la comunicación e interacción de los dispositivos con aplicaciones externas, teniendo los datos y la información de la arquitectura centralizada y siempre disponible, con esto se abren las puertas para aprovechar estos datos y utilizarlos ya sea para un fin informativo o también, para llevar a cabo acciones a partir de ellos.

Todo esto descrito con el fin de contribuir a las bases tecnológicas que necesitaría un campus para hace parte de un enfoque creciente hoy en día, el de Smart Campus (campus inteligente).

# 1. Objetivos

## 1.1 Objetivo General

Diseño de un módulo de backend con una arquitectura capaz de soportar e integrar un conjunto de gateways y dispositivos, para almacenar y retornar los datos generados por estos y que puedan ser consumidos por diferentes aplicaciones.

Diseñar un componente software extensible y escalable de tipo backend que permita integrar un conjunto de gateways y dispositivos, analítica de datos y aplicaciones en una plataforma IoT.

## 1.2 Objetivos Específicos

- Construir un API de servicios para responder a las solicitudes de información por parte de diferentes aplicaciones.
- Diseñar una arquitectura escalable que aproveche de la mejor forma los recursos disponibles.
- Implementar el sistema de modo que soporte la concurrencia, para que pueda responder a la demanda de información de varios usuarios simultáneamente.
- Hacer un código robusto, con tolerancia a las excepciones y fallos por parte de los sensores.

# 2. Estado del arte

A continuación, se presenta una breve inducción a algunos de los aportes hechos en los campos del Internet de las Cosas y las arquitecturas de microservicios, así como un poco de historia sobre el desarrollo y los retos a los que estos campos se han enfrentado:

## 2.1 Microservicios

El primer diseño de una arquitectura de software fue el llamado diseño monolítico, en el cual el proyecto software entero se ejecuta desde un solo servidor y en un solo proceso. Algunas de las implicaciones negativas de este tipo de arquitectura es el hecho de que su poder de procesamiento se limita al poder de procesamiento de la máquina servidor, y también que, en el caso de ser necesario hacer un ajuste para subir a producción, la aplicación entera debe ser interrumpida.

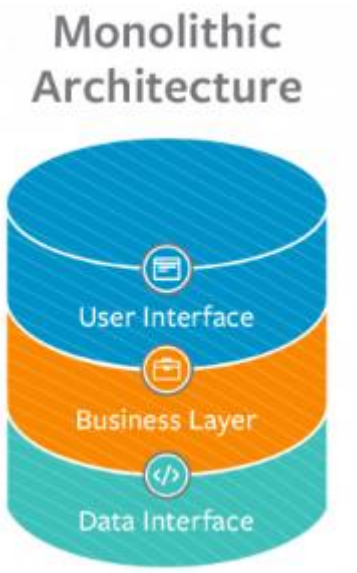


*Figura 1*. Diagrama de una arquitectura monolítica. Tomada de Watts, S., Shiff, L. (2018).

En la búsqueda de solucionar estos problemas, surgen las Llamadas a Procedimiento Remoto (RPC), cuyo objetivo era permitir a los desarrolladores hacer llamadas transparentes a procedimientos en otras máquinas, para facilitar la escalabilidad de los sistemas. Sin embargo, conforme aumentaron el espacio de memoria y el poder de los procesadores, este enfoque demostró no ser muy adecuado, pues las máquinas del sistema se volvían dependientes de demasiada comunicación, lo cual generaba un nuevo problema, que era la sobrecarga de red (Brown, 2016).

Un segundo diseño de arquitectura fue el de las arquitecturas orientadas a servicios, o SOA. Sin embargo, una de las desventajas de este enfoque fue la dependencia de los servicios entre sí, lo cual implicaba la necesidad de coordinar los esfuerzos de los desarrolladores para poder garantizar la correcta operación de los servicios. Aún había mucha interdependencia y la escalabilidad seguía siendo un factor problemático.

Surgen entonces los microservicios, que son una especie de evolución de SOA, y que ofrecen diferentes ventajas frente a esta. Un microservicio es un servicio pequeño y de ejecución independiente al resto de los componentes de una aplicación. Su comunicación se hace a través de algún protocolo ligero como HTTP o AMQP, y cada microservicio se encarga de una parte determinada de la aplicación, lo que se conoce como “bounded context”. (Watts, Shiff, 2018)

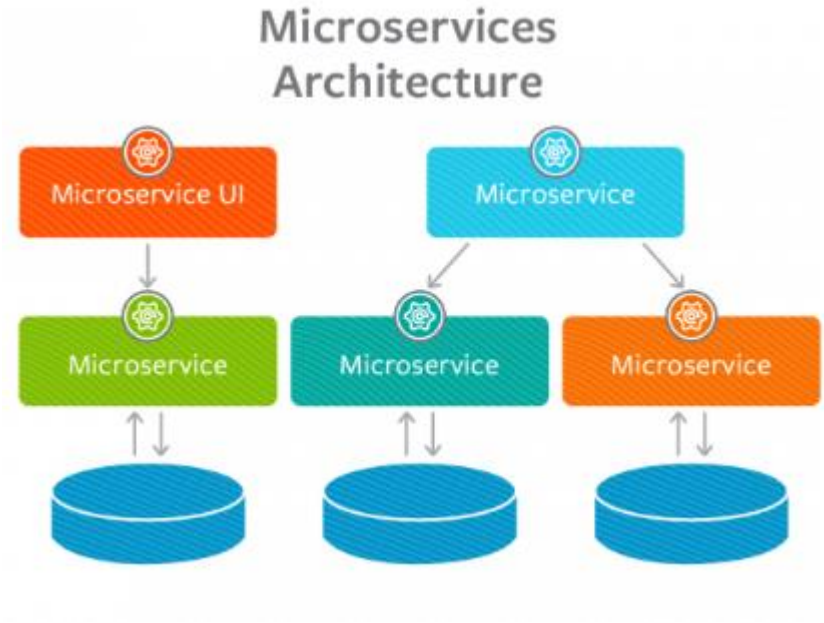


*Figura 2*. Diagrama de una arquitectura de microservicios. Tomada de Watts, S., Shiff, L. (2018).

Este enfoque trajo diversas ventajas en desarrollo y ejecución, tales como: capacidad de dividir el equipo de trabajo en diferentes "dominios", autonomía para desarrollar cada microservicio con el lenguaje preferido/más apropiado para el caso, aplicaciones más escalables, mayor trazabilidad de errores, menor dependencia entre los componentes de un sistema, y capacidad de reutilizar los microservicios en diferentes aplicaciones.

Algunas herramientas de gran utilidad en la evolución de la arquitectura de microservicios son:

Eureka: Es un servicio basado en REST cuyo principal objetivo es el descubrimiento de servicios con el fin de proveer balanceo de cargas (distribución de solicitudes de usuarios entre las diferentes instancias de un microservicio, basada en tráfico, uso de recursos, estado de las instancias, etc). (Netflix, 2014)

Hystrix: Es un servicio de cortocircuito diseñado y utilizado en la arquitectura de microservicios de Netflix. Su objetivo, como indica su nombre, consiste en interrumpir la comunicación con un determinado servicio en caso de reconocer que este tiene una falla, para evitar el consumo innecesario de recursos y un potencial fallo en el sistema. (Netflix, 2018)

## 2.2 Internet de las Cosas

El término IoT se le atribuye a Kevin Ashton, quien es llamado el inventor del IoT ya que fue él quien utilizó el término por primera vez en 1999 para referirse a sistemas que permitieran la conexión de Internet con el mundo real a través de sensores en diversas ubicaciones.

Sin embargo, 50 años atrás, para el año 1949, apareció lo que sería tal vez la primera noción de objetos conectados a Internet, cuando Norman Joseph Woodland, ingeniero del IBM, concibió la idea básica para el primer prototipo del código de barras que se usa en la actualidad. (Press, 2010)

Menos de 5 años después de la afirmación de Ashton, ya existen alrededor de 500 millones de dispositivos inteligentes, y fue tan solo algunos años después, en el 2007, cuando apareció un dispositivo tan icónico con el primer iPhone de Apple. (Evans, 2011)

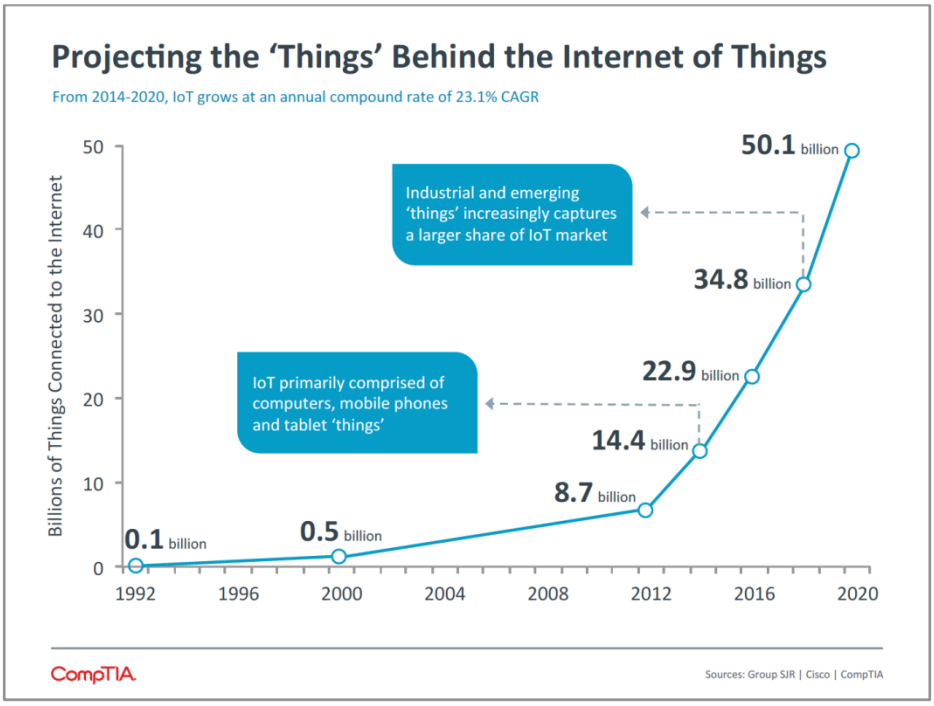


*Figura 3*. Diagrama de cantidad de cosas inteligente vs tiempo. Tomada de CompTIA. (2015).

Ya para el año 2010 se alcanza un nuevo hito en la historia del internet de las cosas, pues gracias al crecimiento explosivo de los smartphones y las tables, por primera vez en la historia hay más dispositivos inteligentes que personas en el mundo. (Evans, 2011)

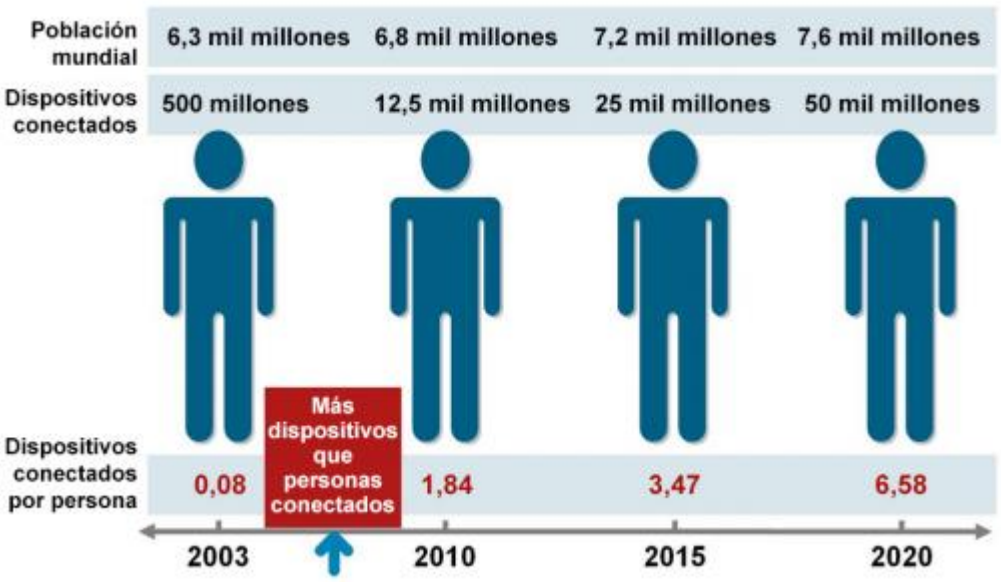


*Figura 4*. Comparación de la cantidad de dispositivos inteligentes frente a la población mundial en el tiempo. Tomada de Davis. (2011).

Algunos de los desarrollos claves para el Internet de las Cosas son:

Internet Protocol (IP): Es el principal protocolo de comunicación entre máquinas conectadas en una red de computadoras.

Advanced Message Queuing Protocol: Es un protocolo de comunicación de la capa de aplicación, ligero y de mensajería basada en publish-subscribe.

Algunas de las grandes compañías de las tecnologías de la información han optado por crear sus propias plataformas IoT en la nube donde ofrecen diferentes soluciones de software, servicios y control de servicios a nivel cloud y edge para conectar dispositivos y almacenar, procesar, y analizar los datos que estos dispositivos generan. Algunas de estas plataformas son: Google Cloud IoT, AWM, de Amazon, y Azure, de Microsoft.

# 3. Marco de Referencia

## 3.1 Internet de las Cosas (IoT)

El internet de las cosas es un concepto relacionado con la conexión de cualquier dispositivo al Internet. Más específicamente, se refiere a la red de dispositivos físicos que poseen una dirección IP y por lo tanto pueden comunicar datos entre sí.

IoT es uno de los temas más populares en el sector de las tecnologías de la información, gracias a su gran potencial para cambiar completamente la forma en que millones de procesos se llevan a cabo hoy en día. En el año 2010, la cantidad de objetos conectados al internet superó la población humana, y para el año 2020 se espera llegar a la cifra de 50.000 millones. (CompTIA, 2015)

Además, esta área de estudio está siendo fuertemente investigada en sectores como la salud, la manufactura y los negocios, y se espera que para el 2020 más del 65% de compañías estén utilizando IoT de una u otra manera. IoT ha mostrado, además, ser altamente redituable, pues el 94% de las empresas que han decido invertir en IoT ya han recuperado su inversión. (Bera, 2019).

En esencia, IoT tiene dos componentes principales. La primera de ellas es el internet, que actúa como infraestructura de comunicación. La segunda son las cosas, un concepto inherente a esta tecnología que se refiere a las entidades físicas que poseen un identificador en la red que les permite recibir y enviar datos a través de ella, y un sistema embebido que les permite procesar estos datos y/o generar una respuesta a los mismos (Bharadwaj, 2016). Las cosas pueden

clasificarse, según la dirección de los datos que manejan, en sensores, que generan y envían datos a la red; actuadores, que reciben y procesan estos datos de alguna manera; y dispositivos que cumplen ambas funciones.

### 3.2 Smart Campus

El Internet de las cosas, si bien es un concepto con un gran potencial, es también un concepto bastante abstracto. Es por esto que durante su evolución han surgido diferentes campos de aplicación que intentan adoptar IoT para hacerlo más concreto, siendo uno de estos Smart Campus, cuyo objetivo es similar al de las Smart Cities (aprovechamiento eficiente de recursos para mejorar la calidad de vida de sus habitantes) pero aplicado a los Campus Universitarios.

Algunas de las preguntas que Smart Campus ataca hoy en día son: ¿Cómo facilitar y/o mejorar la experiencia de la vida en el campus para la comunidad universitaria?, ¿Cómo aprovechar de mejor manera los recursos que la universidad posee?, ¿Cómo proveer un entorno más seguro, menos propenso a situaciones de riesgo, pero también más preparado para responder ante ellas? Y todas estas preguntas se pueden, de una u otra forma, abordar desde la perspectiva de la información. (Ruckus, s.f)

Los campus universitarios son entornos donde día tras día ocurre una enorme cantidad de sucesos, que se pueden interpretar como una constante producción de cientos de datos, los cuales utilizados inteligentemente pueden traer beneficios para la comunidad educativa. Un sistema de datos centralizado, por ejemplo, podría permitir un acceso rápido al estado de las diferentes zonas de un campus, que traería ventajas en materia de seguridad al poder conocer zonas afectadas por alguna situación de riesgo, o en materia de gestión de recursos al conocer qué zonas de campus están utilizando energía innecesariamente, entre otros.

## 3.3 Arquitectura

Para construir una plataforma tecnológica que soporte un Smart Campus, se tienen que tener en cuenta las necesidades específicas de la institución en la cual se quiere implementar, pero además se deben considerar algunos parámetros comunes, como el hecho de que, en términos generales, las instituciones de educación superior requieren de una arquitectura de Software robusta para soportar la cantidad que datos que manejan. Una buena arquitectura es un factor que puede marcar la diferencia entre una solución IoT exitosa y una fallida.

Una arquitectura software para IoT, y en especial para Smart Campus, debe cumplir con los siguientes requerimientos:

**3.3.1. Escalabilidad.** Considerando que actualmente existen más de 3 dispositivos conectados por cada persona en el mundo, y que esta cifra puede ser aún mayor en los campus universitarios, no es difícil imaginar que las soluciones IoT deben estar diseñadas teniendo en una enorme cantidad de usuarios potenciales. La arquitectura debería entonces tener la capacidad de soportar aumentos en su tamaño sin comprometer su funcionamiento. (Castro, 2016) Factores que hay que tener en cuenta para esto son: el diseño de un sistema de almacenamiento que permita soportar un gran número de datos, un código lo más optimizado posible para aprovechar eficientemente los recursos del sistema y así responder de la mejor manera a la concurrencia, y un diseño de red que facilite la configuración y puesta en marcha de nuevos dispositivos.

**3.3.2. Extensibilidad.** La naturaleza de las cosas utilizadas en el contexto de Smart Campus es muy amplia, abarcando desde sensores de temperatura y movimiento, hasta aires acondicionados, dispositivos móviles, pantallas, entre otros. En este sentido, extensibilidad se refiere a la capacidad del sistema para expandirse de modo que pueda comenzar a manejar más y

más de estos diferentes dispositivos conforme surja la necesidad. El sistema debe poder adaptarse a medida que se introduzcan nuevas tecnologías. (Kirkland, 2017)

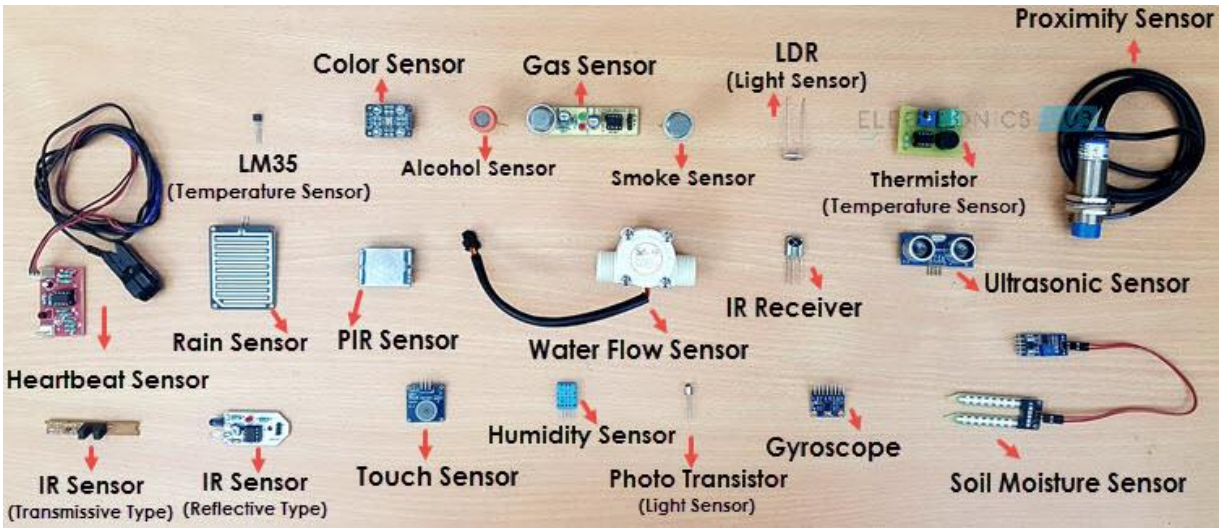


*Figura 5*. Diferentes tipos de sensores según la magnitud física que detectan. Tomada de Teja. (2017).

**3.3.3. Desempeño.** Evidentemente, para que un sistema IoT sea considerado competente, debe tener unos tiempos de respuesta adecuados a las solicitudes de sus clientes. En esto el modelo de datos juega un papel importante, ya que, si la información está correctamente dispuesta, podrá ser consultada y modificada mucho más eficientemente.

**3.3.4. Fiabilidad.** En un sistema IoT (y en un sistema software en producción en general) pueden ocurrir cualquier tipo de situaciones inesperadas. Un software de calidad debe ser fiable, y esto quiere decir principalmente dos cosas: debe prevenir fallos, y debe tolerarlos. Prevenir fallos quiere decir evitar que se introduzcan fallos en el sistema durante su desarrollo, principalmente ocasionados por errores de diseño. Tolerar fallos quiere decir tener la capacidad de continuar

funcionando, al menos durante un tiempo, a pesar de tener fallos. La tolerancia a errores es una parte esencial de una arquitectura robusta. (Blet, s.f)

### 3.4 Microservicios

Una buena alternativa de diseño de arquitectura para una solución IoT son los microservicios. Este tipo de arquitectura consiste en un conjunto de servicios.

Algunas de las características de esta arquitectura son:

- Independencia: En una arquitectura de microservicios, los servicios son pequeños, cada uno independiente de los otros, y débilmente acoplados. Esto implica también que pueden ser desplegados y actualizados independientemente.
- Cada servicio tiene su propio código, lo cual permite que diferentes grupos de trabajo manejen diferentes servicios.
- Cada servicio es responsable de persistir sus datos.
- Los servicios se comunican entre sí a través de un API bien definida.
- Los servicios no necesitan estar escritos con el mismo framework o en el mismo lenguaje. (Wasson, M., Roberts, J., Wilson, M., Buck, A. & Celarier, S., 2018)

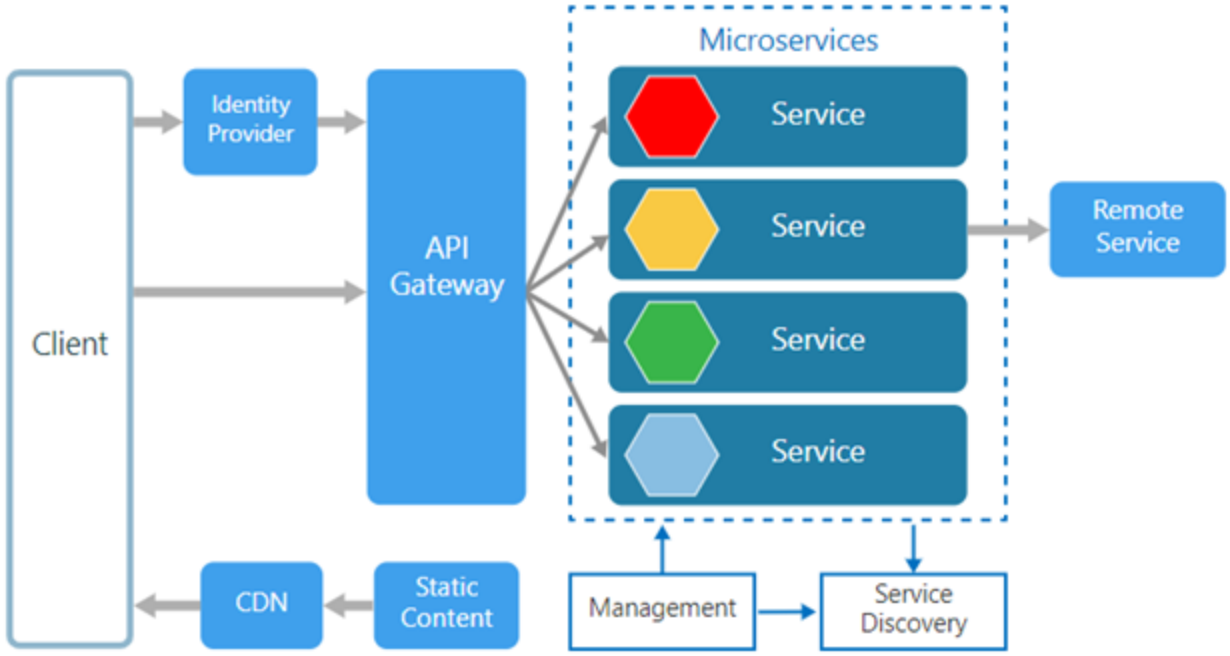


*Figura 6*. Diagrama del flujo de datos en una arquitectura de microservicios. Tomada de Wasson, Roberts, Wilson, Buck, & Celarier. (2018).

Otros componentes, como se puede observar en la imagen anterior, son:

- API Gateway: Su función es ser un punto de acceso único de los clientes a los servicios, redireccionando las solicitudes a sus servicios correspondientes
- Service Discovery: Mantiene una lista de servicios que localiza en la red con el propósito de balancear cargas.
- Management: Balancea la carga de solicitudes entre las instancias de los servicios, según su disponibilidad.

## 3.5 Representational State Transfer

Representational State Transfer, más popularmente conocido como REST, es un estándar de comunicación de la capa de aplicación, cuyo objetivo es facilitar la comunicación entre diferentes sistemas. Las características principales de este estándar son:

- La implementación del cliente y el servidor están separadas, lo que significa que el código del lado del cliente y el del servidor pueden cambiar independientemente sin que el otro lado se entere.
- Los sistemas que utilizan REST (también llamados sistemas RESTful) son sistemas sin estado, lo que quiere decir que el servidor no requiere saber nada acerca del estado del cliente y viceversa, así, la comunicación no depende de mensajes previamente enviados.
- Dado que REST utiliza HTTP como protocolo de comunicación, esta funciona de la forma solicitud-respuesta. (Codecademy, s.f.)

**3.5.1. Solicitudes Rest.** En REST, los sistemas cliente hacen solicitudes al servidor, las cuales consisten en:

- Un endpoint, es decir, una URL a un recurso
- Un encabezado que define metadatos de la solicitud
- Un cuerpo (opcional) con datos de la solicitud
- Un verbo HTTP. (Codecademy, s.f.)

**3.5.2. Verbos HTTP.** Los principales verbos HTTP son:

- GET: Recupera información de un recurso, no se usa para ningún tipo de modificación.
- POST: Crea un nuevo recurso en una colección.

- PUT: Ejecuta una actualización de un recurso (si no existe previamente, puede o no crearlo).
- DELETE: Se utiliza para eliminar recursos. (HTTP Methods, s.f.)

### 3.6 Advanced Message Queuing Protocol

Advanced Message Queuing Protocol, o AMQP, es otro protocolo de la capa de aplicación que permite la conexión entre clientes y brokers, que actúan como un middleware para mensajes. El objetivo de esta tecnología es permitir que las aplicaciones se conecten entre sí y envíen mensajes, asegurándose de que estos serán recibidos incluso si el destinatario de un mensaje no se encuentra disponible en un momento determinado.

Esto lo consigue a través de una cola de mensajes, la cual provee un almacenamiento temporal para los mensajes en caso de que el programa no esté conectado para recibir. Una cola es una secuencia de mensajes esperando para ser procesados. (Johansson, 2014)

La arquitectura básica de una cola de mensajes es la siguiente:

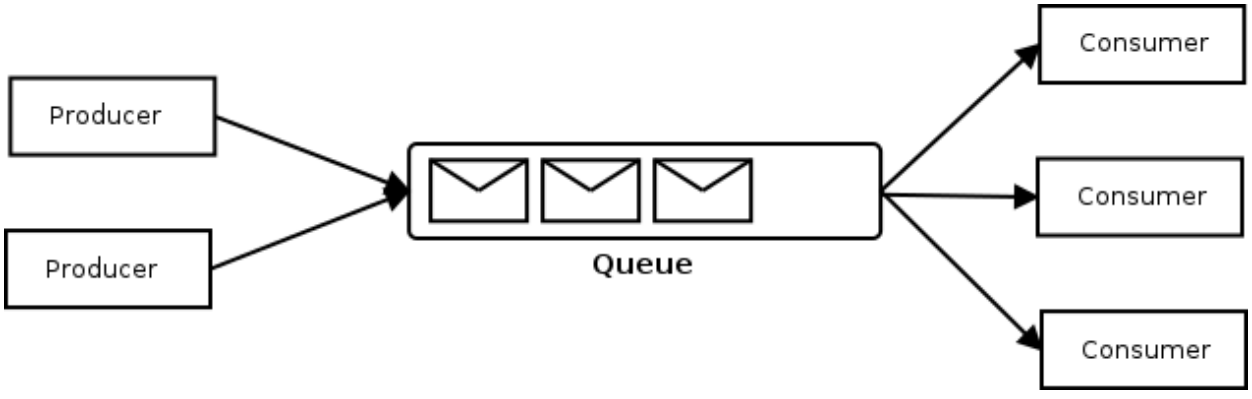


*Figura 7*. Diagrama de la arquitectura de una cola de mensajes. Tomada de Apache. (s.f).

Existen aplicaciones llamadas productores que envían mensajes a uno o varios brokers. Estos brokers pueden contener una o más colas, cada una con un identificador. Otro conjunto de aplicaciones llamadas consumidores se subscriben a una determinada cola y de este modo, cuando un mensaje es publicado en la cola, lo recibe el consumidor más adecuado teniendo en cuenta factores como su disponibilidad y si está o no procesando otro mensaje. (Apache, s.f.)

Una alternativa a las colas de mensajes son los topics, los cuales se diferencian de las primeras en que el mensaje se envía a todos los consumidores suscritos al tópico y no solo al primero.

### 3.7 Edge Computing

Edge computing es un concepto similar al de Cloud Computing, con la diferencia de que la computación, en vez de ocurrir en un servidor remoto (nube), ocurre en los límites de una red interna (edge). A pesar de que podría parecer una alternativa al Cloud Computing, el Edge Computing trabaja en conjunto este para potenciar las soluciones IoT. pues los dispositivos Edge actúan como si fuesen un gateway para facilitar la conectividad de los dispositivos IoT a la nube, la cual tiene un poder de procesamiento mucho mayor al de los dispositivos Edge. (Skaria, S. 2016).

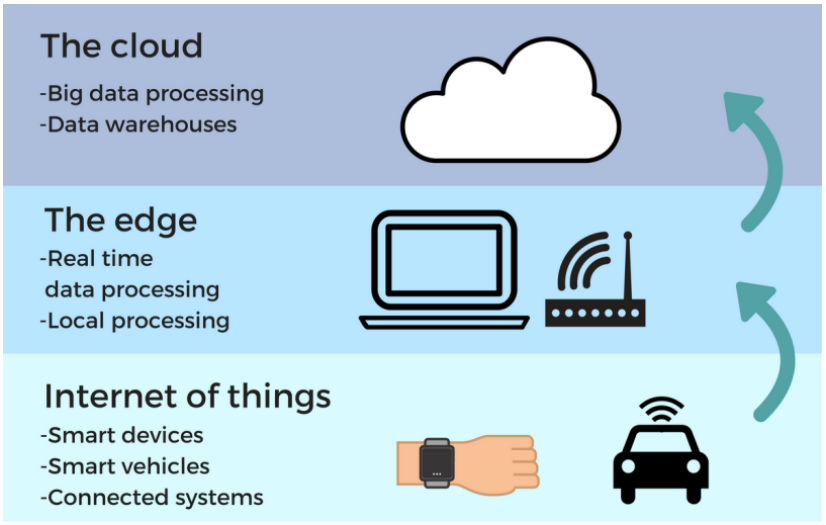


*Figura 8*. Edge Computing y Cloud Computing. Tomada de Solway Communications. (2018).

## 4. Marco metodológico

La metodología que se desarrolló en este proyecto se compuso de cinco fases. La primera fue una fase de capacitación tecnológica, posteriormente vinieron 3 fases iterativas de definición de la arquitectura, prototipado y pruebas de validación del prototipo, y se finalizó con una etapa de implantación.

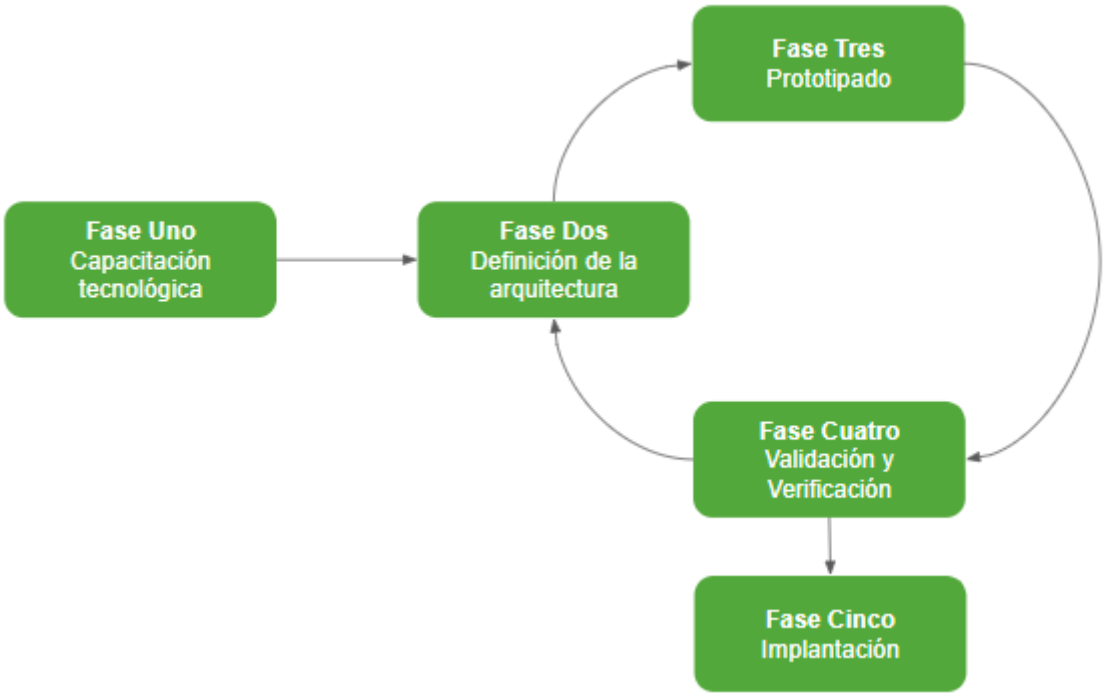


*Figura 9.* Metodología.

### 4.1 Fase 1: Capacitación tecnológica

En esta fase se estudiaron los conceptos clave del proyecto, tales como Internet de las cosas y Smart Campus. También se realizaron cursos para el aprendizaje en el manejo de diferentes herramientas tecnológicas como frameworks y lenguajes que contribuyeran e hicieran posible la buena realización de las actividades de las fases siguientes.

Actividades realizadas:

- Apropiación de los conceptos claves en IoT.
- Realización de cursos para el aprendizaje en el uso de herramientas útiles en el desarrollo IoT.

### 4.2 Fase 2: Definición de la arquitectura

En esta fase se definió la arquitectura de la solución IoT orientada a Smart Campus la cual incluye claramente los módulos que la componen, los protocolos de comunicación que se usaron y el flujo de datos desde su generación hasta su representación a usuarios finales.

Actividades realizadas:

- Definición de la base de datos.
- Comparación de protocolos de comunicación teniendo en cuenta factores como consumo de recursos, velocidad y confiabilidad.
- Definición clara de las funciones del backend.

### 4.3 Fase 3: Prototipado

En esta fase se desarrolló el backend basado en la arquitectura propuesta en la fase anterior. Se hicieron las iteraciones correspondientes y al final se obtuvo un API integrable con los demás elementos de la arquitectura y que cumple con los objetivos propuestos.

Actividades realizadas:

- Diseño del componente backend.
- Pruebas de funcionamiento del componente backend.

### 4.4 Fase 4: Validación y verificación

En esta fase se verificó el desarrollo realizado con los requerimientos establecidos en fases anteriores, para así tener consistencia y control sobre los ajustes que fuesen necesarios, también

se validó el software por medio de pruebas para comprobar que cumplía con todos los requerimientos funcionales y no funcionales anteriormente definidos.

Actividades realizadas:

- Diseño de pruebas.
- Aplicación de pruebas.
- Identificación de la brecha entre el desarrollo actual y esperado.

### 4.5 Fase 5: Implantación

Al final de las iteraciones y luego de validar y verificar todos los módulos de la arquitectura IoT, se procedió a desplegar la arquitectura en su totalidad en un ambiente de producción en el cual se hicieron pruebas de implantación del backend y de los demás componentes de la arquitectura IoT.

Actividades realizadas:

- Despliegue de la arquitectura en un ambiente de producción.
- Pruebas de implantación.

# 5. Requerimientos y arquitectura de referencia

En este proyecto se define una arquitectura de referencia para aplicaciones IoT para Smart Campus, esta arquitectura puede ser utilizada por los desarrolladores de aplicaciones para implementar su funcionalidad.

Por motivos de alcance del proyecto no se definieron requisitos relacionados con la seguridad en la manipulación de los datos.

## 5.1 Requerimientos funcionales

| | |
|---|---|
| **Identificación** | RF01 |
| **Nombre** | Servicio web para gestión de usuarios |
| **Descripción** | Implementación de un conjunto de servicios web tipo REST que permita la consulta, creación, actualización, autenticación, cambios y recuperación de contraseñas, y eliminación de usuarios de la plataforma administrativa. |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RF02 |
| **Nombre** | Servicio web para gestión de dispositivos |
| **Descripción** | Implementación de un conjunto de servicios web tipo REST que permita la consulta, creación, actualización, eliminación de dispositivos (sensores y actuadores) de la plataforma administrativa. |

| | |
|---|---|
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RF03 |
| **Nombre** | Servicio web para gestión de gateways |
| **Descripción** | Implementación de un conjunto de servicios web tipo REST que permita la consulta, creación, actualización, eliminación de dispositivos gateway de la plataforma administrativa. |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RF04 |
| **Nombre** | Servicio web para gestión de procesos |
| **Descripción** | Implementación de un conjunto de servicios web tipo REST que permita la consulta, creación, actualización, eliminación de procesos de gateway de la plataforma administrativa. |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RF05 |
| **Nombre** | Servicio web para gestión de aplicaciones |
| **Descripción** | Implementación de un conjunto de servicios web tipo REST que permita la consulta, creación, actualización, eliminación de aplicaciones de la plataforma administrativa. |

| | |
|---|---|
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RF06 |
| **Nombre** | Servicio web para la gestión de notificaciones |
| **Descripción** | Implementación de un conjunto de servicios web tipo REST que permita la consulta, confirmación de lectura, creación, actualización y eliminación de notificaciones de la plataforma administrativa. |
| **Prioridad** | Baja |

| | |
|---|---|
| **Identificación** | RF07 |
| **Nombre** | Servicio para el monitoreo de dispositivos |
| **Descripción** | Implementación de un conjunto de servicios REST que retornen y modifiquen el estado de los dispositivos y los gateways, y una tarea programada que consulte estos servicios y haga uso de los servicios de notificación para notificar al usuario en caso de un cambio de estado de un dispositivo (activo/inactivo). |
| **Prioridad** | Media |

| | |
|---|---|
| **Identificación** | RF08 |

| | |
|---|---|
| **Nombre** | Cliente para intercepción de mensajes de los dispositivos Edge |
| **Descripción** | Implementación de un cliente consumidor AMQP que se subscriba a un conjunto de colas para recibir los mensajes publicados por los dispositivos Edge, valide la configuración de cada mensaje y lo guarde en la base de datos para su posterior consulta. |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RF09 |
| **Nombre** | Cliente para el reenvío de mensajes de los dispositivos Edge |
| **Descripción** | Implementación de un cliente productor AMQP que, luego de haber interceptado y validado los mensajes de los dispositivos Edge, los publique en los tópicos destinados para consumo de las aplicaciones de usuario. |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RF10 |
| **Nombre** | Servicio para consulta de históricos de datos |
| **Descripción** | Implementación de un servicio web tipo REST que permita la consulta de los mensajes recibidos por el backend en un determinado periodo de tiempo. |
| **Prioridad** | Media |

| | |
|---|---|
| **Identificación** | RF11 |
| **Nombre** | Servicio para la difusión de instrucciones a los actuadores |
| **Descripción** | Implementación de un servicio web REST que difunda un mensaje a los dispositivos Gateway con datos de instrucciones para los actuadores. |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RF12 |
| **Nombre** | Servicio para consulta de estadísticas del sistema |
| **Descripción** | Implementación de un conjunto de servicios web tipo REST que permita la consulta de algunas estadísticas sobre el estado de los dispositivos y los mensajes enviados por gateway y por proceso. |
| **Prioridad** | Alta |

### 5.2 Requerimientos no funcionales

| | |
|---|---|
| **Identificación** | RNF01 |
| **Nombre** | Manejo de errores |
| **Descripción** | Implementación, en todos los métodos donde sea necesario, de una lógica para el manejo de los potenciales errores y excepciones. |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RNF02 |
| **Nombre** | Manejo de conexiones de múltiples instancias |
| **Descripción** | Implementación de un framework que gestione un pool de conexiones para soportar la concurrencia de operaciones en las diferentes instancias de los microservicios. |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RNF03 |
| **Nombre** | Arquitectura escalable |
| **Descripción** | Diseñar la arquitectura de manera que los servicios sean modulares e independientes, para así poder hacer uso de los beneficios en términos de escalabilidad que ofrece el mecanismo de despliegue del proyecto "Definición de una plataforma cloud de alta disponibilidad en un entorno distribuido para el despliegue de una plataforma IoT (Rojas, J. 2019)". |
| **Prioridad** | Alta |

| | |
|---|---|
| **Identificación** | RNF04 |
| **Nombre** | Arquitectura extensible |
| **Descripción** | Diseñar la arquitectura de manera que la implementación de nuevas tecnologías en la plataforma IoT se pueda hacer de forma simple. |

| | |
|---|---|
| **Prioridad** | Alta |

### 5.3 Arquitectura de referencia

A continuación, se muestra la arquitectura de referencia, creada con base en las arquitecturas estudiadas durante la etapa de Capacitación tecnológica y representa la plataforma Smart Campus desde el punto de vista del usuario, es decir, es una guía para los usuarios que deseen implementar una aplicación utilizando la plataforma desarrollada.

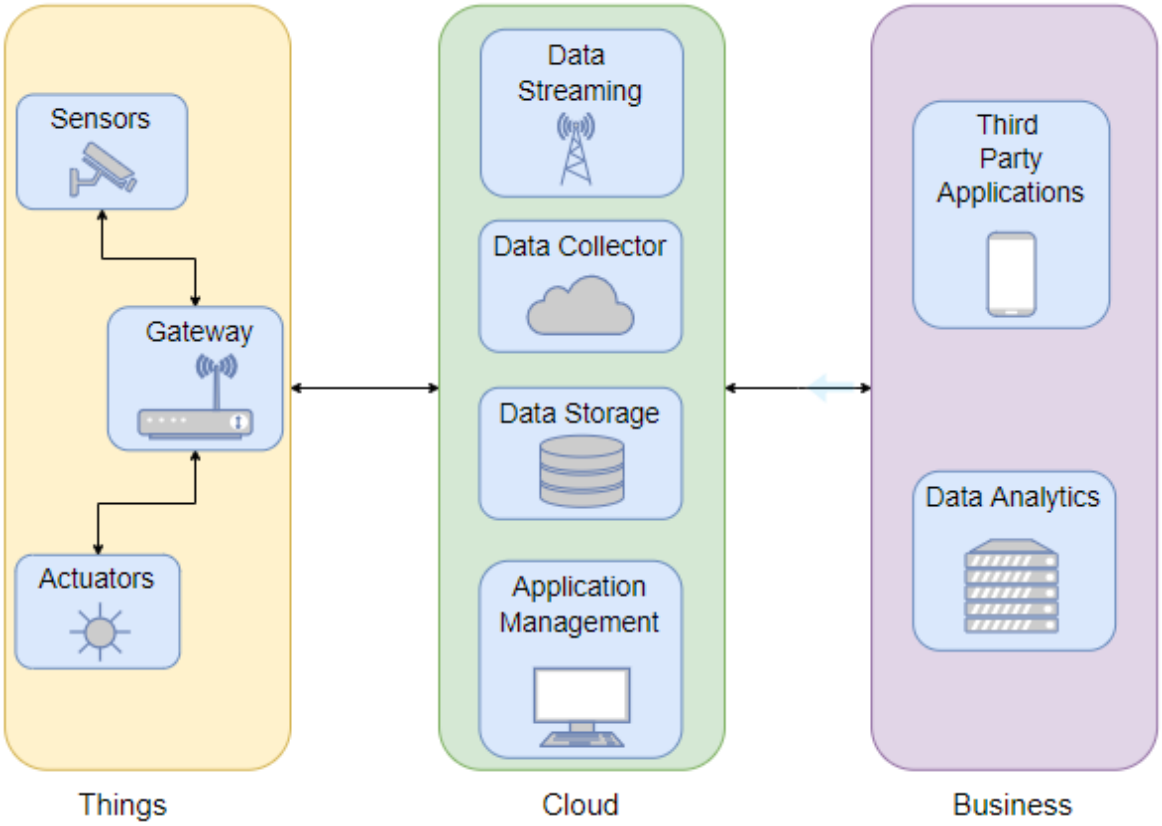


*Figura 10.* Arquitectura de referencia

Esta se compone por tres capas, la capa Things que representa la capa sensorial, y está compuesta por sensores, actuadores y gateways; la segunda es la capa Cloud donde se encuentran las aplicaciones de transmisión de datos, almacenamiento de información y gestión de aplicaciones; finalmente la capa Business donde se encuentran las aplicaciones de terceros o externas y las aplicaciones de analítica de datos desarrolladas por los usuarios.

Para el desarrollo de una aplicación o caso de uso, los usuarios deben construir software para extraer información de sensores y recibirla en actuadores; esta información es enviada utilizando el software Gateway que se provee en la plataforma Smart Campus, donde se procesa y envía la información al software Backend o Cloud, el cual a su vez la expone para que otras aplicaciones de usuarios transformen o procesen la información recibida y, finalmente la muestren y den uso mediante aplicaciones externas o sea enviada a aplicaciones de analítica de datos.

# 6. Desarrollo del proyecto

## 6.1 Contexto del proyecto

El objetivo de este proyecto fue crear un software extensible y escalable de tipo Backend para una plataforma IoT orientada a Smart Campus, cabe resaltar que este trabajo se encuentra en el marco de un macroproyecto que se compone también de otros tres trabajos de grado que se desarrollaron paralelamente a este, estos otros tres trabajos son elementos cruciales también en la plataforma, el primero, es un framework software extensible para dispositivos tipo Gateway que se encarga de gestionar a nivel de Edge Computing los datos de los dispositivos y enviarlos a un broker de donde

el backend los toma, el segundo, es la plataforma web de administración de los elementos de la plataforma IoT, esta se encarga de darle a los usuarios las herramientas necesarias para administrar las entidades y los elementos de la plataforma y el tercero es el encargado del despliegue de la misma.

Todos los elementos creados, componen una plataforma IoT para Smart Campus, con esto se sientan las bases tecnológicas para poder crear aplicaciones IoT que favorezcan a la comunidad, como por ejemplo el riego de las plantas, el control automatizado de acceso a las aulas y los edificios, smart parking y otros.

## 6.2 Capacitación tecnológica

**6.2.1. Arquitecturas IoT.** Para tener una buena referencia y guía para iniciar el desarrollo del proyecto, se hizo un leve estudio de algunas de las plataformas IoT más famosas del mercado, Azure IoT y AWS IoT.

Las dos ofrecen características esenciales en este tipo de plataformas como escalabilidad, disponibilidad y solidez en sus soluciones, dado que han venido trabajando en estos proyectos mucho tiempo, cuentan con una serie de opciones dependiendo del tipo de usuario que vaya a implementar la solución.

Las arquitecturas (mostradas a continuación) que plantean, sirvieron como guía para el desarrollo del presente proyecto y los que lo acompañaron en su desarrollo.

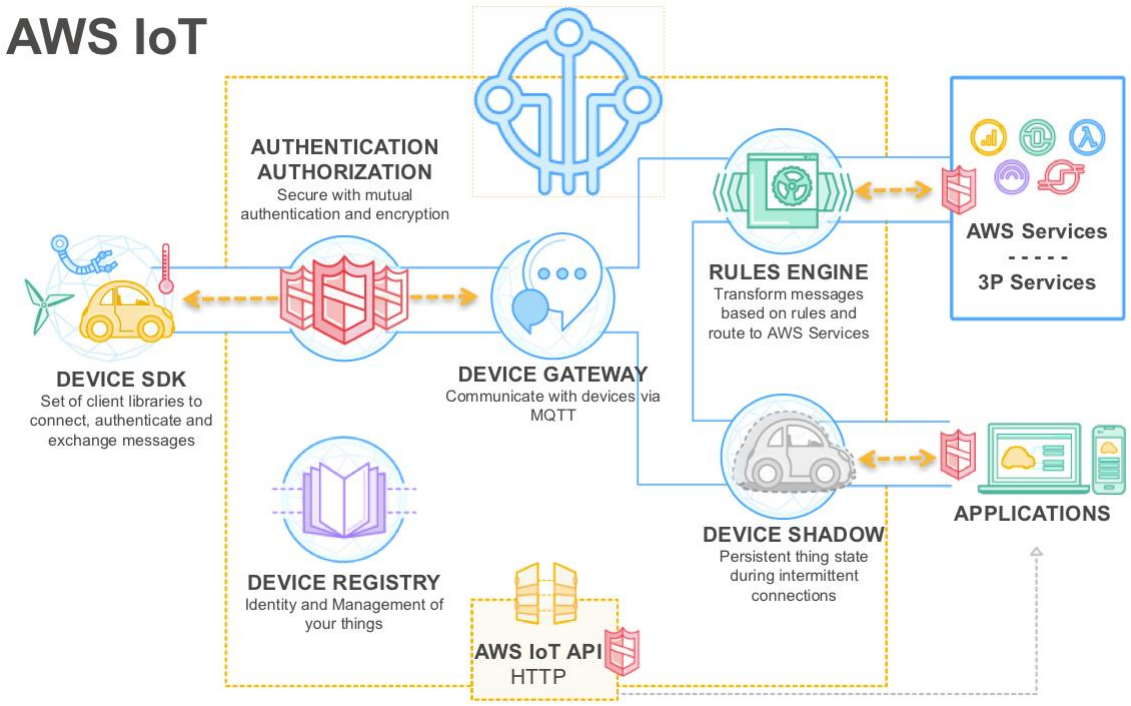


*Figura 11.* Arquitectura de AWS IoT. Tomada de McCauley, R. (2016).

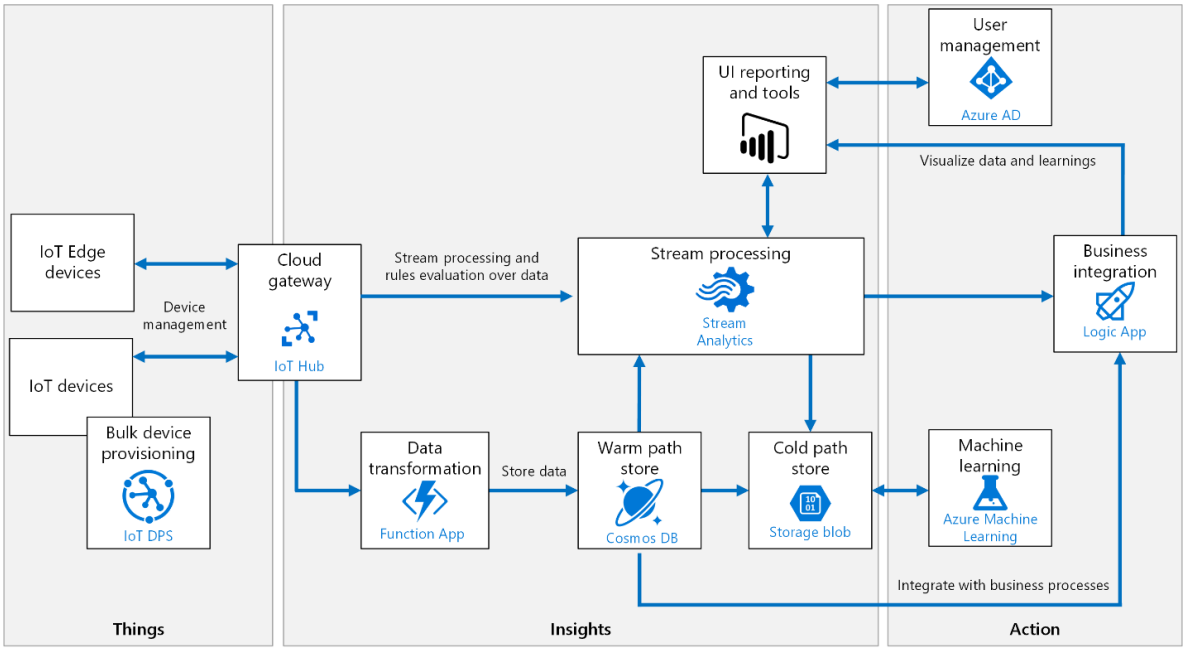


*Figura 12.* Arquitectura de Azure IoT. Tomada de Microsoft Azure (2019).

Aunque claramente se observan algunos componentes distintos, se identifican unas capas en común, las cuales son:

- Dispositivos: En esta capa es donde se encuentran los sensores y actuadores, que son quienes proporcionan los datos y las acciones que se almacenan y efectúan en la plataforma.
- Gateway: Esta es la capa encargada del Edge computing, permite realizar una conexión entre el backend o cloud y los dispositivos para hacer un pre procesamiento de los datos y toma decisiones críticas cuando es necesario.
- Cloud: Es el soporte lógico de la plataforma, contiene la lógica más compleja y almacena todos los datos de la misma, presta también un conjunto de APIs que permiten a aplicaciones externas integrarse con la plataforma y así extender su funcionalidad de una manera sencilla.
- Administración: Dado que este tipo de plataformas son complejas, es necesario tener un punto de acceso centralizado e intuitivo para que los usuarios puedan tener un panorama global de sus aplicaciones, para esto son diseñadas las plataformas de administración web y/o móvil.

**6.2.2. Tecnologías.** A continuación, se mencionarán las tecnologías y herramientas usadas en el desarrollo del proyecto, desde el lenguaje hasta el framework usado.

- Lenguaje de programación:

El lenguaje usado para el desarrollo del proyecto fue JAVA, ya que es uno de los lenguajes más estables y es también robusto, pero además cuenta con ciertas características que lo hacen óptimo para una plataforma IoT:

- Es orientado a objetos, lo cual lo hace útil ya que permite tener una representación lógica bien estructurada de los elementos de la arquitectura.
- Es muy tradicional y uno de los más usados en el mundo, por lo cual tiene mucha documentación.
- Es muy portable y no posee limitaciones de hardware.

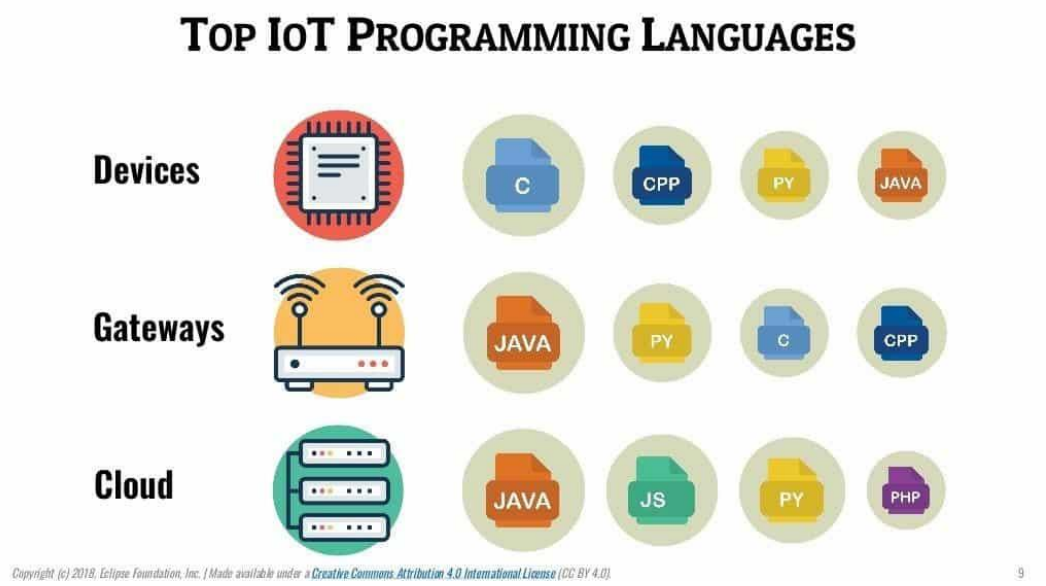


*Figura 13.* Lenguajes de programación preferidos para IoT. Tomada de iot for all. (2018).

- Protocolos de comunicación:

HTTP: Es un protocolo de la capa de aplicación para la transmisión de documentos hipermedia, como HTML. Fue diseñado para la comunicación entre los navegadores y servidores web, aunque puede ser utilizado para otros propósitos también. Sigue el clásico modelo cliente-servidor, en el que un cliente establece una conexión, realizando una petición a un servidor y espera una respuesta del mismo. Se trata de un protocolo sin estado, lo que significa que el servidor no guarda ningún dato (estado) entre dos peticiones. Aunque en la mayoría de casos se basa en una conexión del tipo TCP/IP, puede ser usado sobre cualquier capa de transporte segura o de confianza, es decir, sobre cualquier protocolo que no pierda mensajes silenciosamente, tal como UDP. (Mozilla, 2019).

AMQP: Es un protocolo usado en la comunicación cliente/servidor en el manejo de dispositivos IoT. Es eficiente, portable, multicanal y seguro. Es un protocolo binario y ofrece autenticación y encriptación vía SASL o TLS, se basa en un protocolo de la capa de transporte como lo es TCP. (Rouse, 2018). Este protocolo, además puede ser usado para publicar mensajes a una cola, lo cual, debido a la arquitectura de microservicios usada en el proyecto, es una manera fácil de controlar la integridad de los datos del sistema.

- Bases de datos:

MySQL: Es una de las bases de datos más usadas, tiene un buen performance y además cuenta con una herramienta visual de diseño (MySQL Workbench) que resulta siendo muy útil.

MongoDB: Es un motor de bases de datos NoSQL y es uno de los más usados por su ligereza y su rapidez al momento de buscar información.

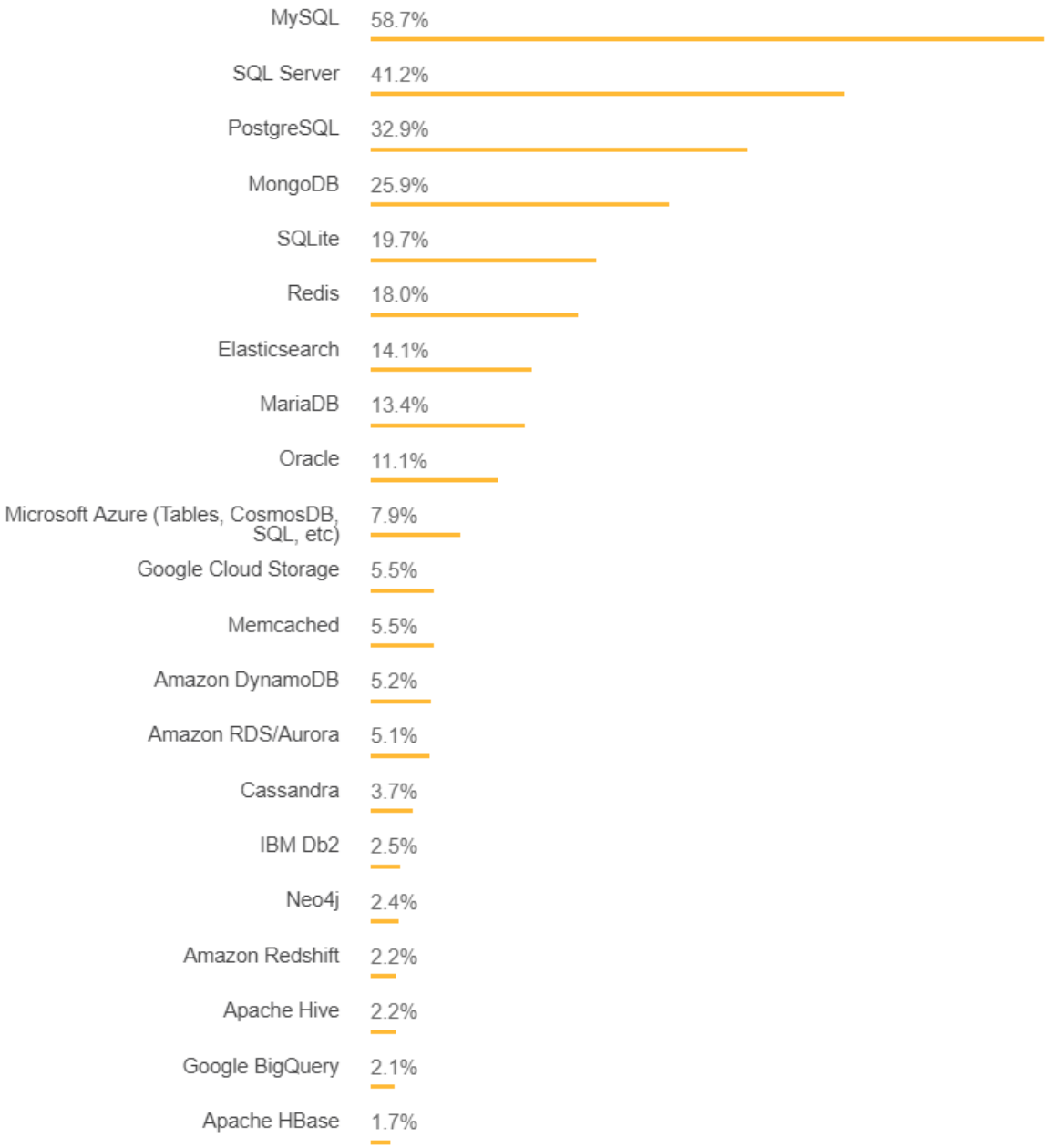


*Figura 14.* Most used databases in 2018. Tomada StackOverflow. (2018).

- Framework:

Spring Boot: Se eligió este framework que es uno de los más usados en Java, posee una gran documentación y soporte, está en constante actualización, es de fácil uso y posee muchas funcionalidades útiles para el desarrollo de microservicios.

### 6.3 Definición de la arquitectura

Como se mencionó anteriormente, este proyecto hace parte de un macroproyecto que se compone del presente y tres proyectos más, desarrollados paralelamente, todos contribuyeron de una manera esencial a los elementos necesarios para la plataforma IoT para Smart Campus planteada en los objetivos.

Es importante entonces, dar a conocer la arquitectura global para luego explicar dentro del contexto la del presente proyecto.

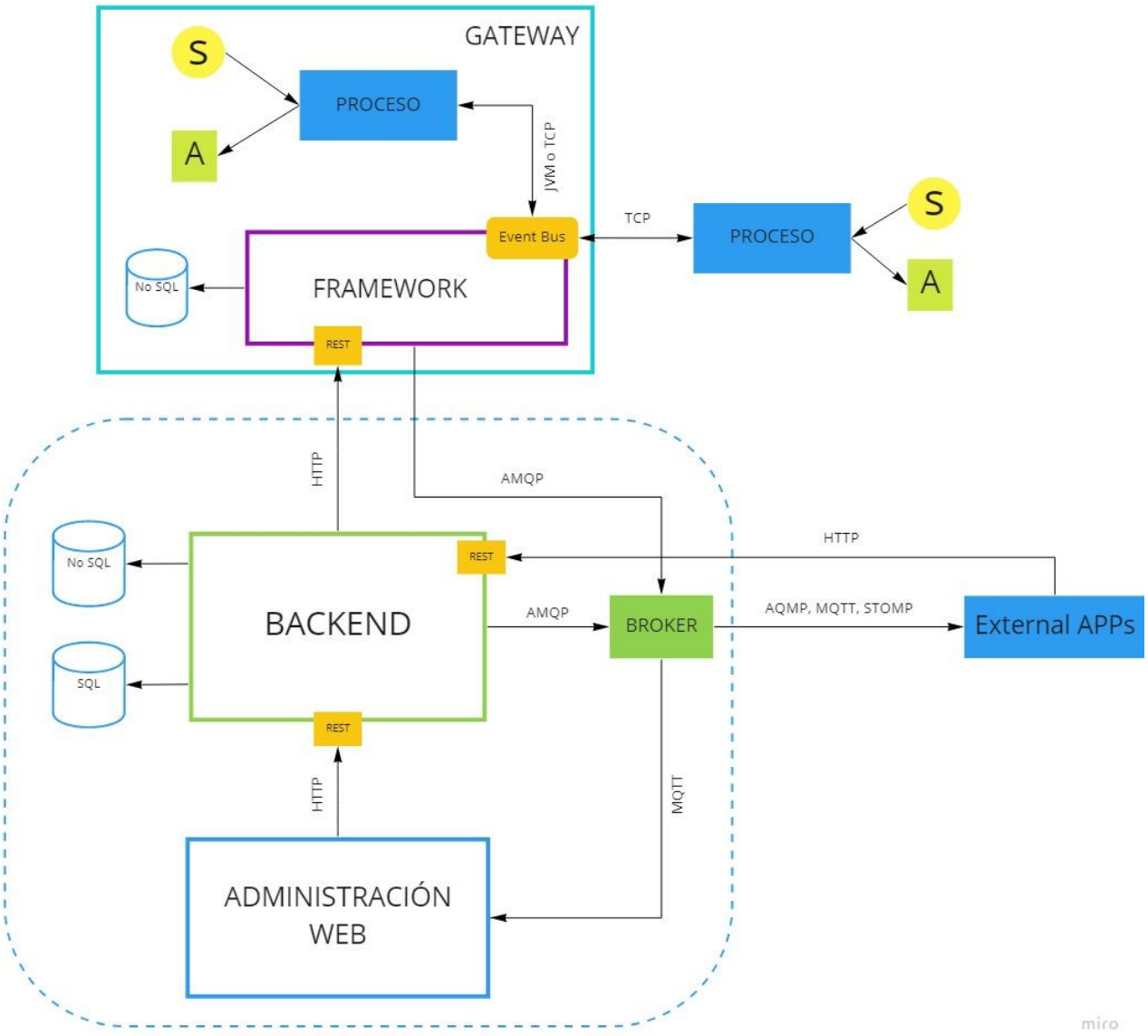


*Figura 15.* Arquitectura IoT para Smart Campus

Los trabajos que corresponden a los otros elementos de la arquitectura son:

- Gateway: En este proyecto de grado se presenta el diseño de un framework de software extensible que permite a dispositivos tipo gateway la comunicación y el almacenamiento de datos producidos y recibidos por sensores y/o actuadores al mismo

tiempo que provee la capacidad de conectarse con plataformas IoT enfocadas en Smart Campus. Esto permite que los diferentes casos de uso de IoT sean implementados con mayor rapidez enfocándose en el manejo de los dispositivos finales (sensor y actuador) y delegando al framework lo referente al exterior y funciones generales (Gutiérrez, 2019).

- Administración Web: En este proyecto de grado se presenta el diseño de una aplicación web que permite gestionar una arquitectura Smart Campus mediante la cual los usuarios pueden registrar y gestionar sus casos de uso y los dispositivos asociados a los mismos permitiéndoles de una manera sencilla crear aplicaciones que contribuyan a generar esta transformación digital (Camacho, 2019).

- Despliegue: En este trabajo de investigación se presenta el diseño de una infraestructura software para el despliegue de una plataforma IoT en una infraestructura cloud de alta disponibilidad en un entorno distribuido con el fin de proveer un entorno que pueda soportar cantidades masivas de solicitudes permitiendo una escalabilidad horizontal en la infraestructura hardware. (Rojas, 2019).

Luego de definir la arquitectura general, se hizo la definición de la arquitectura para este proyecto, la del backend, que se explicará a detalle a continuación:

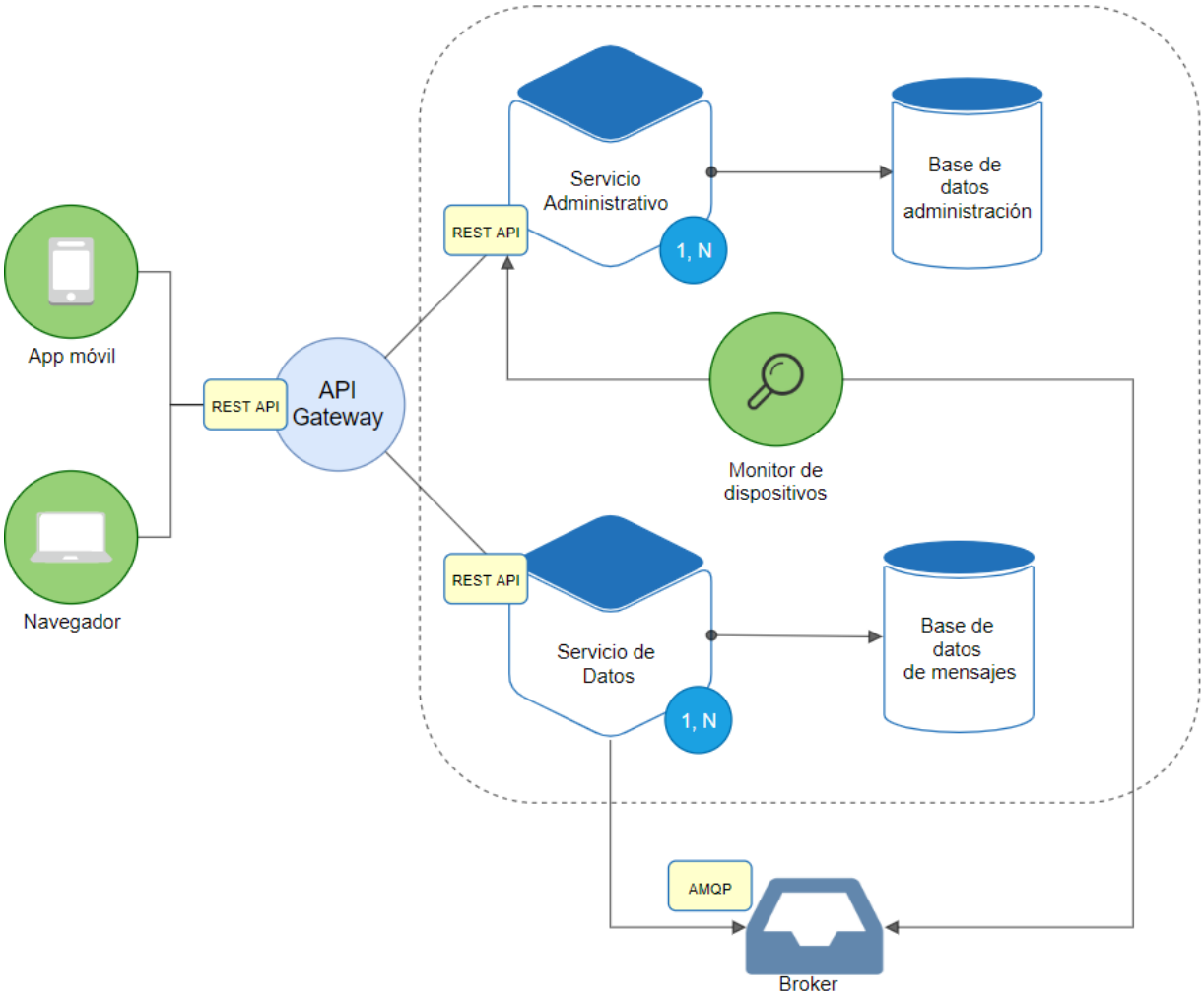


*Figura 16.* Arquitectura Backend

**6.3.1. API Gateway.** El API Gateway actúa como un punto de acceso a los servicios del backend. Sus funciones principales son ofrecer un endpoint base único para todos los servicios REST, y encargarse del balance de carga, o en otras palabras la distribución de la carga de trabajo (solicitudes) entre las diferentes instancias de los servicios de acuerdo con su disponibilidad.

Cabe mencionar que el desarrollo de este componente de la arquitectura no pertenece a este proyecto, sino al proyecto “Definición de una infraestructura cloud de alta disponibilidad en un entorno distribuido para el despliegue de una plataforma IoT (Rojas, J. 2019)”.

**6.3.2. Servicio Administrativo.** El componente backend de la plataforma se conecta con diferentes aplicaciones de usuario que deseen hacer uso de la plataforma. Sin embargo, existe una aplicación particularmente importante, encargada de la gestión de la plataforma. El servicio administrativo provee los métodos que esta aplicación consume.

Hay 5 entidades principales en el servicio administrativo:

Usuarios: Representa los usuarios de la plataforma, este usuario le permite crear el modelo lógico de sus aplicaciones en la plataforma. Cuenta con una serie de servicios que permite la administración de los usuarios.

Aplicaciones: Representa las aplicaciones que tiene el usuario en el sistema, tales como lo podría ser una aplicación de Smart Parking. Cuenta con una serie de servicios que permite la administración de las aplicaciones.

Gateways: Es la representación lógica de los gateways físicos que se encuentran en la plataforma. Cuenta con una serie de servicios que permite la administración de los gateways a nivel lógico y también le comunica los cambios a su respectivo gateway físico.

Procesos: Los procesos son la representación de las aplicaciones a nivel de Gateway, son aplicaciones instaladas en los gateways físicos que contienen la lógica de los sensores y actuadores, estos se detallan más a fondo en el trabajo de grado “Diseño de un framework software

extensible para dispositivos tipo gateway integrados en plataformas IoT para Smart Campus (Gutiérrez, 2019).”. Cuenta con una serie de servicios que permite la administración de los procesos a nivel cloud y también le comunica los cambios a su respectivo gateway físico donde se encuentren.

Dispositivos: Es la representación lógica de los dispositivos físicos que se encuentran conectados a los gateways de la plataforma. Cuenta con una serie de servicios que permite la administración de los dispositivos a nivel lógico y también le comunica los cambios a su respectivo gateway físico donde se encuentren.

Adicionalmente, se encuentran también:

Notificaciones: encargada del envío de alertas al usuario de la aplicación administrativa.

Estadísticas: encargada del envío de información acerca del estado de los dispositivos y gateways.

**6.3.3. Servicio de Datos.** La plataforma tiene conectados varios dispositivos gateway, que a su vez tienen conectados varios sensores y actuadores. Los sensores envían datos los gateways, los cuales se encargan de hacer un preprocesamiento de estos datos y publicarlos a una cola de mensajes en un broker. Aquí es donde inicia la participación del servicio de datos en la arquitectura, pues este ofrece un conjunto de utilidades para que el backend pueda acceder a los mensajes del broker, persistirlos en su respectiva base de datos, y reenviarlos para que los usuarios tengan acceso a ellos.

De igual manera, el servicio de datos permite generar solicitudes REST a los dispositivos gateway, para el envío de instrucciones a los actuadores conectados a estos, además de permitir la consulta de historiales de datos por fecha, usuario, gateway, proceso, tópico (se explicará más adelante su significado) y generar algunas estadísticas respecto a estos datos (cantidad de mensajes por proceso y gateway de un usuario determinado).

**6.3.4. Base de datos de administración.** Es la base de datos que contiene la información referente a la infraestructura física, es su representación lógica y relaciones entre sí. Se usó el motor de bases de datos MySQL que es uno de los más usados.

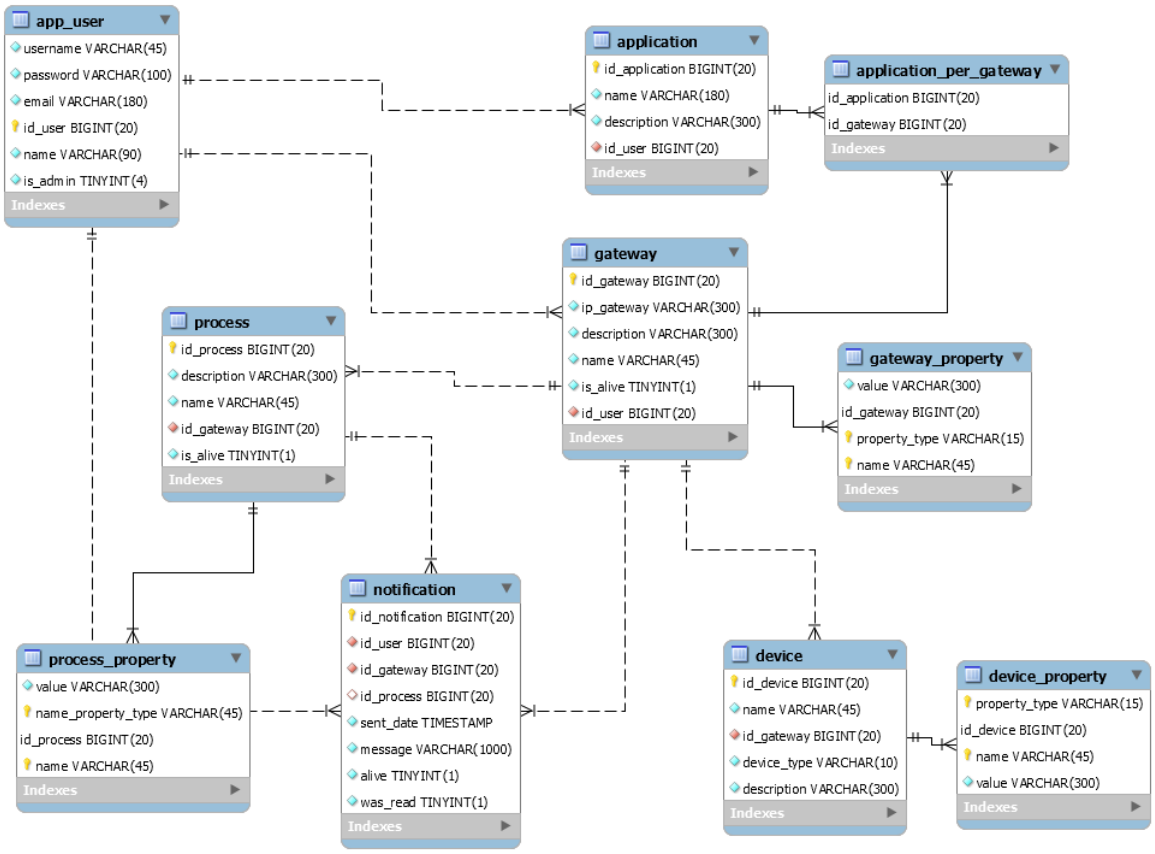


*Figura 17.* Modelo de la base de datos

Aquí se encuentran las entidades presentes en la plataforma como los gateways y los dispositivos, también están presentes entidades lógicas como los procesos (son los casos de uso a nivel del gateway), y algunas adicionales útiles para el usuario administrador como las notificaciones donde se guarda el registro de los cambios de estado de los elementos físicos de la plataforma.

Es necesario clarificar algunos puntos acerca del modelo de la plataforma:

- Un usuario puede tener 0 o muchas aplicaciones.
- Una aplicación puede tener 0 o muchos gateways asociados, pero solo puede estar asociada a un usuario.
- Un gateway puede estar relacionado con una aplicación, pero puede no estarlo y puede tener 0 o muchos procesos.
- Un proceso pertenece a un solo gateway y tienen una propiedad muy importante llamada tópico que se hace obligatoria en el registro del mismo en la plataforma de administración web (uno de los proyectos de los que se hizo mención anteriormente), este tópico le permite al usuario tener un canal de comunicación tanto para recibir los datos de los procesos como para enviar un mensaje a los que tengan el mismo tópico asociado.
- Un dispositivo pertenece a un solo gateway.

**6.3.5. Base de datos de mensajes**. Es la base de datos donde se almacenan todos los mensajes generados por los dispositivos presentes en la plataforma, dado que estos mensajes se

almacenan en grandes volúmenes y con data no estructurada, la base de datos elegida fue MongoDB, que es una de las más populares dentro de las No SQL.

**6.3.6. Monitor de dispositivos.** Dado que la plataforma IoT está diseñada para el campus, es probable que en algún punto hayan tantos dispositivos que su monitoreo sea difícil de hacer, para ello se creó este plug-in, se encarga de cada cierto tiempo, enviar un mensaje de difusión a los gateways del sistema para que estos le respondan con el estado de los procesos que se estén ejecutando en ese momento, así los usuarios pueden tener la información actualizada del estado de la plataforma de sus aplicaciones sin tener que ir al sitio físico a revisarlo.

**6.3.7. Broker.** Es altamente recomendable sino obligatorio manejar protocolos de bajo consumo para la comunicación entre los dispositivos Edge (Gateways) de una plataforma IoT con el cloud (Backend), por ello se usó un Broker cuya función es recibir los mensajes provenientes de los dispositivos y tenerlos disponibles para que el Backend los tome, los almacene y los ponga a disposición de los usuarios que hacen uso de la plataforma. También se usó para enviar las notificaciones del estado de los dispositivos para que puedan ser consultados desde aplicaciones externas.

### 6.3.8. Arquitectura interna de los servicios web.

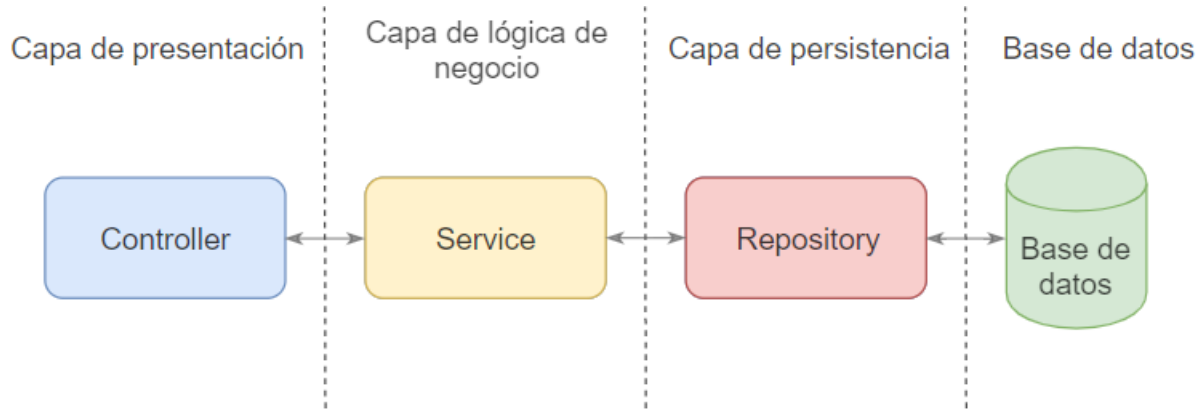


*Figura 18.* Modelo de arquitectura por capas

Para la construcción de los servicios web que contienen las funcionalidades específicas dentro de cada microservicio, se usó un modelo por capas que permite hacer la separación de las funcionalidades y mantener el código y el modelo lógico ordenado.

La arquitectura de los servicios se diseñó como un modelo de tres capas que se explicarán a continuación junto con la capa de datos (base de datos):

- Controller o capa de presentación: Esta es la capa donde se expone el endpoint del servicio REST, hace la transformación respectiva de los datos, en este caso los pasa de entidad a DTO, la entidad es una representación fiel a la base de datos y los DTO (Data Transfer Object) son objetos planos que son una representación plana de las entidades que hacen fácil la comunicación cliente-servidor, ya que son serializables.

- Service o capa de lógica de negocio: Aquí se efectúa toda la lógica de negocio, se hace la abstracción de lógica interna y se centraliza la información, esto con el fin de hacer una

división entre la capa de presentación y la de persistencia para que, si luego se necesitan hacer cambios, solo se hagan acá y no afecte la aplicación trasversalmente.

- Repository o capa de persistencia: Esta es la capa que accede los datos directamente, se apoya en una serie de clases llamadas Entidades, que son representaciones en objetos de las tablas de la base de datos para su fácil acceso, gracias a esto y al framework usado (Spring Boot), la comunicación es transparente con la base de datos y cada instancia de una entidad, representa un registro en la base de datos (en ocasiones también sus relaciones).
- Base de datos: Aquí se almacena toda la información de la plataforma y algunas representaciones lógicas útiles para la plataforma IoT, se explicó más a detalle en la sección 6.3.4.

## 6.4 Prototipado

Con respecto a la implementación del prototipo, la determinación que se tomó fue abordarla tomando como lineamiento los tres conjuntos de funcionalidades más principales, y desarrollarlos, en principio, independientemente, y luego de tener una base de código fuerte proceder a realizar la integración en donde fuese necesario.

**6.4.1. Servicio administrativo.** Como se mencionó anteriormente, el framework escogido para realizar el prototipo fue Spring Boot. Este framework facilita la configuración de Spring MVC, el cual ofrece diversos patrones de diseño, de entre los cuales se escogió para utilizar el patrón de arquitectura por capas Controller – Service – Repository. A continuación se hará una descripción de cada uno de estos:

***6.4.1.1. Controller.*** *El* controller *es el componente de la capa de presentación. Su principal funcionalidad es servir como punto de acceso al API REST para los usuarios, pues este componente ofrece los endpoints a los cuales los usuarios deben apuntar para hacer uso de los servicios.*

A continuación, podemos observar algunos snippets de un controller simple, con base en los cuales se pueden hacer algunas observaciones:

```
@RestController
```

En Spring Boot, los controllers para servicios REST deben llevar una anotación @RestController, la cual marca la clase como un componente de tipo controller, e implícitamente marca a todos sus métodos como @ResponseBody, lo que quiere decir que todos sus métodos van a retornar una respuesta HTTP. El marcar una clase como component implica que esta clase es un Bean de Spring y será tratado como un singleton.

```
@RequestMapping("/users")
```

Los controllers, además, deben llevar una anotación @RequestMapping que indica bajo qué dominio se van a exponer los servicios de ese controller. En este caso, el dominio es "/users", lo que quiere decir que para llamar a cualquiera de los métodos del UserController, el endpoint debe comenzar por la dirección IP/nombre de dominio del servidor, seguido de la cadena "/users", seguido de la ruta relativa del servicio.

```
@Autowired
private IUserService service;
```

Cada controller tiene, cuando menos, un service asociado. Este service debe estar marcado con la anotación @Autowired para permitir que realice inyección de dependencias al controlador. Esto quiere decir que el service estará disponible en el controller sin necesidad de ser instanciado, y sin necesidad de llamarse usando getters o setters, sino que será Spring Boot quien gestione la instancia del service.

```
@PostMapping (path = "/user", consumes = "application/json", produces =
        "application/json")

  public ResponseEntity<AppUserDTO> register (@RequestBody final AppUserDTO userDTO)
```

Cada uno de los métodos del controller debe ir anotado con una de las anotaciones de los verbos HTTP (@PostMapping, @GetMapping, @PutMapping, @DeleteMapping) y dentro de esta anotación algunos parámetros según corresponda. Como mínimo, toda anotación de verbo HTTP deben tener un atributo “path” que representa la ruta relativa desde la cual se va a llegar a un determinado servicio, pero para los casos de PUT y POST, estos deben tener además unos atributos “consumes” y “produces” que indican el formato de los datos que el servicio va a recibir en la solicitud, y el formato que va a tener la respuesta, respectivamente.

Adicional a esto, cada uno de los parámetros de los métodos del controller deben ir anotados con una de las siguientes anotaciones:

- @RequestBody: Indica que el parámetro va a venir en la forma de un objeto en el cuerpo de la solicitud.
- @PathVariable: Indica que el parámetro va a venir en la URL, como parte del endpoint (e.g. /usuarios/usuario/1). Este tipo de parámetros se usan generalmente para indicar el valor de un campo de una entidad específica.

- @RequestParam: Indica que el parámetro va a ser del tipo queryparam (e.g. /usuarios?tamaño=10). Este tipo de parámetros se usan, generalmente, para ordenar/filtrar la lista de resultados del servicio.

```
Assert.hasText(userDTO.getUsername(), USERNAME_REQUIRED);
Assert.hasText(userDTO.getPassword(), PASSWORD_REQUIRED);
Assert.hasText(userDTO.getEmail(), EMAIL_REQUIRED);
Assert.hasText(userDTO.getName(), NAME_REQUIRED);
```

El controller se encarga también de verificar que la solicitud venga correctamente configurada por el usuario. En caso de que no se cumpla alguna de las condiciones definidas en los métodos Assert, el controller retorna una excepción inmediatamente. Se hablará más del manejo de excepciones en la sección 6.4.1.5.

```
return new ResponseEntity<>(
        service.registerUser(AppUser.fromDTO(userDTO)).toDTO(),
        HttpStatus.CREATED);
```

Otra observación que se puede hacer es que, además del acceso al API de servicios, el controller se encarga de convertir las entidades que retornan los métodos del service a entidades planas llamadas DTOs, cuya función es retornar información al usuario de la forma más comprensible posible, sin respetar necesariamente la estructura o restricciones del modelo de datos.

***6.4.1.2. Service.*** El service es el componente de la capa de lógica de negocio. Funciona como un middleware entre el controller y el repository, e implementa toda la lógica de negocio, las validaciones de los inputs de usuario, y el manejo de excepciones.

A continuación, podemos observar algunos snippets de un service simple, con base en los cuales se pueden hacer algunas observaciones:

```
@Service
```

En Spring Boot, los services deben llevar una anotación @Service, la cual marca la clase como un componente de tipo service.

```
private static final Logger LOGGER = LoggerFactory.getLogger(GatewayService.class);
```

El service está encargado de llevar un log de la ejecución de la aplicación. En este log pueden ir anotaciones de diferente naturaleza, como mensajes informativos, advertencias, y mensajes de error en caso de que se presente alguno.

```
@Autowired
private IGatewayRepository repository;
@Autowired
private IUserRepository userRepository;
```

Cada service tiene, cuando menos, un repository asociado. Este repository debe estar marcado con la anotación @Autowired para permitir que realice inyección de dependencias al controlador. Esto quiere decir que el repository estará disponible en el service sin necesidad de ser instanciado, y sin necesidad de llamarse usando getters o setters, sino que será Spring Boot quien gestione las instancias.

```
@Transactional
```

Un método de un service puede estar marcado con la anotación @Transactional, lo que quiere decir que el método va a ser ejecutado dentro de una transacción. En otras palabras, antes de que un método con la anotación @Transactional se ejecute, Spring Boot abre una transacción para todos los potenciales cambios a la base de datos. Una vez concluída la ejecución del método, si no hubo ninguna excepción, Spring Boot hace commit a la transacción, y en el caso contrario, hace rollback para revertir todos los cambios. La razón por la que se utiliza esta anotación es porque en métodos donde se ejecuta un código que puede arrojar errores también puede ocurrir que previamente se haya alterado la base de datos, y en caso de ocurrir un error, la base quedaría alterada, y el sistema podría desincronizarse.

Es en el service en donde se hace el manejo de errores más fuerte. En caso de que ocurra algún error en la comunicación con la base o con algún servicio externo, o en caso de que ocurra una excepción en el código mismo, los métodos del service se encargan de intentar ejecutar un código de respaldo para normalizar la ejecución o, en caso de no ser esto posible, guardar la excepción en el log.

```
            // Subscribe to the new queue (gatewayId)
            CommonUtils.getHTTPResponse(ERoute.DATA + "/data/subscribe/" +
            savedGateway.getId(), HttpMethod.GET, String.class);
```

Como se mencionó previamente, el service también se encarga de la comunicación con aplicaciones externas, por ejemplo en el snippet anterior, el service hace una solicitud al servicio de datos (sección 6.4.2) para indicarle que debe suscribirse a una cola para recibir los datos del gateway registrado.

***6.4.1.3. Repository.*** El repository es el componente de la capa de persistencia, es decir, es el encargado de la comunicación con la base de datos para obtener y almacenar registros. Cada Repository realiza operaciones sobre una entidad, que a su vez se corresponde con una tabla o relación de la base de datos. Se puede encontrar una explicación más detallada sobre las Entidades en la sección 6.4.1.4.

A continuación, podemos observar algunos snippets de un repository simple, con base en los cuales se pueden hacer algunas observaciones:

```
@Repository
```

Todo repository debe ir marcado con la anotación @Repository. Esta anotación, además de marcar la clase como un componente de Spring, tiene como función capturar las excepciones de persistencia y lanzar excepciones de Spring en su lugar.

```
extends CrudRepository<Gateway, Long>
```

Cada repository está asociado a una entidad de la base de datos. Para esto, es crucial que el repository herede de la clase CrudRepository o una de sus subclases. Esta es una clase especial que facilita Spring Boot, la cual provee métodos para manejar la comunicación con la base de datos tales como save, findById, delete, count.

```
public List<Gateway> findByApplicationsId(long applicationId);
```

CrudRepository ofrece además la posibilidad de definir métodos para hacer consultas basadas en los campos de su entidad asociada. En el snippet anterior, por ejemplo, el método findByApplicationsId permite consultar los gateways que tengan una asociación en la base de

datos con una aplicación determinada. El campo ApplicationsId es un campo definido en la entidad Application, entidad Gateway posee un objeto Application como atributo.

```
@Query(value = "SELECT id_process, value FROM process_property AS PP WHERE
id_process IN (SELECT id_process AS P FROM process WHERE id_gateway
    = :gatewayId) AND PP.name = 'topic'", nativeQuery = true)
public List<Object[]> getAssociatedTopics(@Param("gatewayId") long gatewayId);
```

A pesar de que los Repositories trabajan con entidades, puede ocurrir alguna situación en la que se desee hacer una query particular para algún proceso específico. Esto es posible haciendo uso de la anotación @Query, que permite además crear query con parámetros que se pasan a través del método. En el caso de tratarse de consultas, las queries particulares retornan un tipo genérico Object[], en lugar de una entidad.

```
@Modifying
```

En la mayoría de ocasiones, las queries se desean para realizar consultas específicas que requieran subconsultas o cláusulas join. Sin embargo, también se puede hacer uso de la anotación @Modifying realizar queries que modifiquen datos.

***6.4.1.4. Entity.*** Finalmente, las clases Entity son la representación en clases de la base de datos. Una entity puede representar una tabla, una relación o incluso una llave compuesta. Es a través de estas clases que los repositories envían y reciben información de la base de datos. Las Entities son en realidad clases especiales que ofrece la implementación de JPA que hace Spring.

A continuación, podemos observar algunos snippets de una entity simple, con base en los cuales se pueden hacer algunas observaciones:

```
@Entity
```

```
@Table(name = "app_user")
```

La anotación @Entity indica que la clase es una entidad JPA y corresponde a una tabla. En la mayoría de casos el nombre de la clase y de la tabla no son iguales, por lo tanto es necesario utilizar una anotación @Table para indicar el nombre de la tabla que la Entity va a representar.

```
@NamedQuery(name = "AppUser.findAll", query = "SELECT a FROM AppUser a")
```

Las entities pueden definir queries personalizadas con la anotación @NamedQuery, las cuales luego pueden ser utilizadas por los Repositories.

Cada campo de la tabla de la entidad se corresponde con un atributo de la clase, que debe tener un tipo compatible. Algunas de las anotaciones de JPA importantes son:

- @Id: Indica que el atributo corresponde a la llave primaria de la tabla que la clase representa.
- @GeneratedValue: Determina qué estrategia de generación se va a utilizar para los valores de la llave primaria. Algunos ejemplos de estategías de generación son tomar los valores de una secuencia, o generarlos automáticamente desde JPA.
- @Column: Al igual que con las tablas, puede pasar que los nombres de los campos de la tabla no correspondan con los atributos de la clase, en cuyo caso se utiliza esta anotación junto al atributo “name” para indicar el nombre del campo que el atributo va a representar. Además de esto, la anotación permite determinar si la columna en la base de datos debe tener un valor único, o si puede ser nulo.

- @OneToMany: La forma en la que JPA maneja las relaciones entre tablas fuertes es a través de listas de entidades como atributos de otras entidades. Para definir el tipo de relación existe esta anotación, entre otras como @ManyToOne o @ManyToMany.

```
public AppUserDTO toDTO() {
        return new AppUserDTO(this.id, this.username, this.email);
}

public static AppUser fromDTO(AppUserDTO dto) {
        return new AppUser(dto.getId(), dto.getEmail(), dto.getUsername());
}
```

Una determinación que tomó durante el desarrollo fue la de implementar, en cada Entity, métodos para convertirla a su(s) DTO(s) correspondientes(s) además de métodos para convertir estos DTOs de vuelta a la entidad, pues se determinó que así la estructura estaba mejor organizada y el código era más fácil de usar.

***6.4.1.5. Manejo de errores.*** Además del manejo de errores mencionado anteriormente, se implementó una clase ApiExceptionHandler para manejar las excepciones que llegaban hasta el nivel más alto del código sin ser capturadas. El comportamiento por defecto de un API REST cuando ocurre un error es retornar un stack trace incomprensible para el usuario, pero gracias al ApiExceptionHandler, el servicio de administración captura las excepciones y las devuelve en un formato mucho más fácil de comprender.

***6.4.1.6. Lista de servicios.*** A continuación, una lista de los servicios por entidad:

- Usuarios

Ruta base: /users

Tabla 1.
*Servicios de usuarios*

| **Ruta** | **Método HTTP** | **Descripción** |
|---|---|---|
| /user | POST | Registra un nuevo usuario en la base de datos. |
| /user/{idUsuario} | PUT | Modifica el usuario, previamente registrado en la base de datos. |
| /user/{idUsuario} | DELETE | Elimina el usuario de la base de datos. |
| /user/{idUsuario} | GET | Retorna el usuario. |
| /password/{idUsuario} | PUT | Actualiza la contraseña del usuario. |
| /authentication | POST | Verifica las credenciales de un usuario para permitirle el ingreso a la plataforma administrativa. |
| / | GET | Retorna todos los usuarios. |
| /pass/{correo} | GET | Verifica si existe un usuario con el correo asociado a su cuenta, y le envía un correo de recuperación de contraseña. |

- Aplicación

Ruta base: /applications

Tabla 2.
*Servicios de aplicaciones*

| **Ruta** | **Método HTTP** | **Descripción** |
|---|---|---|
| /application | POST | Registra una nueva aplicación en la base de datos. |
| /application /{idApp} | PUT | Modifica la aplicación, previamente registrada en la base de datos. |
| /application /{idApp} | DELETE | Elimina la aplicación de la base de datos. |
| /application /{idApp} | GET | Retorna la aplicación. |
| /user/{idUsuario} | GET | Retorna la lista de aplicaciones del usuario. |
| /application/ gateway/{assign} | PUT | Asigna o desasigna un gateway de una aplicación basado en el parámetro booleano de la solicitud. |

- Gateway

Ruta base: /gateways

Tabla 3.
*Servicios de gateways*

| **Ruta** | **Método HTTP** | **Descripción** |
|---|---|---|

| | | |
|---|---|---|
| /gateway | POST | Registra un nuevo gateway en la base de datos y envía una solicitud al gateway físico para registrarlo allí también. |
| /gateway /{idGtw} | PUT | Modifica el gateway, previamente registrado en la base de datos y envía una solicitud al gateway físico para modificarlo allí también. |
| /gateway /{idGtw} | DELETE | Elimina el gateway de la base de datos y envía una solicitud al gateway físico para eliminarlo allí tambíen. |
| /gateway /{idGtw} | GET | Retorna el gateway. |
| / | GET | Retorna todos los gateways |
| /application /{idApp} | GET | Retorna la lista de gateways de la aplicación. |
| /topic?topic={topic**}** | GET | Retorna la lista de gateways asociados al tópico. |
| /process/{idProcess**}** | GET | Retorna la lista de gateways que tienen el proceso. |
| /user/{idUsuario} | GET | Retorna la lista de gateways asociados a aplicaciones del usuario. |
| /ip/topic?topic={topic**}** | GET | Retorna una lista de Gateways, con su IP y sus procesos asociados, según el tópico al que pertenece. |

| | | |
|---|---|---|
| /ip/process | GET | Recibe una lista de procesos y retorna una lista de los Gateways asociados a esos procesos, cada uno con su IP y todos sus procesos asociados. |
| /gateway /{idGtw}/topic | GET | Recibe una lista de tópicos asociados al proceso. |
| /application/gateway/ assignment | PUT | Asigna un gateway a una aplicación. |
| /gateway /{idGtw}/keepAlive | GET | Envía una solicitud de estado al gateway físico y luego persiste y retorna la información de los cambios de estado recibida. |

- Proceso

Ruta base: /processes

Tabla 4.
*Servicios de procesos*

| **Ruta** | **Método HTTP** | **Descripción** |
|---|---|---|
| /process | POST | Registra un nuevo proceso en la base de datos y envía una solicitud al gateway físico para registrarlo allí también. |

| | | |
|---|---|---|
| /process/{idProceso} | PUT | Modifica el proceso, previamente registrado en la base de datos y envía una solicitud al gateway físico para modificarlo allí también. |
| /process/{idProceso} | DELETE | Elimina el proceso de la base de datos y envía una solicitud al gateway físico para eliminarlo allí también. |
| /process/{idProceso} | GET | Retorna el proceso. |
| /gateway/{idGtw} | GET | Retorna la lista de procesos del gateway. |
| /application /{idApp} | GET | Retorna la lista de procesos de la aplicación. |
| /user/{idUsuario} | GET | Retorna la lista de procesos del usuario. |
| /topic?topic={topic} | GET | Retorna la lista de procesos asociados al topico. |
| /process/ids?userId={uId}&applicationId={aId}&topic={t}&gatewayId={gId} | GET | Retorna la lista de IDs de los procesos que cumplan con los filtros establecidos por los query params. |
| /process/{idProceso/deploy/{deploy} | PUT | Replica la solicitud recibida al gateway, para gatillar el despliegue del proceso. El parámetro *deploy* es un booleano que si es true |

| | | |
|---|---|---|
| | | despliega/redespliega el proceso y si es falso lo desactiva. |

- Dispositivos

Ruta base: /devices

Tabla 5.
*Servicios de dispositivos*

| **Ruta** | **Método HTTP** | **Descripción** |
|---|---|---|
| /device | POST | Registra un nuevo dispositivo en la base de datos y envía una solicitud al gateway físico para registrarlo allí también. |
| /device/{idDispo} | PUT | Modifica el dispositivo, previamente registrado en la base de datos y envía una solicitud al gateway físico para editarlo allí también. |
| /device/{idDispo} | DELETE | Elimina el dispositivo de la base de datos y envía una solicitud al gateway físico para eliminarlo allí también. |
| /device/{idDispo} | GET | Retorna el device. |
| /gateway/{idGtw} | GET | Retorna la lista de dispositivos del gateway. |

| | | |
|---|---|---|
| /application /{idApp} | GET | Retorna la lista de dispositivos de la aplicación. |
| /user/{idUsuario} | GET | Retorna la lista de dispositivos del usuario. |

- Notificaciones

Ruta base: /notifications

Tabla 6.
*Servicios de notificaciones*

| **Ruta** | **Método HTTP** | **Descripción** |
|---|---|---|
| /notification | POST | Registra una nueva notificación en la base de datos. |
| /notification/{idNoti} | PUT | Modifica la notificación, previamente registrada en la base de datos. |
| /notification/{idNoti} | DELETE | Elimina la notificación de la base de datos. |
| /user/{idUsuario}/read | GET | Marca como leídas las notificaciones del usuario. |
| /user/{idUsuario} | GET | Retorna la lista de notificaciones del usuario. |
| /user/{idUsuario}/count | GET | Retorna el número de notificaciones del usuario. |

- Estadísticas

Ruta base: /statistics

Tabla 7.
*Servicios de estadísticas.*

| **Ruta** | **Método HTTP** | **Descripción** |
|---|---|---|
| /{idUsuario} | POST | Retorna todas las estadísticas del usuario. Estas estadísticas indican cuántos gateways y procesos activos/inactivos hay en las aplicaciones del usuario; cuantas aplicaciones tiene, cuantos dispositivos, y cuáles son los cambios de estado más recientes hechos por el servicio de monitoreo. |

**6.4.2. Servicio de datos.** Este microservicio nace de la necesidad de manejar grandes volúmenes de datos no estructurados, por eso, se apoya de una base de datos no relacional y un broker AMQP como se puede observar en la figura 15. Para la construcción de los servicios se usó el modelo de capas explicado en la arquitectura (figura 17).

Las funcionalidades de este servicio se pueden dividir en tres, la del manejo de datos en tiempo real, la de la consulta de los datos históricos y la de difusión de mensajes.

***6.4.2.1. Manejo de datos en tiempo real.*** Para esta funcionalidad, se crearon 2 clases: ActiveMQService.java y MsgListener.java.

- ActiveMQService: Esta clase es de la capa de negocio (explicado en la sección 6.4.1.2), aparte de contar con estas propiedades, tiene otras funcionalidades necesarias para el manejo de los datos. Aquí se efectúan dos procesos cruciales, uno que crea un

consumidor para una cola del broker y otro que usa ese consumidor y se subscribe a cada una de las colas, existe una cola para cada gateway físico donde este pone los mensajes que generen sus procesos.

Luego, se construyó un método para crear consumidores para suscribirse a cada una de las colas, este método se llama en el startup de la aplicación, primero hace una petición al servicio de administración para conocer todos los gateways físicos presentes en la aplicación y luego para cada uno crea un consumidor para subscribirse a la cola respectiva de ese gateway, este proceso se reintenta 3 veces con una espera de 10 segundos entre intento.

- MsgListener: En esta clase, que está anotada con `@Component`, lo que significa que se puede inyectar en otros lugares de la aplicación, también tiene la etiqueta `@Scope("prototype")` que permite que se creen múltiples instancias de esta clase, necesitamos una por cada una de los consumidores que creamos para que se subscriban a cada cola. La función de esta clase es recibir los mensajes de la cola, validar que el gateway exista en el sistema y si existe, guardar los mensajes y publicarlos bajo el tópico que la persona estableció para el respectivo proceso que generó el mismo.

***6.4.2.2. Consulta de datos históricos.*** Para cumplir con esta funcionalidad se crearon dos servicios, uno que recupera los datos históricos, recibe como parámetros fecha inicial, fecha final, id usuario, id aplicación, id gateway, tópico y id proceso, el otro consulta las estadísticas de un usuario determinado, trae la cantidad de mensajes que han generado sus gateway y procesos a través del tiempo. Estos servicios están expuestos en el API REST.

Es importante mencionar que este servicio se apoya de un repository y a diferencia del explicado en la sección 6.4.1.3, este hereda de MongoRepository, que ayuda a hacer consultas a hacer consultas a la base de datos No SQL, Mongo, de una manera sencilla.

***6.4.2.3. Difusión de mensajes***. Las anteriores funcionalidades explicadas se relacionan con los mensajes generados por los sensores (que son obtenidos por las aplicaciones instaladas en los gateways físicos llamados procesos), esta funcionalidad, es la que le permite al usuario, enviarles instrucciones a sus actuadores o más bien, a los procesos instalados en el gateway que controlan estos actuadores.

Existe un servicio REST que tiene como parámetros de entrada una lista de id de procesos(opcional), un tópico(opcional) y un mensaje, la lista de id de procesos y el tópico son opcionales pero debe ir al menos uno de ellos, estos son útiles para encontrar los destinatarios a los cuales va dirigido el mensaje y el mensaje es lo que sea que necesite la persona que llegue al proceso para ejecutar la acción necesaria en el actuador.

**6.4.3. Monitor de dispositivos.** Dado que los dispositivos físicos de la plataforma (sensores, actuadores y gateways) y el software que los controla (procesos, framework) pueden tener fallos y que pueden estar distribuidos en un área geográfica grande, es necesario tener un mecanismo para hacerle saber al usuario estos fallos para que pueda tomar las acciones necesarias para corregirlas, esta función es la que cubre el monitor de dispositivos, es una aplicación creada en el backend que se ejecuta cada 5 minutos y envía una solicitud a todos los gateways y estos responden con el estado de sus procesos y dispositivos.

Esta aplicación tiene disponible un pool de conexiones para que los procesos que se efectúan en él, puedan hacerse en paralelo y así maximizar el uso de los recursos y la eficacia.

Las tareas que realiza esta aplicación son:

- Enviar una petición a los gateways (keep alive) para saber el estado de sus dispositivos, si el gateways no está disponible quiere decir que tanto el como sus dispositivos están presentando algún inconveniente, con la respuesta de los gateways, se procede a almacenar en la base de datos esta información para tenerla disponible para los usuarios.
- Enviar las notificaciones y estadísticas actuales de la plataforma en tiempo real al usuario final, esto se logra al enviarlas broker para que la plataforma de administración web los tome y los presente al usuario.
- Adicionalmente, esta aplicación de monitoreo de dispositivos se usó para verificar en el inicio del backend que la url que se tiene registrada en el archivo de propiedades coincida con la que se tiene almacenada en base de datos, si no es así, entonces se actualiza en la base de datos y se envía la actualización a los gateways físicos.

### 6.4.4. Lista de servicios

Ruta base: /data

Tabla 8.
*Servicios de datos*

| **Ruta** | **Método HTTP** | **Descripción** |
|---|---|---|

| | | |
|---|---|---|
| /historic | GET | Retorna los mensajes almacenados dependiendo de los parámetros que se le envíen. |
| /message | POST | Permite al usuario hacer una difusión de un mensaje a los actuadores a los que desee que les llegue. |
| /subscribe/{topico} | GET | Es usado cuando se registra un nuevo gateway y se debe realizar la subscripción al topico del mismo en él broker. |
| /statistics/{idUsuario} | GET | Retorna las estadísticas históricas de los mensajes enviados por los procesos y gateways de determinado usuario. |

### 6.5 Validación

Una vez desarrollado por completo el API de servicios, se procedió a probar cada uno de estos. El software utilizado para las pruebas fue Postman, una solución para desarrolladores que permite realizar solicitudes a servicios web y examinar sus respuestas.

A continuación, un ejemplo del desarrollo de estas pruebas, utilizando el método “editar usuario”.

- Para cada servicio, se realizó una solicitud correcta a la base de datos, y se guardó tanto la solicitud como la respuesta.

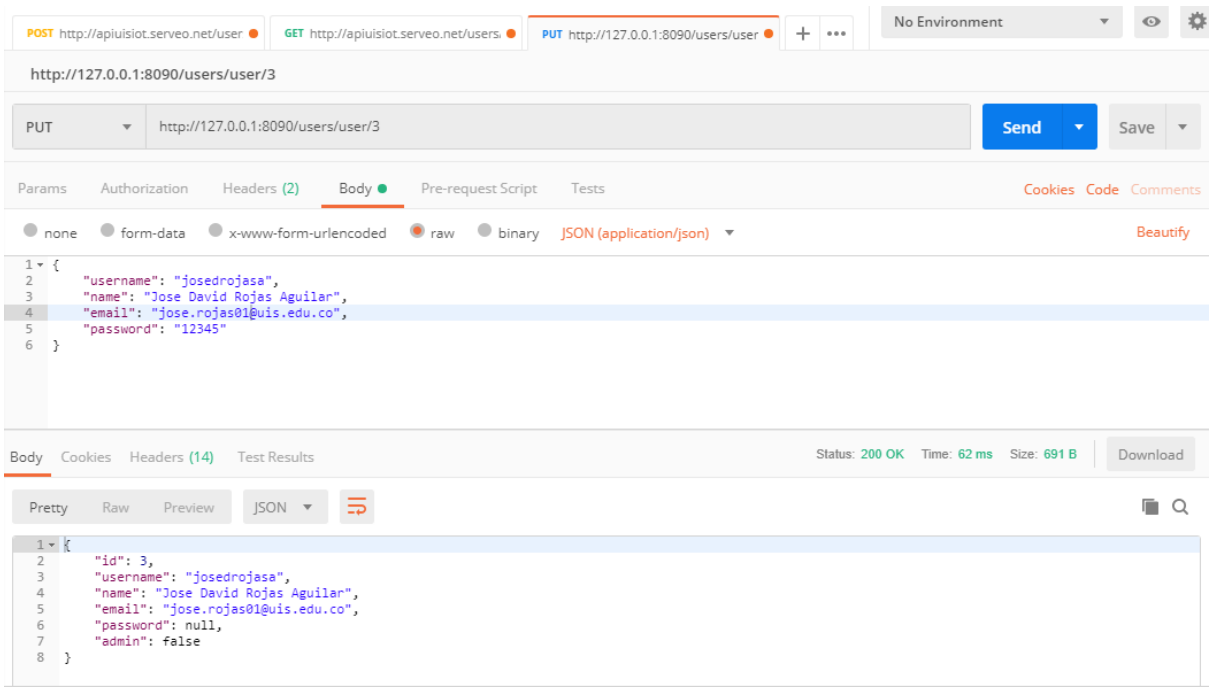


*Figura 19.* Solicitud y respuesta del servicio editar usuario en Postman.

- Para el caso de los servicios que modifican la base de datos, una vez recibida la respuesta del servicio, y si esta era positiva, se procedía a mirar la base de datos en busca de los cambios. De esto también se hizo una captura de pantalla.

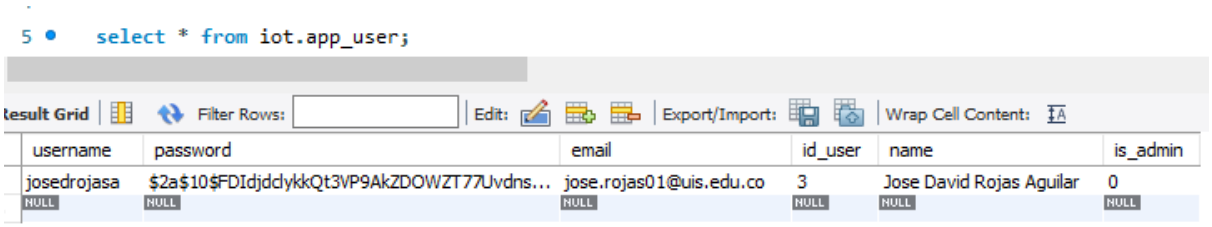


*Figura 20.* Evidencia en la base de datos de la ejecución exitosa del servicio de actualizar usuario.

- Una vez ejecutado el caso exitoso, se procedió a probar el servicio con solicitudes erróneas, para verificar el correcto manejo de excepciones. No se tomaron capturas de pantalla de estas solicitudes.
- En el caso de los servicios que se comunican con el gateway, se solicitó al autor del proyecto correspondiente una copia del jar del framework para correr una instancia local y así probar la correcta integración entre los sistemas y el correcto flujo de datos en la infraestructura.

**6.6.1 Pruebas de estrés.** En conjunto con el proyecto de definición de la infraestructura se diseñaron unas pruebas para verificar la capacidad del sistema de soportar la concurrencia de peticiones. A continuación, se muestran las tablas con los resultados de la aplicación de estas pruebas.

Tabla 9. *Tiempo promedio de respuesta para 1000 solicitudes por minuto*

| **Escenario** | **Milisegundos** |
|---|---|
| **Monolítico** | 413 |
| **1 instancia** | 455 |
| **2 instancias** | 434 |
| **3 instancias** | 419 |

Tabla 10. *Número máximo de solicitudes por minuto sin solicitudes fallidas*

| Escenario | Solicitudes por minuto |
|---|---|
| **Monolítico** | 2690 |
| **1 instancia** | 2616 |
| **2 instancias** | 3800 |
| **3 instancias** | 4320 |

Tabla 11. *Porcentaje de solicitudes fallidas para 3500 solicitudes por minuto*

| Escenario | % |
|---|---|
| **Monolítico** | 24.26 |
| **1 instancia** | 36.28 |
| **2 instancias** | 0 |
| **3 instancias** | 0 |

### 6.6.2. Análisis de resultados

- Podemos observar con los resultados de la tabla 9 que el ambiente monolítico es el de mejor respuesta, esto debido a que no requiere comunicarse a través de la red. Sin embargo, para el caso en el que hay 3 instancias, el tiempo de respuesta ha disminuido hasta ser casi idéntico al del monolito.

- En la tabla 10 podemos observar que el número máximo de solicitudes por minuto sin fallos aumenta conforme aumenta el número de instancias. Sin embargo, la mejora es menor entre mayor sea el número de instancias, lo que implica que eventualmente la mejora va a hacerse despreciable.

- De acuerdo con los datos de la tabla 11, a pesar de que el porcentaje de error en las solicitudes aumenta en el escenario de 1 instancia respecto al monolito, este error mejora drásticamente cuando se genera una instancia más, disminuyendo a 0 el error.

- En términos generales, se evidencia que el potencial de la infraestructura se aprovecha mayormente cuando existen múltiples instancias de los servicios.

### 6.6 Caso de uso: Implementación e Implantación

En esta sección se realizó el despliegue de la plataforma IoT enfocada a Smart Campus, implementado un prototipo para demostrar que todos los componentes se integran correctamente. Para lograr esto, fue necesaria la participación de los autores de los proyectos de grado mencionados en los capítulos anteriores.

En el escenario planteado un supervisor desea activar una red de bombillos ubicados en dos laboratorios de ingeniería eléctrica cuando el voltaje de un generador eléctrico pase un umbral de seguridad para alertar al personal presente en dichos espacios.

Además de esto, el usuario desea monitorear la temperatura generada en otro laboratorio en el cual se almacenan algunas sustancias químicas que deben permanecer constantemente sobre una temperatura mínima. Para esto, instala un sensor de temperatura en el laboratorio, y un led en

su oficina, el cual deberá encenderse en caso de que la temperatura en el laboratorio descienda bajo un umbral previamente determinado.

Para simular el escenario anterior se hizo uso de dos dispositivos tipo gateway, un minicomputadora con pines análogos, tres actuadores (LEDs), un sensor de temperatura y un potenciómetro; a su vez, se crearon 4 procesos y 1 aplicación externa en la que se muestran los datos capturados en el caso de uso.

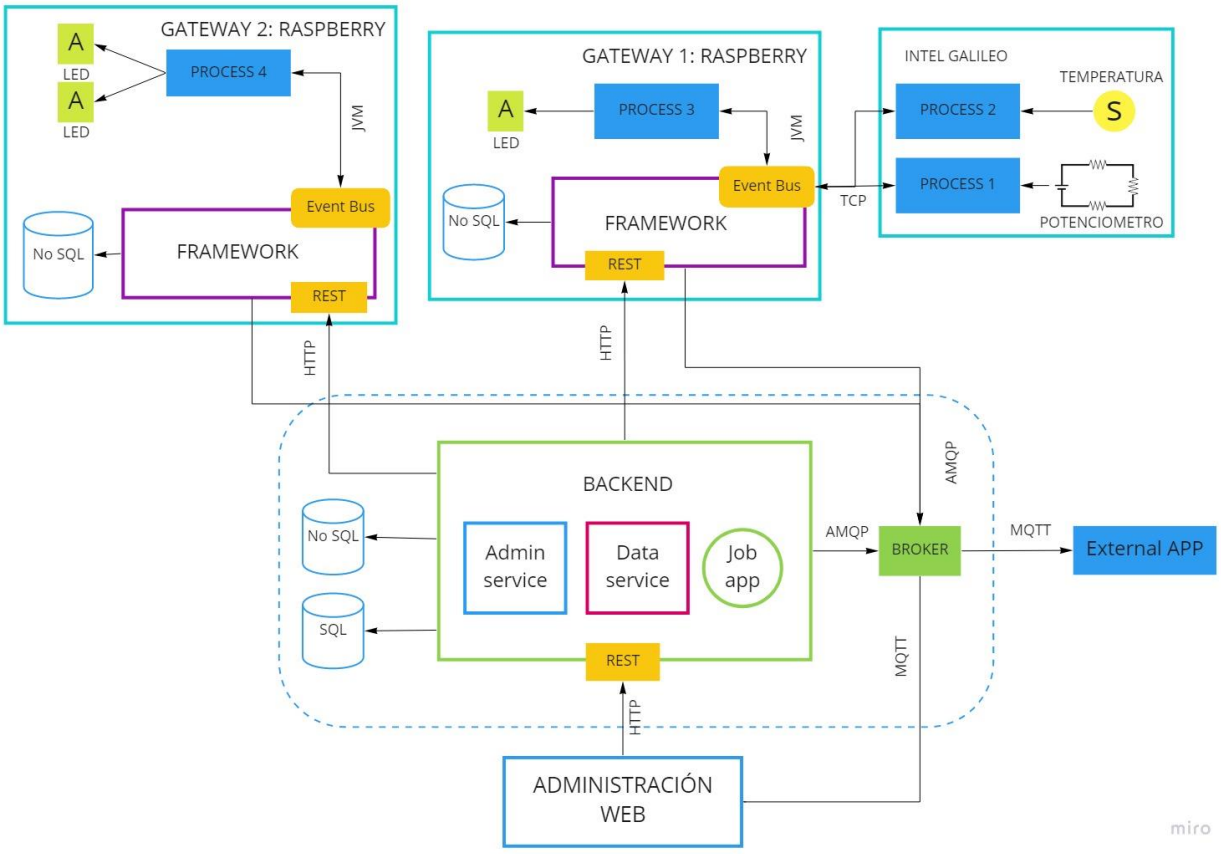


*Figura 21.* Arquitectura planteada para el caso de uso

En el Apéndice D se muestra un vídeo explicativo sobre la ejecución de la plataforma IoT con cuatro procesos en dos gateway

Los procesos que están conectados a los LED están programados para encender estos cuando reciben una señal determinada. Los otros dos, encargados del sensor de temperatura y de recibir los datos generados por el circuito eléctrico, envían datos constantemente al gateway, que a su vez los publica en una cola para que los reciba el backend, el cual se encarga en primer lugar de almacenarlos en la base de datos para una potencial consulta de históricos y en segundo lugar de publicarlos en un topic para el consumo de las aplicaciones externas.

Cuando la señal, sea de voltaje o de temperatura, está por fuera de los rangos definidos, la aplicación envía una instrucción en una petición REST al backend, y este busca la información necesaria para saber exactamente a que actuadores y a que gateways pertenece esa instrucción y se encarga de enviarla.

Para este ejemplo, se desplegaron 3 instancias de los dos microservicios del backend, esto para poder asegurar el funcionamiento de la plataforma en caso de que alguna de estas falle. El despliegue también se encarga de levantar una nueva instancia si alguna llega a falla

## 7. Trabajo a futuro

En caso de que algún desarrollador desee continuar mejorando este proyecto, los dos factores principales que sería recomendable tener en cuenta son los siguientes:

En el momento, el modelo está concebido para que cada aplicación sea manejada por un solo usuario. Sin embargo, puede ocurrir el caso en que los desarrolladores de una aplicación deseen tener más de un administrador de la aplicación, o deseen diferentes roles de administración. Se podría ampliar el modelo para dar soporte a esta funcionalidad.

También, un módulo adicional que mejoraría notablemente el proyecto, sería un rule engine, que basado en los datos recibidos retorne unas instrucciones para los actuadores de forma automática. Esto facilitaría la automatización de procesos en el campus.

# 8. Conclusiones

- Una plataforma IoT para Smart Campus requiere un soporte backend con dos propiedades esenciales: alta disponibilidad y escalabilidad. Con el fin de proveer estas propiedades se diseñó una arquitectura basada en microservicios que permite manejar una gran cantidad de dispositivos y los datos que estos generan de manera eficiente.

- Con el API de servicios REST que se elaboró, es posible administrar el modelo de la plataforma planteada y además obtener también los datos generados por los dispositivos que se integren en la plataforma.

- Se comprobó que es posible integrar y soportar un conjunto de dispositivos en diferentes gateways, recibiendo y enviando datos a estos.

- El software que se construyó es fácilmente extensible gracias a la arquitectura de microservicios y el framework (Spring Boot) usado puesto que cada una de las entidades maneja su lógica por separado, es decir, están débilmente acoplados y por lo tanto, se pueden añadir nuevos componentes como microservicios o servicios que provean la plataforma de nuevas funcionalidades.

- Si bien el protocolo más usado en IoT es MQTT por su ultra bajo consumo de recursos, dado que se usó una arquitectura de microservicios para la alta disponibilidad y escalabilidad, esto trae consigo que existan múltiples instancias que tomen los datos del bróker, esto podría generar duplicidad en los datos que se toman puesto que MQTT solo

funciona con tópicos, por ello se decidió usar AMQP que es un protocolo también de bajo consumo pero que permite el uso de colas y soluciona el problema de la duplicidad.

- Se logró también, mediante un servicio adicional, permitir al usuario saber el estado (si están disponibles o no) de los dispositivos y gateways que hagan parte de la plataforma, para que pueda tomar las acciones que crea necesarias con base en esta información.

## Referencias Bibliográficas

Apache. (s.f). Clustering. Recuperado de https://activemq.apache.org/clustering.html

Bera, A. (2019). 80 Mind-Blowing IoT Statistics (Infographic). Recuperado de https://safeatlast.co/blog/iot-statistics/

Bharadwaj, A. (2016). What is the simplest meaning of Internet of Things? Recuperado de https://www.quora.com/What-is-the-simple-meaning-of-Internet-of-Things

Blet, N. (s.f.). Fiabilidad y tolerancia a fallos. Recuperado de https://www.dsi.fceia.unr.edu.ar/images/capitulo516.pdf

Brown, K. (2016). Más allá de clichés: Una breve historia de los patrones de microservicios. Recuperado de https://www.ibm.com/developerworks/ssa/cloud/library/cl-evolution-microservices-patterns/index.html

Camacho, D. (2019). Diseño de una aplicación Web extensible para la administración de una plataforma IoT diseñada para Smart Campus. Bucaramanga, Colombia.

Castro, L. (2016). ¿Qué es escalabilidad? Recuperado de https://www.aboutespanol.com/que-es-escalabilidad-157635

Codecademy. (s.f.). What is REST. Recuperado de https://www.codecademy.com/articles/what-is-rest

Comptia. (2015). Sizing Up the Internet of Things. Recuperado de: https://www.comptia.org/resources/sizing-up-the-internet-of-things

Evans, D. (2011). Internet de las Cosas: Cómo la próxima evolución de Internet lo cambia todo. Recuperado de https://www.cisco.com/c/dam/global/es_mx/solutions/executive/assets/pdf/internet-of-things-iot-ibsg.pdf

Gutiérrez, C. (2019). Diseño de un framework software extensible para dispositivos tipo gateway integrados en plataformas IoT para Smart Campus. Bucaramanga, Colombia.

HTTP Methods. (s.f.). Recuperado de https://restfulapi.net/http-methods/

Iot for all. (2018). Top 3 Programming Languages for IoT Development In 2018. Recuperado de https://www.iotforall.com/2018-top-3-programming-languages-iot-development/

Johansson, L. (2014). What is message queuing? Recuperado de https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html

Kirkland, J. (2017). Cinco formas en que el código abierto acelera la IoT. Recuperado de https://searchdatacenter.techtarget.com/es/opinion/Cinco-formas-en-que-el-codigo-abierto-acelera-la-IoT

McCauley, R. (2016). Using Alexa Skills Kit and AWS IoT to Voice Control Connected Devices. Recuperado de https://developer.amazon.com/de/blogs/post/Tx3828JHC7O9GZ9/Using-Alexa-Skills-Kit-and-AWS-IoT-to-Voice-Control-Connected-Devices

Microsoft Azure. (2019). Azure IoT reference architecture. Recuperado de https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/iot/

McCauley, R. (2016). Using Alexa Skills Kit and AWS IoT to Voice Control Connected Devices. Recuperado de https://developer.amazon.com/de/blogs/post/Tx3828JHC7O9GZ9/Using-Alexa-Skills-Kit-and-AWS-IoT-to-Voice-Control-Connected-Devices

Mozilla. (2019). HTTP. Recuperado de https://developer.mozilla.org/es/docs/Web/HTTP

Netflix. (2014). Eureka at a Glance. Recuperado de https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

Netflix. (2018). Hystrix: Latency and Fault Tolerance for Distributed Systems. Recuperado de https://github.com/Netflix/Hystrix

Press, G. (2010). A Very Short History Of The Internet Of Things. Recuperado de http://www.forbes.com/sites/gilpress/2014/06/18/a-veryshort-history-of-the-internet-of-things/

Rouse, M. (2018). Advanced Message Queuing Protocol (AMQP). Recuperado de https://whatis.techtarget.com/definition/Advanced-Message-Queuing-Protocol-AMQP

Rojas, J. (2019). Definición de una infraestructura cloud de alta disponibilidad en un entorno distribuido para el despliegue de una plataforma IoT. Bucaramanga, Santander

Ruckus. (s.f). Introducing the Smart Campus. Recuperado de https://ruckus-www.s3.amazonaws.com/pdf/solution-briefs/sb-smartcampus-ebook.pdf

Skaria, S. (2018). Edge computing vs. Cloud computing: Where does the future lie? Recuperado de https://www.linkedin.com/pulse/edge-computing-vs-cloud-where-does-future-lie-saju-skaria/

Solway Communications. (2018). What is edge computing and why should I care? Recuperado de https://www.solwaycomms.com/edge-computing/

Stack Overflow. (2018). Developer Survey Results 2018. Recuperado de https://insights.stackoverflow.com/survey/2018

Wasson, M., Roberts, J., Wilson, M., Buck, A. & Celarier, S. (2018). Microservices architecture style. Recuperado de https://docs.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices

Watts, S., Shiff, L. (2018). An Overview of Monolithic vs Microservices Architecture (MSA). Recuperado de https://www.bmc.com/blogs/microservices-architecture/