



UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA

Tecnologías Geoespaciales en Plataformas de Smart Cities

INFORME DE PROYECTO DE GRADO PARA LA OBTENCIÓN DEL
TÍTULO DE INGENIERO EN COMPUTACIÓN

Mayo, 2021

Autores:

Gastón Abellá - gaston.abellalopez@gmail.com
Ana Machado - ampeigonet@gmail.com
Daniel Susviela - d.susviela@gmail.com

Tutores:

Raquel Sosa
Bruno Rienzi

Resumen

Actualmente la mayor parte de la población mundial vive en ciudades, y esta tendencia ha ido en aumento a lo largo de los últimos años. Como consecuencia, ha crecido la necesidad de resolver problemáticas como la contaminación o la accesibilidad a los servicios provistos en una ciudad. Para solucionar estos problemas mediante el uso de la tecnología, surge el concepto de *Smart Cities* (ciudades inteligentes).

El desarrollo de las *Smart Cities* se apoya principalmente en tecnologías del área de *Internet of Things* (IoT), así como en otras tecnologías relacionadas a los Sistemas de Información Geográfica (SIG). El área de IoT presenta distintas plataformas que se encargan del manejo de datos provenientes de sensores y otras fuentes de información en una ciudad. En particular FIWARE, una plataforma IoT de código abierto facilita, mediante el desarrollo de una serie de componentes y estándares, el despliegue de aplicaciones para ciudades inteligentes. A su vez, casi la totalidad de los sensores ofrecen datos geográficos, por lo que queda abierta la posibilidad de incluir tecnologías geoespaciales que exploten dicha información.

En los últimos años se realizaron distintos trabajos que integran plataformas de *Smart Cities* con componentes geoespaciales, pero resultaron en soluciones no estandarizadas y específicas a cada problema. Por esto, surge la necesidad de estudiar la integración de estas tecnologías utilizando estándares existentes, con el fin de alcanzar una solución que pueda ser reutilizada en distintos dominios.

Este proyecto estudia la integración de FIWARE con tecnologías geoespaciales que siguen estándares del *Open Geospatial Consortium* (OGC). En base al estudio se propone una plataforma extendida con dichas tecnologías con el fin de facilitar el desarrollo de aplicaciones para *Smart Cities* con capacidades geográficas avanzadas. La plataforma propuesta maneja la recepción de datos de dispositivos, y ofrece un conjunto de servicios Web geoespaciales para acceder a dichos datos y operar con ellos.

Para validar la solución, se implementaron prototipos a partir de dos casos de aplicación utilizando datos geográficos de Montevideo. Uno de ellos resuelve la detección de desvíos en líneas de ómnibus pertenecientes al Sistema de Transporte Metropolitano. El otro permite obtener información en tiempo real sobre un conjunto de playas de la ciudad, y brinda un *ranking* de las mejores según las preferencias del usuario, considerando la densidad de personas presentes y el aforo de cada playa.

Palabras claves: Smart Cities, IoT, SIG, OGC, FIWARE.

Índice

1. Introducción	1
1.1. Contexto y motivación	1
1.2. Objetivos	1
1.3. Aportes del proyecto	2
1.4. Organización del documento	2
2. Marco conceptual	3
2.1. Marco teórico	3
2.1.1. Sistemas de Información Geográfica	3
2.1.2. Estándares OGC	5
2.1.3. IoT	6
2.1.4. Smart Cities	10
2.1.5. FIWARE	11
2.2. Trabajos relacionados	19
2.2.1. A Data Integration Approach for Smart Cities: The Case of Natal	20
2.2.2. Designing a FIWARE Cloud Solution for Making your Travel Smoother: the FLIWARE Experience	21
2.2.3. Trabajo realizado en la Intendencia de Montevideo	23
3. Análisis y solución propuesta	27
3.1. Análisis	27
3.1.1. Estudio de trabajos relacionados	27
3.1.2. Análisis sobre componentes SIG	29
3.2. Solución propuesta	30
3.2.1. Arquitectura	30
3.2.2. Diseño	33
3.3. Casos de aplicación seleccionados para desarrollar el prototipo	35
4. Implementación de la plataforma	37
4.1. Configuración FIWARE	37
4.1.1. Service group	37
4.1.2. Alta de sensor	38
4.1.3. Suscripción FIWARE	40
4.2. Servidor SIG	42
4.2.1. Features	43
4.2.2. Procesos	45
4.2.3. Módulos de ayuda implementados para procesos	47
5. Aplicación de gestión de sensores	51
5.1. Introducción	51
5.2. Funcionalidades	51
5.2.1. Crear sensores	52
5.2.2. Editar sensores	53
5.2.3. Simular datos	54

5.2.4. Uso avanzado	54
5.3. Arquitectura	55
5.4. Diseño	56
5.4.1. Creación de sensores	57
5.4.2. Simulación de datos	57
5.5. Implementación	58
5.5.1. Representación de datos	58
5.5.2. Modelos de datos usados para sensores de playa y ómnibus	60
5.5.3. Algoritmo de simulación	62
6. Aplicaciones de usuario implementadas	65
6.1. Aplicación de desvíos de ómnibus	65
6.1.1. Funcionalidades	65
6.1.2. Arquitectura	69
6.1.3. Diseño	72
6.1.4. Implementación	73
6.2. Aplicación de ranking de playas	75
6.2.1. Funcionalidades	75
6.2.2. Arquitectura	78
6.2.3. Diseño	81
6.2.4. Implementación	82
6.3. Conclusiones	85
7. Gestión del proyecto	87
7.1. Cronograma de actividades	87
7.2. Metodología de trabajo	88
8. Conclusiones y trabajos futuros	91
8.1. Conclusiones	91
8.2. Trabajos futuros	91
9. Referencias	95
Glosario	99

1. Introducción

1.1. Contexto y motivación

Las ciudades juegan un rol muy importante en varios aspectos económicos, sociales y medioambientales a nivel mundial. En un estudio realizado por la ONU (Organización de las Naciones Unidas) en 2018 se presentó una proyección que decía que para el 2050 aproximadamente un 68 % de la población mundial viviría en áreas urbanas [1]. La mayor parte de los recursos son consumidos en ciudades [2]. Naturalmente, esto es causa de preocupación no sólo por la protección al medio ambiente, sino también por la calidad de vida de los habitantes en las ciudades. Como parte de la solución, y en paralelo con avances de algunas tecnologías, emergen las *Smart Cities* (ciudades inteligentes) [2].

El desarrollo en el marco de *Smart Cities* se basa en el uso de TICs (Tecnologías de la información y comunicación), en particular IoT (*Internet of Things*) [3]. Dentro de IoT, existen plataformas como FIWARE¹, que permiten gestionar la recepción de información de sensores y otras fuentes [4]. Asimismo, para múltiples aplicaciones, es necesario contar con la localización relacionada a los sensores [5]. Por lo cual resulta interesante explorar la inclusión de tecnologías geoespaciales en el desarrollo de *Smart Cities*.

Durante los últimos años en el mundo y la región, se han visto múltiples iniciativas de aplicaciones en plataformas de *Smart Cities* que contienen componentes geoespaciales [6][7]. Particularmente en Uruguay, la Intendencia de Montevideo ha estado trabajando con tecnologías geoespaciales y con componentes FIWARE para desarrollar sistemas en el marco de ciudades inteligentes.

Sin embargo, la integración de la plataforma FIWARE con componentes geoespaciales suele ser específica al problema que se intenta resolver [6][7]. Como consecuencia las capacidades de los componentes geoespaciales de las soluciones están limitadas al dominio que se quiere atacar. Esto significa que dichas integraciones son poco reutilizables en nuevos problemas ya que carecen de generalidad.

Surge entonces la motivación de explorar integraciones entre plataformas basadas en FIWARE y los componentes geoespaciales, que utilicen estándares existentes y que no estén limitadas a una problemática específica.

1.2. Objetivos

El objetivo general del proyecto es investigar la integración de plataformas para *Smart Cities* con tecnologías geoespaciales basadas en estándares. En particular la plataforma para *Smart Cities* elegida para este trabajo es FIWARE.

Para cumplir con este objetivo, se plantean los siguientes objetivos específicos:

- Relevar trabajos realizados con la plataforma FIWARE y tecnologías geoespaciales.
- Evaluar alternativas de integración entre FIWARE y tecnologías geoespaciales ba-

¹<https://www.fiware.org/>

sadas en estándares OGC, y proponer la arquitectura de una plataforma que soporte dicha integración.

- Desarrollar un prototipo de la plataforma propuesta.
- Implementar prototipos de aplicaciones para probar la plataforma con datos geoespaciales de Montevideo.

1.3. Aportes del proyecto

Se listan a continuación los aportes del proyecto:

- Análisis de trabajos basados en FIWARE que integran tecnologías geoespaciales, identificando elementos y buenas prácticas comúnmente aplicadas en las arquitecturas de este tipo de plataformas.
- Documentación de las principales características de la arquitectura de la plataforma FIWARE desarrollada por la Intendencia de Montevideo, que se utiliza para varias aplicaciones públicas y de uso interno.
- Propuesta de una plataforma con capacidades geoespaciales basada en FIWARE. La misma consiste en una arquitectura y los flujos de comunicación entre sus componentes, y se basa en las buenas prácticas resultantes del análisis.
- Implementación de un prototipo de la plataforma que consiste en una aplicación Web de gestión y un servidor que provee capacidades geoespaciales.
- Implementación de aplicaciones Web que se basan en dos casos de aplicación que utilizan datos geográficos de Montevideo, utilizadas para validar la plataforma.

1.4. Organización del documento

El resto de este documento se organiza de la siguiente manera. En el Capítulo 2 se desarrolla el marco conceptual donde se presentan los conceptos clave para el documento y los trabajos relacionados estudiados a lo largo del proyecto. El Capítulo 3 presenta el análisis realizado para proponer una solución e incluye los aspectos principales de la arquitectura y diseño de la plataforma propuesta.

En los Capítulos 4, 5 y 6 se presentan aspectos sobre la implementación tanto de la plataforma (Capítulo 4), así como de las aplicaciones (Capítulos 5 y 6). En estos últimos también se presentan funcionalidades de las mismas.

Luego, el Capítulo 7 presenta la metodología de trabajo utilizada por el equipo a lo largo del proyecto. Finalmente, en el Capítulo 8 se encuentran las conclusiones y se presentan trabajos futuros que se podrían realizar.

2. Marco conceptual

En este capítulo se desarrolla un marco teórico que contiene conceptos relevantes para la comprensión del presente trabajo. Entre ellos se encuentra información relacionada a Sistemas de Información Geográfica y sus estándares asociados, conceptos de *Internet of Things* y la plataforma FIWARE. En la segunda parte del capítulo, se presentan artículos que describen trabajos realizados en el marco regional e internacional que utilizan la plataforma FIWARE y componentes geoespaciales en el contexto de *Smart Cities*. Además se incluye la descripción del sistema desarrollado por la Intendencia de Montevideo como trabajo relacionado a nivel nacional.

2.1. Marco teórico

2.1.1. Sistemas de Información Geográfica

Un Sistema de Información Geográfica (SIG) es un conjunto de componentes responsables de mantener, almacenar y proveer datos geográficos [8][9].

Los SIG manejan principalmente datos físicos, biológicos, culturales, económicos o demográficos, por lo que se han convertido en activos importantes en distintas áreas de conocimiento como son las ciencias naturales, sociales, o la ingeniería [10]. Los distintos datos que se quieren almacenar se representan como entidades espaciales (*features*), también llamados como objetos cartográficos. Estos datos son almacenados en una base de datos espacial (*spatial database*) y son representados mediante un modelo de datos espacial (*spatial data model*), que se define como los objetos en la base de datos más las relaciones entre ellos [11].

Las coordenadas por su parte juegan un rol esencial en el modelado y representación de *features*, ya que son utilizadas para definir locaciones espaciales y extensiones de objetos geográficos. Estas pueden consistir en pares o triples de números que especifican la ubicación en relación a un punto de origen y cuantifican la distancia desde dicho origen. A su vez, los grupos de coordenadas son organizados para representar formas y límites que definen los objetos. Los modelos de datos difieren según cómo representan sus coordenadas y existen diversos estándares para definirlas [11].

Por otro lado, la manera más común de agrupar datos en los SIG es mediante capas (*layers*), conocidas también como capas temáticas (*thematic layers*). En éstas se organizan los atributos y datos espaciales para un conjunto dado de *features* en una región de interés. Algunos ejemplos son: capa de rutas, capa de suelos, capa de elevaciones, capa de población [11].

2.1.1.1 Sistemas de Referencia de Coordenadas

Un Sistema de Referencia de Coordenadas (SRC) define, con la ayuda de coordenadas, cómo las dos dimensiones de un punto se proyectan a ubicaciones reales de la tierra. Cada SRC define una proyección diferente de las coordenadas en una esfera (o elipsoide como la tierra) sobre un plano o mapa [12]. En particular se destaca el SRC

conocido como WGS (*World Geodetic System*) 84 que es un sistema que utiliza la tierra como centro, considerando su forma, tamaño, y otras medidas de interés [13].

2.1.1.2 Representación de datos

Para la representación de datos en los SIG, existen dos conceptualizaciones principales, vectorial y raster, presentadas gráficamente en la siguiente imagen [14].

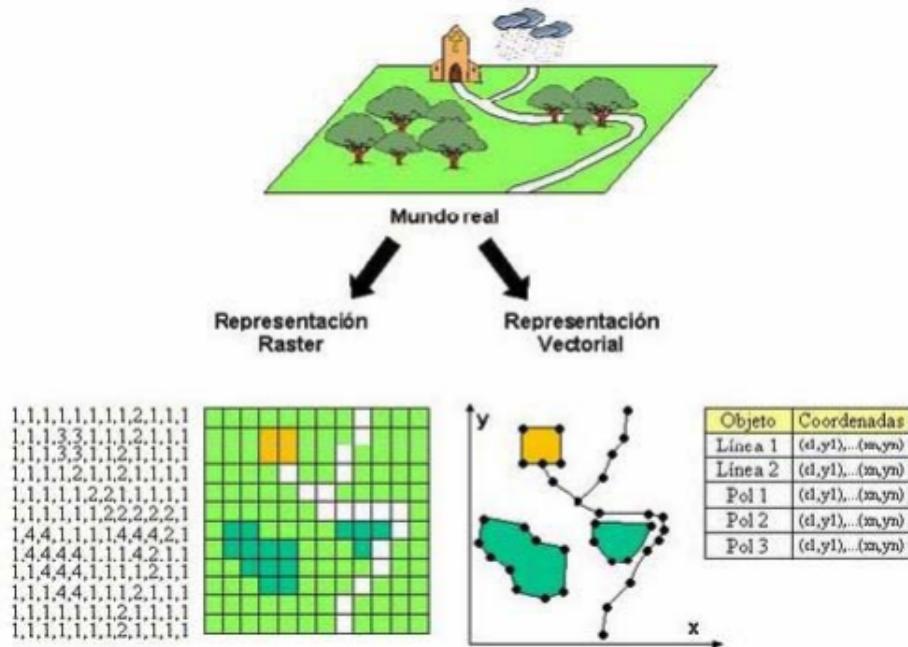


Figura 1: Modelo raster vs. Modelo vectorial [15].

Representación raster

El modelo de datos raster abstrae una determinada región geográfica en un conjunto de celdas que conforman una matriz. Cada celda tiene un dato asociado de lo que se desea modelar, y a lo largo de ellas se representan atributos que pueden variar continuamente a través de una región. Por esto los datos raster son convenientes para representar datos continuos de la realidad. Elevación, temperatura media y promedio de lluvia son algunos ejemplos de propiedades representadas como campos continuos [16]. En el ejemplo de la Figura 1 las celdas correspondientes al camino contienen datos relacionados al camino, mientras que las celdas que representan al edificio contienen otra información. A su vez, la posición de la celda dentro de la matriz brinda la información espacial de cada entidad.

Representación vectorial

El modelo de datos vectorial se basa en la representación mediante vectores de la realidad geográfica que se quiere modelar. Las unidades básicas del modelo son los

puntos, las líneas y los polígonos (*point*, *line* y *polygon* respectivamente). A su vez cada uno de ellos está representado en un conjunto de coordenadas previamente definidas [14]. En el ejemplo de la Figura 1 se pueden observar dos unidades básicas y cómo pueden representar entidades de la realidad. El camino se representa con una línea, mientras que los bosques y el edificio como polígonos.

2.1.2. Estándares OGC

Open Geospatial Consortium (OGC) es una organización internacional sin fines de lucro enfocada en el desarrollo y la definición de estándares abiertos para la comunidad geoespacial, con el fin de permitir la interoperabilidad entre distintos sistemas y servicios de datos [17]. OGC provee especificaciones de estándares con el objetivo de facilitar y motivar el uso de éstos, de los cuales se destacan los denominados OGC Web Services (OWS) y OGC Application Programming Interface (OGC API) para el desarrollo de servicios Web.

2.1.2.1 OWS

OWS es un conjunto de protocolos de servicios para acceso y procesamiento de datos. Los pedidos a OWS se hacen a través del protocolo HTTP (*Hyper Text Transfer Protocol*), y los parámetros pueden ser comunicados mediante pares de *key-value* en la URL del pedido o mediante XML (*eXtensible Markup Language*) [18]. Dentro de los mismos se destacan:

- **Web Map Service (WMS)** es un estándar para servir y consumir mapas dinámicos en la Web. Provee una interfaz por HTTP a la cual se le pueden hacer pedidos de mapas que se encuentran en bases de datos geoespaciales. Un pedido a WMS define las capas y la zona a procesar, y la respuesta es una imagen con el mapa (generalmente en formato JPEG o PNG) correspondiente a los parámetros enviados en la solicitud [19].
- **Web Feature Service (WFS)** es un estándar que define operaciones de descubrimiento, creación, consulta, y transacción de datos de *features* representadas de forma vectorial. Asimismo provee operaciones para gestionar el acceso a los datos, y para el almacenamiento y manejo de consultas parametrizadas [20].
- **Web Processing Service (WPS)** es un estándar para publicar y ejecutar procesos geoespaciales. El mismo provee reglas para estandarizar las entradas y salidas de dichos procesos, definiendo también cómo los clientes deben hacer los pedidos de ejecución de los diversos procesos y cómo la salida de los mismos debe ser interpretada [21].

2.1.2.2 OGC API

Los estándares de OGC API definen bloques de construcción de APIs modulares con el fin de construir APIs Web espaciales de una manera consistente aplicando patro-

nes REST (*REpresentational State Transfer*)². La familia de estándares OGC API se organiza por tipo de recurso espacial, definidos a continuación [22].

OGC API: Features

Es un estándar desarrollado por OGC que proporciona bloques de construcción de APIs para crear, modificar y consultar *features* en la Web. Éste se compone de varias partes, siendo cada una de ellas un subestándar separado [23].

- **Parte 1 - Core.**

El “Core”, especifica las capacidades básicas y está restringido a buscar *features* cuyas geometrías se encuentran representadas en el Sistema de Referencia de Coordenadas WGS 84. Además, ofrece operaciones de consulta para datos y metadatos de colecciones de *features*, así como de consulta para *features* específicas [24].

- **Parte 2 - Coordinate Reference Systems by Reference.**

Esta parte extiende las capacidades del “Core”, con la habilidad de usar SRC distintos a los definidos por defecto. En resumen, extiende las operaciones anteriores permitiendo expresar qué SRC debe tener el conjunto de datos de respuesta, así como la forma de pedirlos en un sistema de coordenadas específico [25].

- **Parte 3 - Common Query Language (CQL).**

En esta extensión del estándar se propone mejorar las capacidades de consulta sobre las *features*, introduciendo la capacidad de uso de CQL³ como mecanismo de filtrado. También se permite el uso de este lenguaje para la inserción o actualización condicional de elementos [26].

- **Parte 4 - Simple Transactions.**

Es la última parte del conjunto de estándares OGC API Features, que provee la habilidad de insertar, reemplazar, modificar o borrar *features* individuales dentro de una colección. Se introduce el uso de nuevos métodos HTTP (POST, PUT, DELETE) cumpliendo con los requerimientos que debe cumplir un servicio REST [27].

OGC API Processes

Es un estándar que se encuentra en proceso de desarrollo por OGC que busca la especificación de una API REST, la cual debe servir para la instanciación de procesos y la obtención de datos y metadatos sobre estos. Estos procesos a priori pueden trabajar con cualquier tipo de dato, pero típicamente suelen estar asociados a los tipos de datos geográficos mencionados anteriormente [28].

2.1.3. IoT

El Internet de las cosas o mejor conocido como *Internet of Things* (IoT) refiere a la idea de la interconexión y cooperación de dispositivos que se conectan entre sí mediante

²<https://restfulapi.net/>

³<https://docs.geoserver.org/latest/en/user/tutorials/cql/cqlTutorial.html>

Internet [29]. La necesidad surge a partir de la automatización de la recolección de datos ya que en la actualidad casi la totalidad de los 50 petabytes de información disponible en Internet fue recolectada por personas [30].

El IoT es un área grande en el mundo de la computación y como tal, se pueden estudiar varios aspectos dentro de la misma. Existe una rama concentrada en el desarrollo de los sensores y los actuadores, otra enfocada en la investigación de los protocolos de comunicación entre los sensores y por último se encuentra la visión enfocada en los problemas comunes del área [29].

La siguiente información es recopilada en su totalidad de la misma referencia [31].

Al hablar de sistemas IoT, independientemente de su contexto, en general se pueden encontrar los componentes mostrados en la Figura 2.

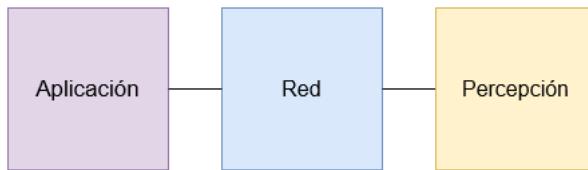


Figura 2: Capas genéricas de un sistema IoT [31].

La capa de percepción es la que se encuentra más cercana al mundo físico y es de donde salen los datos o se realizan acciones según el caso. Dentro de esta capa se encuentran las *Smart Things*. En la capa de aplicación se encuentra cualquier aplicación que haga uso de los datos provistos por sensores o los maneje de alguna forma. Por último, la capa de red es la que actúa como nexo entre las capas mencionadas anteriormente.

2.1.3.1 Sensores y actuadores

Los sensores y actuadores son la base de las plataformas IoT. Son la interfaz a través de la cual el software se comunica con el mundo real. Por un lado están los sensores, que son dispositivos que miden una variable física de interés. Por otro lado, un actuador es un dispositivo que afecta el ambiente, convirtiendo energía eléctrica a acciones en el mundo real. Los sensores y actuadores son a su vez orquestados por una plataforma para cumplir un fin [31].

Un caso de uso de ejemplo podría ser la aplicación de multas a vehículos en exceso de velocidad por una calle. Para este ejemplo se tiene un sensor de detección de velocidad que envía los datos a una plataforma IoT, y además una cámara de fotos la cual oficia de actuador, que tomará fotografías de los vehículos según la plataforma lo disponga. En la Figura 3 se puede apreciar un esquema sobre este caso de uso.

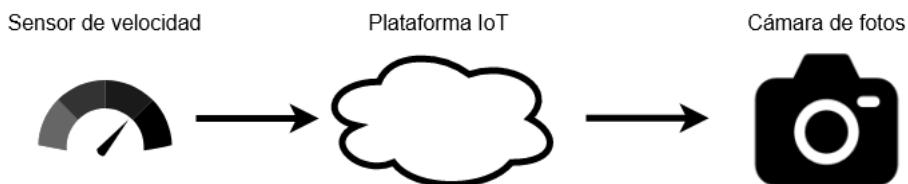


Figura 3: Ejemplo de sensores actuadores.

2.1.3.2 Protocolos de comunicación

Existen diversos protocolos empleados en la comunicación entre los distintos componentes de las aplicaciones IoT. A continuación se destacan algunos de ellos.

Protocolos de capa de aplicación

MQTT

La información sobre el protocolo MQTT (*Message Queuing Telemetry Transport*) de la siguiente sección es extraída de la especificación del protocolo disponible en la siguiente referencia [32].

Es un protocolo de conectividad máquina a máquina del IoT. Fue diseñado para ser súper ligero en la funcionalidad de publicación y suscripción de mensajes. Es extremadamente útil para las conexiones a lugares remotos donde es ideal tener dispositivos que admiten poca capacidad de cómputo y/o donde se quiere optimizar al máximo el uso de red de datos.

Se basa en el patrón de diseño publicación-suscripción, donde múltiples clientes pueden suscribirse a recibir datos de un área de interés. El protocolo incluye mensajes para establecer conexiones, y una vez que estas están prontas para el uso, los clientes solo pueden mandar mensajes de PUBLISH (publicar) o de SUBSCRIBE (suscripción).

MQTT trabaja sobre el protocolo TCP (*Transmission Control Protocol*), lo cual brinda el respaldo de que los mensajes no se pierdan mientras que la conexión TCP sea estable. El protocolo también agrega por encima de TCP dos niveles extra de seguridad en caso de que la conexión TCP no sea estable, teniendo un total de tres niveles de calidad de servicio.

HTTP

HTTP es un protocolo para aplicaciones distribuidas, colaborativas o sistemas de información. Es un protocolo genérico, que no preserva estado y puede ser empleado para la obtención, publicación, eliminación y alteración de recursos en diversos servicios. Estas funcionalidades se logran a través de los cabezales de información los cuales indican información sobre el pedido realizado. Además el protocolo estandariza códigos de errores, lo que permite una universalización de los casos de uso comunes [33].

Protocolos para datos

JavaScript Object Notation (JSON)

JSON es un protocolo de estructuración de datos basado en texto definido en el RFC 7159⁴. JSON define tipos de datos atómicos o compuestos, que facilitan el intercambio de información. Dentro de los tipos atómicos están los *strings* (cadenas), enteros

⁴RFC JSON - <https://datatracker.ietf.org/doc/html/rfc7159>

y booleanos. Por otro lado, dentro de los compuestos se encuentran los arreglos, que se pueden componer de cualquier tipo de dato, y los objetos que consisten en diccionarios cuyas claves deben ser *strings* y sus valores cualquier tipo de dato. JSON tiene la ventaja de ser sencillo de entender para las personas y computadoras [34].

Se muestra a continuación un ejemplo de un objeto JSON.

```
{
    "clave": "valor-string",
    "clave2": 1,
    "clave3": [1, "1", { "clave_anidada": true }]
}
```

Código 1: Ejemplo de objeto JSON

Ultralight 2.0

Ultralight 2.0 es un protocolo de envío de datos sencillos sobre texto, y esta diseñado para ahorrar ancho de banda, por lo que se caracteriza por su alto nivel de compacidad. En su expresión más simple, los datos se envían en pares clave-valor, donde la clave es (según las recomendaciones del protocolo) sólo una letra y los valores pueden ser números o palabras, ambos concatenados por el carácter *pipe* (|). La tira del mensaje se crea concatenando los pares mencionados anteriormente también con dicho símbolo. Además, Ultralight puede ser utilizado para el envío de comandos siguiendo la sintaxis dispositivo@comando | parámetros[35].

2.1.3.3 Plataformas IoT

En la siguiente sección se presentan las plataformas IoT en base a la fuente [36].

Una plataforma IoT es el conjunto de tecnologías y componentes que brindan soporte para la integración de sensores con las aplicaciones. Las plataformas IoT siguen por lo general la arquitectura esquematizada en la Figura 4.

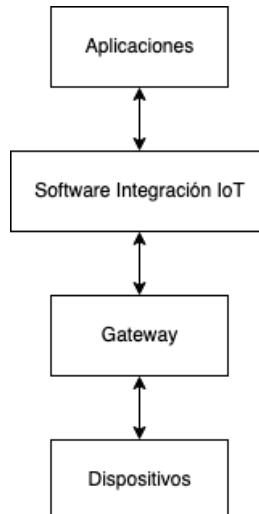


Figura 4: Arquitectura de una plataforma genérica IoT [36].

Las capas se presentan en orden de nivel de abstracción de la realidad, donde las capas superiores son las que están más lejos de las variables físicas que se estén midiendo o alterando.

La capa de “Aplicaciones” es la más alejada y se compone del software que cumple las funciones requeridas por la aplicación. Aquí se pueden tener aplicaciones que muestren los datos en tiempo real para su estudio o pueden también haber interfaces de los estados de los actuadores y sus efectos en el ambiente en el que están instalados.

La capa de “Software de Integración IoT” se compone del software que maneja a más bajo nivel los sensores y su estado. Es además la responsable de almacenar los datos de los mismos. Del punto de vista funcional, es el nexo entre los dispositivos y las aplicaciones anteriormente descriptas.

La capa de “Gateway” es un componente particular y puede no estar siempre presente. Se la incluye porque en general existe y funciona para organizar y coordinar varios flujos de datos de distintos sensores y actuadores en un único canal de comunicación.

Finalmente la capa de “Dispositivos” es en donde se encuentran los sensores y los actuadores de la solución IoT implementada.

2.1.4. Smart Cities

Si bien no existe una definición universal específica para el término *Smart City*, el consenso general habla de ciudades en las que se cuenta con componentes TICs⁵ para desarrollar y promover prácticas sustentables. Surgen como necesidad al problema del crecimiento de urbanización, y el uso poco eficiente de los recursos [2].

Los dominios sobre los que desarrollan aplicaciones en el marco de *Smart Cities* son todos los que puedan estar relacionados a aspectos de una ciudad, y en principio no es una lista definida. Sin embargo, varios autores mencionan la importancia de tener indicadores como fuente de información para volcar sobre el desarrollo de aplicaciones y sistemas. Para esto se recomienda la norma ISO:37120:2014⁶ (*Sustainable development of communities — Indicators for city services and quality of life*) que describe los indicadores, así como metodologías de medición para los mismos. Algunos dominios para los que existen indicadores son medioambiente, recreación, transporte, planeamiento urbano, entre otros.

Es por esto que el rol de las tecnologías IoT es fundamental para el desarrollo de ciudades inteligentes, ya que proveen respuestas para obtener información de los distintos aspectos relevantes para cada dominio. En la Figura 5 se presentan algunos ejemplos de aplicaciones IoT de las distintas verticales [3].

⁵Tecnologías de la Información y la Comunicación

⁶<https://www.iso.org/standard/62436.html>

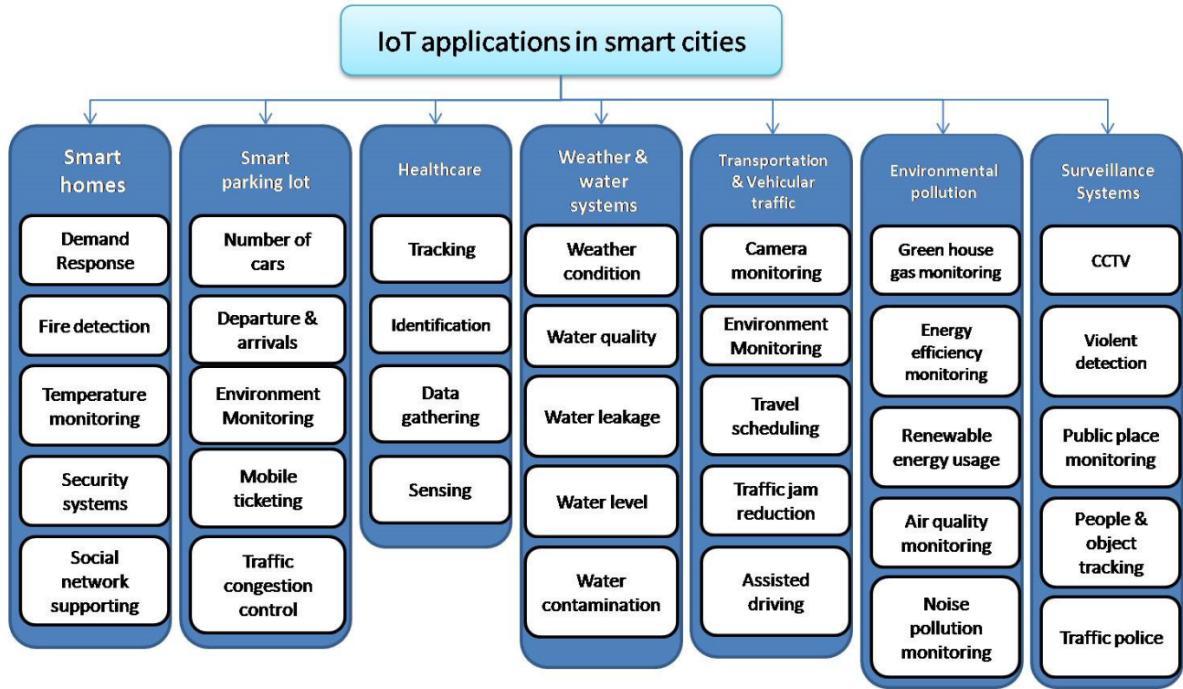


Figura 5: Aplicaciones IoT para Smart Cities [3].

2.1.5. FIWARE

FIWARE es una iniciativa *open source* que define un conjunto universal de estándares para el manejo de datos dentro de plataformas IoT, facilitando el desarrollo de aplicaciones para distintos dominios en el marco *Smart Cities*. Esta plataforma, surge como resultado de una serie de proyectos impulsados por la Unión Europea, y se promueve activamente por la FIWARE Foundation [4].

2.1.5.1 Arquitectura FIWARE

En esta sección se describe la arquitectura FIWARE en base a la fuente [37].

Como se explica en la sección 2.1.3, las plataformas de IoT tienen la necesidad de recolectar y manejar información de la realidad. Esta información proviene principalmente de sensores, entre otras fuentes, que reportan grandes volúmenes de datos a lo largo del tiempo. El conjunto de datos existentes en un instante del tiempo conforma el contexto en una plataforma IoT, y se lo conoce como datos de contexto o información de contexto.

Para satisfacer esta necesidad, FIWARE propone una arquitectura estándar de plataforma IoT basada en cuatro conjuntos de componentes, denominados Generic Enablers (GE). Cada GE a su vez está compuesto por diversas APIs y aplicaciones las cuales pueden ser incluidas o no según las necesidades de negocio. La Figura 6 muestra los conjuntos de GEs existentes.



Figura 6: GEes organizados por área [37].

Procesamiento, análisis y visualización del contexto

Grupo de componentes cuyo objetivo es procesar, analizar y visualizar información de contexto. Estos componentes implementan diversas funcionalidades que se adaptan a distintos casos de uso. Por ejemplo, existen módulos como WireCloud [38] que se encargan de la construcción de paneles y gráficos que presentan la información de contexto en forma amigable, así como Kurento [39] que permite realizar procesamiento en tiempo real de flujos (*streams*) de video y tratarlos como sensores.

Gestión del contexto

Los componentes de este conjunto implementan las herramientas necesarias para almacenar, acceder, procesar y analizar datos que conforman una aplicación inteligente. Asimismo algunos cuentan con la funcionalidad de permitir suscripciones a los cambios en la información permitiendo a clientes suscritos recibir datos en base a condiciones completamente personalizables.

De este grupo de GEes se destaca el componente Context Broker (CB) y en particular su implementación Orion, la más popular provista por FIWARE. Orion implementa la *Next Generation Service Interface API* (NGSI)⁷, una interfaz que ofrece operaciones CRUD⁸ para el manejo de datos de contexto. Esta API es un pilar fundamental en FIWARE, ya que estandariza las comunicaciones con el CB y flexibiliza la gestión del contexto, permitiendo que cualquier sistema que actúe como cliente pueda hacer uso de esta plataforma. A su vez, Orion utiliza MongoDB⁹ para almacenar la información del contexto.

⁷https://fiware-orion.readthedocs.io/en/1.13.0/user/walkthrough_apiv2/index.html

⁸CRUD (*Create, Read, Update, Delete*) es un acrónimo para las maneras en las que se puede operar sobre información almacenada.

⁹<https://www.mongodb.com>

Además de Orion existen implementaciones alternativas para el CB, así como otros componentes que ofrecen funcionalidades adicionales. Por ejemplo, hay GEs que permiten almacenar datos históricos de contexto a medida que se van reportando.

Interfaz con IoT, robots y sistemas de terceros

Comprende una serie de componentes cuya principal función es permitir y simplificar la comunicación con dispositivos IoT, Robots y sistemas de terceros. De esta forma se resuelve el problema de recolectar información de contexto valiosa y generar acciones como respuesta a actualizaciones en el contexto. Uno de los GEs más importantes en este grupo es el *Integrated Data Acquisition System* (IDAS), ya que se conforma de una amplia variedad de Agentes IoT que facilitan el intercambio de información con dispositivos.

FIWARE ofrece una amplia gama de agentes que sirven para conectar la NGSI API del CB con los distintos protocolos de los dispositivos. En particular existen Agentes IoT para HTTP/MQTT con payload JSON y Ultralight 2.0. A su vez FIWARE ofrece una librería con la que los desarrolladores pueden crear sus propios Agentes IoT, brindando así soporte a todos los protocolos.

Gestión, publicación y monetización de datos de contexto

GEs encargados principalmente de brindar funciones de seguridad sobre los componentes de cualquier sistema implementado con FIWARE. Algunas de las funcionalidades provistas por estos módulos son: autenticación, autorización, control de acceso, gestión de publicación de grupos de datos y asignación de precios y tarifas de pago por uso, entre otros.

Un ejemplo de GEs en este grupo es *Keyrock Identity Management* [40]. Este componente brinda soporte para implementar autenticación para usuarios y dispositivos, gestión de perfiles de usuario, preservación de privacidad de datos personales, entre otros.

2.1.5.2 FIWARE NGSI API

En esta sección se explica la FIWARE NGSI API en base a la fuente [41].

Todas las interacciones entre aplicaciones o componentes de plataforma y el Context Broker se hacen utilizando FIWARE NGSI RESTful API. Su última versión estable es la 2.0, por lo que es conocida también como NGSI v2.

Su especificación define:

- Un **modelo de datos** para información de contexto, basada en un modelo simple de información que utiliza la noción de entidades de contexto (*context entities*).
- Una **interfaz de datos de contexto** para intercambiar información por medio de operaciones de consulta, suscripción y actualización.
- Una **interfaz de disponibilidad de contexto** para el intercambio de información sobre cómo obtener datos de contexto.

Modelado de datos de contexto

El modelo de datos de la API NGSI v2 consiste esencialmente de tres elementos principales: las entidades de contexto, sus atributos y metadatos.

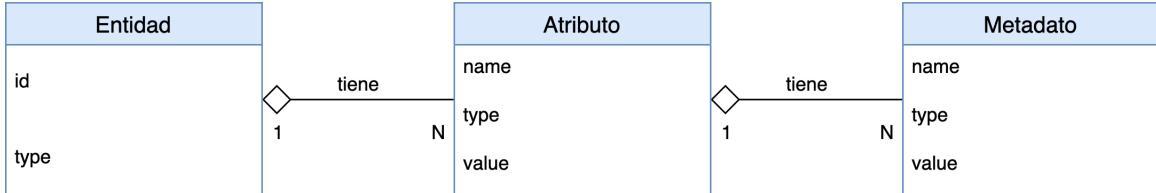


Figura 7: Modelo de datos de FIWARE NGSI v2 [41].

Las entidades de contexto, son el elemento más importante del modelado de información en NGSI v2. Una entidad representa un objeto (por ejemplo sensor, persona, habitación, etc.) en el sistema. Las mismas consisten de un identificador y un tipo, que describe la clase de objeto representado por ella. Por ejemplo, una entidad con id `sensor-365` podría tener el tipo `temperatureSensor`, representando un sensor de temperatura específico. Cada entidad se identifica de forma única por la combinación de su id y tipo.

Los atributos de contexto son propiedades de las entidades. Continuando con el ejemplo del sensor de temperatura, éste puede tener el atributo `current_temperature` que modela el valor más reciente de temperatura que mide el sensor. En el modelo de datos NGSI los atributos tienen un nombre, un tipo, un valor y pueden tener metadatos asociados.

Los metadatos son opcionales, y puede haber más de uno por atributo. Contienen un nombre, un tipo y un valor.

Representación de datos

El modelo anterior brinda una base a la hora de representar e intercambiar datos de contexto. Sin embargo dependiendo del caso de uso, queda a libre decisión del desarrollador utilizarlo en su máximo grado de expresividad o hacer uso de una versión simplificada del mismo. En ambos casos las entidades, atributos y metadatos se representan mediante objetos JSON, los cuales según el caso pueden tener los siguientes formatos:

- **Extendida.** Representa los atributos de una entidad mediante objetos los cuales contienen no solo el valor sino también los metadatos del mismo.

```
{  
    "id": "car001",  
    "type": "Car",  
    "location": {  
        "value": "41.3763726, 2.186447514",  
        "type": "geo:point",  
        "metadata": {  
            "timestamp": {
```

```
        "value": "2017-06-17T07:21:24.238Z",
        "type": "DateTime"
    }
}
}
}
```

Código 2: Repesentación extendida

- **Clave-valor.** Representa los atributos de una entidad únicamente por sus valores, quitando información sobre el tipo y los metadatos.

```
{
  "id": "R12345",
  "type": "Room",
  "temperature": 22
}
```

Código 3: Representación clave-valor

- **Solo valores.** Este modo representa las entidades como una lista de valores de atributos. No incluye identificador, ni información de tipos y metadatos.

```
[ 'Ford', 'black', 78.3 ]
```

Código 4: Representación solo valores

- **Valores únicos.** Este modo es igual que el de solo valores, con la salvedad de que los valores no se repiten.

Propiedades geoespaciales de entidades

El modelo de datos de FIWARE NGSI API permite representar propiedades geoespaciales de las entidades de contexto mediante atributos como los ya definidos. Existen dos maneras diferentes para representarlos: *Simple Location Format* y *GeoJSON*.

Simple Location Format

Este formato se define con el fin de representar las geometrías básicas (*point*, *line*, *box*, *polygon*) y no ubicaciones complejas de la superficie del planeta. Por ejemplo, aplicaciones que requieran capturar coordenadas de altitud deberán usar *GeoJSON*. Un atributo de una entidad representando una ubicación codificada con *Simple Location Format* debe seguir la siguiente sintaxis:

- El campo **type** del atributo debe ser uno de los siguientes: **geo:point**, **geo:line**, **geo:box** o **geo:polygon**.
- El campo **value** del atributo debe estar compuesto por un conjunto de coordenadas. Las coordenadas son definidas utilizando el sistema de referencia WGS84 Lat Long, EPSG:4326 [42]. Este conjunto de coordenadas permite decodificar la geometría basándose en el tipo. Para el caso del tipo **geo:point** el valor debe ser un string contenido un par válido latitud-longitud separado por una coma. Si es **geo:line**

o `geo:polygon` el valor debe contener un arreglo de pares de coordenadas donde el primer par debe ser el mismo que el último para el caso de los polígonos. En el caso del tipo `geo:box` el valor contiene una lista de exactamente dos coordenadas representando la esquina inferior izquierda y la superior derecha del rectángulo.

En el Código 2 se puede observar un ejemplo de este tipo de representación bajo el atributo `location`.

GeoJSON

GeoJSON [43] es un formato de intercambio de datos geoespaciales basado en el estándar JSON. Provee mayor flexibilidad que *Simple Location Format* debido a que permite la representación de altitudes de puntos y formas geoespaciales más complejas, como lo son las multigeometrías.

Un atributo de contexto representando una locación geográfica usando GeoJSON debe cumplir la siguiente sintaxis:

- El tipo NGSI del atributo debe ser `geo:json`
- El valor del atributo debe ser un objeto GeoJSON válido.

A continuación se muestra un ejemplo de representación de un atributo válido en formato GeoJSON.

```
{  
    "location": {  
        "value": {  
            "type": "Point",  
            "coordinates": [2.186447514, 41.3763726]  
        },  
        "type": "geo:json"  
    }  
}
```

Código 5: Ejemplo de atributo de contexto representado con GeoJSON

Suscripciones FIWARE

FIWARE incluye en NGSI v2 un mecanismo de suscripciones que permite notificar a sistemas externos cuando ciertas entidades reportan nuevos valores y cumplen determinadas condiciones. Este mecanismo se compone de un conjunto de operaciones CRUD sobre suscripciones bajo la ruta `/v2/suscriptions`. Cada una de ellas se modela mediante un objeto JSON que contiene:

- `id`: identificador único de la suscripción.
- `description`: campo opcional usado para describir la suscripción.
- `subject`: objeto JSON que representa el sujeto de la suscripción. Contiene una lista de entidades o patrones de entidades, y condiciones sobre atributos, que representan el objetivo de la suscripción.

- **notification**: objeto que describe la notificación a enviar cuando la suscripción es disparada. Contiene la especificación de qué atributos enviar al ente suscrito, así como atributos necesarios para establecer la comunicación con este, entre otros. Opcionalmente se puede además personalizar el objeto enviado en la notificación, facilitando así la integración con otras herramientas.
- **expires**: campo opcional con fecha de expiración de la suscripción.
- **status**: estado actual de la suscripción (activo, inactivo, expirado).
- **throttling**: período de tiempo mínimo en segundos entre dos notificaciones consecutivas.

2.1.5.3 API de Agentes IoT

En la siguiente sección se explica la API de los Agentes IoT en base a las fuentes [44] [45] [46] [47].

Dentro de una arquitectura FIWARE, los Agentes IoT funcionan como pasamanos de traducción de protocolos utilizados por los sensores, al estándar NGSI. Este proceso de traducción puede ser configurado por el usuario mediante la API disponible para Agentes IoT. La API de Agentes IoT agrupa sus operaciones en dos conjuntos distintos: *Service Group API* y *Device API*.

Service group API

La API de los Agentes IoT permite la creación de *service groups*, los cuales constan de agrupaciones lógicas de los sensores. Estas agrupaciones simplifican la gestión de los dispositivos mediante la configuración de los siguientes parámetros:

- **apikey**. Campo opcional utilizado para validar dispositivos pertenecientes a un *service group*.
- **cbroker**. URI del Context Broker asignado al *service group*.
- **resource**. Ruta dentro de la API del Agente IoT hacia donde los dispositivos del *service group* enviarán datos.
- **entity_type**. Tipo de entidad vinculada.
- **attributes**. Define la traducción de atributos recibidos desde dispositivos del grupo a atributos dentro del modelo de datos de FIWARE.
- **static_attributes**. Define atributos estáticos para los dispositivos representados bajo el *service group*.

La Service group API ofrece una interfaz CRUD para el manejo de *service groups*.

Device API

Los Agentes IoT también ofrecen un mayor nivel de granularidad para la gestión de sensores mediante la *Device API*. A través de ésta los dispositivos pueden ser registrados y configurados en forma específica, en lugar de utilizar la configuración por defecto del

service group. Al igual que para los *service groups*, se expone un conjunto de operaciones CRUD para el manejo de dispositivos. Los campos disponibles son:

- **device_id**. Identificador único del dispositivo dentro del *service group*.
- **entity_name**. Nombre de la entidad.
- **entity_type**. Tipo de entidad vinculada.
- **attributes**. Sobre escribe y define la traducción de atributos recibidos por el dispositivo a atributos dentro del modelo de datos de FIWARE.
- **static_attributes**. Define atributos estáticos para el dispositivo.

2.1.5.4 Modelos de datos de FIWARE

En esta sección se detallan los modelos de datos de FIWARE según las fuentes [48][49].

FIWARE provee un conjunto de modelos de datos específicos a varios dominios de interés. El fin de estos modelos es estandarizar los tipos de datos usados por las aplicaciones inteligentes, especificando cuáles son los atributos que deben tener definidos y bajo qué contexto es pertinente usar cada uno de ellos. El desarrollador tiene control total sobre los mismos y puede extenderlos o modificarlos como precise si estos no se adecúan a sus necesidades. Sin embargo entre los lineamientos se destaca el principio de reutilización, el cual dicta que en lo posible, se intente usar estándares u ontologías ya existentes.

FIWARE organiza sus modelos de datos en distintos dominios de interés, los cuales son: *Alerts, Building, Civic Issue Tracking, Device, Energy, Points Of Interaction, Street Lighting, Transportation*, entre otros.

De los mencionados, se detalla el modelo y dominio *Device*, y el modelo *Vehicle* del dominio *Transportation*.

Device

Los *devices* o dispositivos, son aparatos responsables de una tarea particular (por ejemplo, medir el ambiente, actuar sobre otro sensor, etc). Son objetos tangibles que contienen lógica y son capaces de producir y/o consumir datos, comunicándose a través de la red [50].

Dentro de sus atributos se destacan los siguientes, obtenidos a través de [50]:

- **id**: identificador único.
- **type**: debe ser **Device**.
- **category**: lista de strings identificando las categorías del dispositivo. Dentro de las posibles a seleccionar se tiene **sensor**, **actuator**, **meter**, entre otros.
- **controlledProperty**: lista de strings representando propiedades que puedan ser medidas o controladas. Algunas opciones son **solarRadiation**, **waterPollution**, **trafficFlow**, entre otros.

- **controlledAsset**: la entidad (edificio, objeto, playa) controlada por el dispositivo.
- **location**: locación geográfica del dispositivo representada por una geometría GeoJSON del tipo punto.
- **value**: un valor observado por el dispositivo.

Vehicle

La siguiente información se obtuvo a través de [51] [52]. Los modelos de datos dentro del dominio *Transportation* describen las entidades principales involucradas en aplicaciones inteligentes que intentan resolver problemas de transporte. Se destacan los modelos **TrafficFlowObserved** para representar flujos de tráfico, **Road** para describir calles y **Vehicle** para representar vehículos.

Para el caso de los vehículos, se destacan las siguientes propiedades:

- **id**: identificador único de la entidad.
- **type**: debe ser **Vehicle**.
- **name**: nombre del vehículo.
- **vehicleType**: tipo de vehículo desde el punto de vista estructural. Algunas opciones son **bicycle**, **bus**, **car**, entre otros.
- **category**: lista de strings identificando las categorías del dispositivo. Dentro de las posibles se tiene **public**, **private**, **tracked**, entre otros.
- **heading**: representa la dirección de viaje del vehículo, y se especifica en grados decimales.
- **fleetVehicleId**: identificador del vehículo en el contexto de la flota a la que pertenece.
- **serviceProvided**: lista incluyendo servicios que el vehículo es capaz de proveer. Algunos ejemplos son **urbanTransit**, **streetCleaning**, entre otros.
- **areaServed**: área donde el vehículo brinda servicios.
- **serviceStatus**: estado del vehículo. Los valores permitidos son **parked**, **onRoute**, **broken** y **outOfService**.
- **location**: locación geográfica del dispositivo representada por una geometría GeoJSON del tipo punto.

2.2. Trabajos relacionados

Los trabajos estudiados se buscaron casi en su totalidad a través del portal Timbó¹⁰. El principal método de búsqueda fue a través de palabras clave como IoT, *Smart Cities* y FIWARE. Dentro de los resultados se buscaron ejemplos de trabajos desarrollados en el marco regional, así como internacionales. Además, se tuvo en cuenta que los trabajos a presentar fueran sistemas con distintas características, para enriquecer el análisis

¹⁰<https://foco.timbo.org.uy/home>

posterior de los mismos.

2.2.1. A Data Integration Approach for Smart Cities: The Case of Natal

El artículo describe la implementación de un *middleware* (llamado *Smart Geo Layers*) y el uso del mismo en una aplicación de planeamiento urbano para una iniciativa de *Smart Cities* en Natal, Brasil [6].

El objetivo principal del *middleware* *Smart Geo Layers*, es unificar y normalizar datos de varias fuentes que se encuentran presentes en un entorno de *Smart Cities*. El segundo objetivo es insertar información geoespacial relacionada a los espacios físicos, además de agregar visualización y funcionalidades de análisis de datos.

El primer desafío descrito en el artículo se trata de la normalización de datos de entrada. Debido a que la información proviene de diversos orígenes como son sensores, sistemas, ciudadanos, etc., se define previo a la implementación del *middleware* un modelo de dato compuesto principalmente de las siguientes abstracciones.

- **Layer:** Esta clase representa a un conjunto de entidades georeferenciadas. Se llamará *capa* de aquí en adelante.
- **Entity:** Representa cualquier entidad (sensor, aplicación, ciudadano, etc) que pueda ser o no georeferenciada. Llamada *entidad* de aquí en adelante.

Para la implementación del modelo y los atributos se cuenta con la siguiente tabla, en la que se detalla la clase de cada atributo, su descripción, tipo y se indica si es requerido o no.

Clase	Atributos	Descripción	Tipo	Requerido
Capa y Entidades	<i>name</i>	Nombre de un objeto	String	Sí
	<i>description</i>	Descripción	String	Sí
	<i>properties</i>	Conjunto de propiedades de la capa o entidad	JSON	No
	<i>metadata</i>	Metadatos de capa o propiedad	JSON	No
	<i>draw</i>	Atributos para dibujar en mapas o visualizaciones	JSON	No
Capa	<i>key_path</i>	Clave única para capa	String	Sí
Entidades	<i>geometry</i>	Datos geográficos	GeoJSON	No

Figura 8: Modelo de Datos Caso Natal [6]

Una vez definido el modelo de datos a usar se procede a armar la arquitectura del *middleware* *Smart Geo Layers*

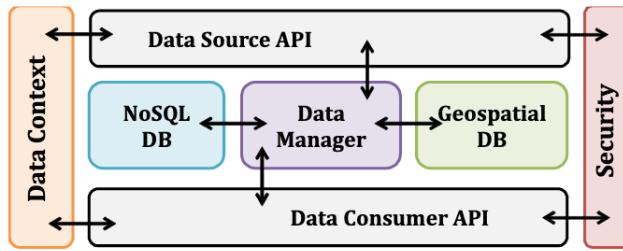


Figura 9: Arquitectura *middleware* Caso Natal [6]

- **Data Source API:** API REST para guardar información relacionada a una *entity* o *layer*.
- **Data Consumer API:** API REST para consumir datos de *Smart Geo Layers*.
- **Data Context:** representa un Context Broker para guardar información de los *layers* registradas en *Smart Geo Layers*.
- **Security:** Componente encargado de seguridad de accesos y autorizaciones para información que pueda ser sensible.
- **Data Manager:** responsable de manejar consultas recibidas por parte de ambas APIs (Data Source y Data Consumer), proveer acceso a las bases de datos y mantenerlas sincronizadas. También se encarga traducir las consultas al lenguaje específico de cada base.
- **NoSQL DB:** Es una base de datos que persiste la información sobre los atributos que no son *geometry*.
- **Geospatial DB:** Es una base de datos que persiste la información de *geometry* de los datos.

Para implementar el *middleware*, los autores se basaron principalmente en los siguientes GEs de FIWARE:

- **Orion Context Broker.**
- **WireCloud:** se trata de un ambiente para crear aplicaciones Web que integran datos de distinto tipo así como lógica de aplicación [38].
- **KeyRock:** es un componente que se encarga de manejar las credenciales y accesos de los usuarios, organizaciones y aplicaciones [40].

Smart Geo Layers ofrece una forma de unificar datos provenientes de distintas fuentes en un entorno *Smart City*. Junto con información geoespacial, el *middleware* ofrece visualización, y funcionalidades de análisis de datos, además de proveer APIs para compartir y consumir datos basados en funciones geográficas.

2.2.2. Designing a FIWARE Cloud Solution for Making your Travel Smoother: the FLIWARE Experience

El artículo describe la estructura, así como algunos aspectos de la implementación de *FLIWARE*. Es una aplicación basada en FIWARE para mejorar la experiencia de usuario al tomar vuelos o hacer uso de algunos servicios que puede ofrecer un aeropuerto.

En el mismo, se detallan los procesos de despliegue de FIWARE como plataforma, así como la implementación de algunos casos de uso básicos de la aplicación [7].

El siguiente diagrama representa los principales componentes de *FLIWARE* y cómo se relacionan entre ellos

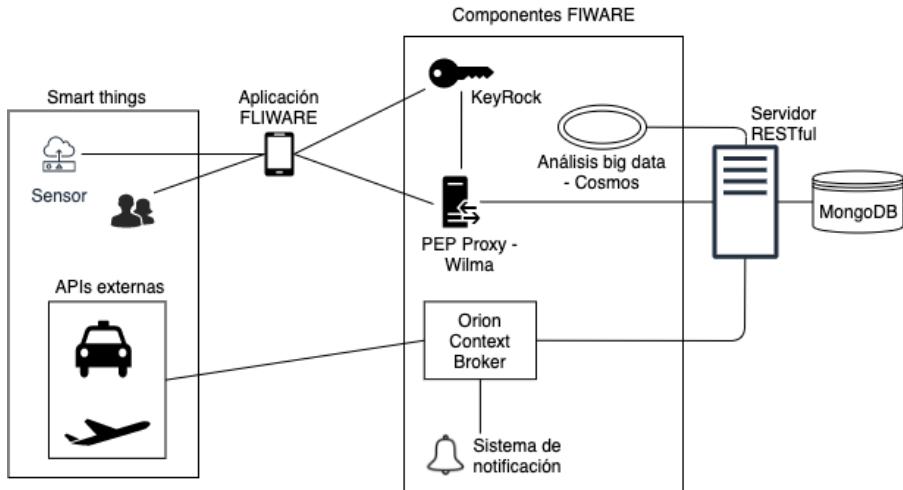


Figura 10: Arquitectura *FLIWARE* [7]

En la primera capa llamada *Smart Things*, se encuentran los usuarios y sensores que reportan a un dispositivo móvil que contiene la aplicación desarrollada. Luego éste se comunica con los componentes referentes a la seguridad, Keyrock [40] y un Policy Enforcement Point (PEP) Proxy Wilma [53], que proveen servicios de autenticación y autorización, similar a lo utilizado por el artículo del trabajo realizado en Natal.

El RESTful server es un servidor de APIs que provee *endpoints* para realizar las funcionalidades de *FLIWARE*. Algunos ejemplos que se describen en el artículo son un *endpoint* para suscribir a un usuario a la información de un vuelo u otro para editar información de un usuario.

La siguiente capa a la que los autores se refieren como *Notification Layer*, representada por “Sistema de Notificación”, se encarga de notificar de cambios a usuarios que están suscritos a distintas fuentes de información proporcionadas por terceros, por ejemplo vuelos y transporte público. La arquitectura de la misma contiene tres componentes fundamentales:

- **Orion Context Broker:** crea un contexto para cada vuelo y transporte y está suscrito para enterarse de cambios en dicha información.
- **OneSignal¹¹:** es un servicio DNS para mandar notificaciones. Cada vez que un usuario se registra, se linkea automáticamente con OneSignal, lo que permite mandar notificaciones cuando hay modificaciones.
- **iotagent-node-lib:** Es una librería NodeJS provista por FIWARE para desarrollar Agentes IoT. Dado que no todas las partes utilizadas en la implementación eran capaces de enviar notificaciones, se desarrolla sobre esta librería para que actuara como interfaz entre Orion y demás software utilizado.

¹¹onesignal.com

Además del caso de uso de suscripción y notificación a datos, los aeropuertos cuentan con aparatos llamados *Beacons* (faroles) que interactúan con la aplicación para dar información sobre la posición del usuario dentro del aeropuerto. Para hacer esto posible, se utiliza el Object Storage GE¹² que se encarga de guardar pequeños pedazos de mapas dentro de contenedores y luego mandarlos a la aplicación, haciendo más sencilla la tarea de asociar los usuarios a una posición dentro del aeropuerto.

Por último, con el fin de proporcionar anuncios personalizados basados en preferencias del usuario, se agrega un componente de inteligencia artificial. Dicho componente, implementado con el GE Cosmos¹³ se encarga de ejecutar un algoritmo de *Machine Learning* con datos de la base MongoDB asociada al Context Broker para aprender sobre el comportamiento de los usuarios y mejorar los anuncios y publicidades presentadas en la aplicación.

2.2.3. Trabajo realizado en la Intendencia de Montevideo

La información sobre el trabajo realizado en la Intendencia de Montevideo fue relevada mediante entrevistas, por lo que no se encuentra en la literatura. Cabe destacar que por motivos de seguridad e integridad, la Intendencia de Montevideo no puede habilitar toda la información de su plataforma, como protocolos de comunicación y detalles de comportamiento de los componentes. Es por esto que la información y los diagramas en esta sección contienen cierto nivel de abstracción.

2.2.3.1 Aplicaciones de la Intendencia de Montevideo

Actualmente, la Intendencia de Montevideo cuenta con varias aplicaciones que hacen uso de la plataforma FIWARE y otras tecnologías relacionadas a *Smart Cities* e IoT. Existen dos categorizaciones de estas aplicaciones. Por un lado se tienen las que se encuentran disponibles al público y por otro se tienen las que se utilizan internamente para el funcionamiento de varios dominios dentro de la ciudad.

Las aplicaciones públicas están desarrolladas sobre una API REST¹⁴ y están disponibles en la aplicación “Montevideo Inteligente”. Algunos ejemplos de las mismas son:

- **Servicio de Playas:** Conjunto de *endpoints* que brinda información básica de las playas, como dirección, bañabilidad (bandera) y habilitación sanitaria.
- **Servicio de Ómnibus:** Conjunto de *endpoints* que permite hacer una variedad de consultas relacionadas a los ómnibus y el estado de los mismos. Algunos ejemplos son consultar qué ómnibus se encuentran cerca de una parada, qué líneas de ómnibus pasan por ella, entre otras.

Como se menciona anteriormente, la Intendencia de Montevideo cuenta con otras aplicaciones que se utilizan internamente para el funcionamiento de varios servicios en la ciudad. Por ejemplo existen aplicaciones de Calidad del Aire, Reclamos y Actividades en la vía pública.

¹²[https://catalogue.fiware.org/enablers/object-storage-ge-fiware- implementation](https://catalogue.fiware.org/enablers/object-storage-ge-fiware-implementation)

¹³<https://fiware-cosmos.readthedocs.io/en/latest/>

¹⁴<https://api.montevideo.gub.uy/docs>

Tanto las aplicaciones privadas como las públicas cuentan con un fuerte componente SIG, dado que a la mayoría de los datos recopilados se les agrega (si no cuentan con ello) datos geoespaciales. Esto permite que sean visualizados en un mapa, utilizados con otros datos o procesados en operaciones geoespaciales.

2.2.3.2 Arquitectura de la plataforma de la Intendencia de Montevideo

En esta sección se pretende mostrar y explicar cómo funciona la plataforma de la Intendencia que soporta los servicios antes mencionados, y cómo se integra FIWARE en la misma. Se tienen dos arquitecturas de plataformas en la Intendencia, una para sensores de ómnibus y otra para los demás sensores de uso general. Dada la gran cantidad de datos relacionados al sistema de transporte público, y que estos son recolectados por las diversas empresas y cooperativas de transporte, es que se hace esta separación.

Arquitectura para sensores generales de la plataforma

En el siguiente esquema se representa la arquitectura que maneja los diversos datos de todos los sensores, con la salvedad de los sensores asociados al transporte. Dado su elevado volumen requieren de una arquitectura dedicada para que los maneje, la cual se detalla en la siguiente sección. El diagrama para esto se muestra en la Figura 11.

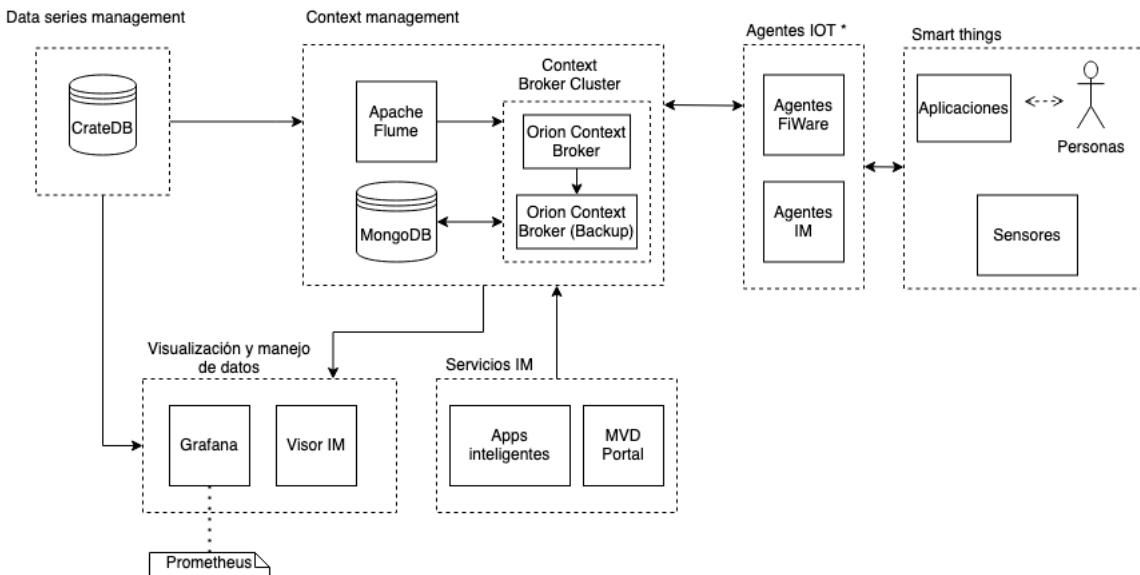


Figura 11: Diagrama de plataforma IM de alto nivel

El flujo comienza desde la derecha, en el componente denominado “*Smart Things*” donde se capturan los datos de interés. Dentro del mismo existen sensores fijos como pueden ser sensores de medición de índice UV, y también existen aplicaciones dedicadas operadas por personas las cuales ingresan manualmente los valores. Estos datos son enviados a través de HTTP a la siguiente capa denominada “Agentes IoT”. En esta capa y según el caso, la Intendencia hace uso de los Agentes IoT de FIWARE así como otros desarrollados por ellos mismos que cubren los distintos protocolos utilizados por los sensores.

Los Agentes IoT entones envían los datos a la siguiente capa denominada “Context management”, y son recibidos por un Context Broker Orion. Este escribe los datos recibidos en una base de datos MongoDB y también los reenvía a otro Context Broker Orion de respaldo, el cual solo contiene la información de los datos de la última semana. Esto le brinda robustez al sistema y lo vuelve más resiliente a las diferentes eventualidades. Los datos de forma interna son manejados por Apache Flume¹⁵, un software dedicado al manejo de datos masivos. Este se encarga de escribir en las bases de datos en lotes, para evitar realizar repetidas consultas cada vez que se reporta un dato nuevo. Este componente es el responsable de enviar los datos a la capa de “Data Series Management” en donde se encuentra una base de datos de series temporales CrateDB¹⁶.

Por último se tienen los clientes, es decir las aplicaciones que hacen uso de estos datos. Si los datos son de carácter privado entonces serán consultados por la capa de “Visualización y manejo de datos”, mientras que si son públicos serán enviados a la capa de Servicios IM. En la primera se encuentra una aplicación Grafana¹⁷, una plataforma que permite visualizar y analizar cantidades masivas de datos en tiempo real y en particular se utiliza un visualizador Prometheus¹⁸. Otro cliente interno que tiene la Intendencia es un visor que desarrollaron para desplegar la información de forma gráfica en el mapa de la ciudad. La segunda capa la componen las diferentes aplicaciones públicas para ciudadanos, así como la API de acceso libre.

Arquitectura para sensores de ómnibus de la plataforma

El siguiente diagrama detalla la arquitectura especial para los datos de ómnibus.

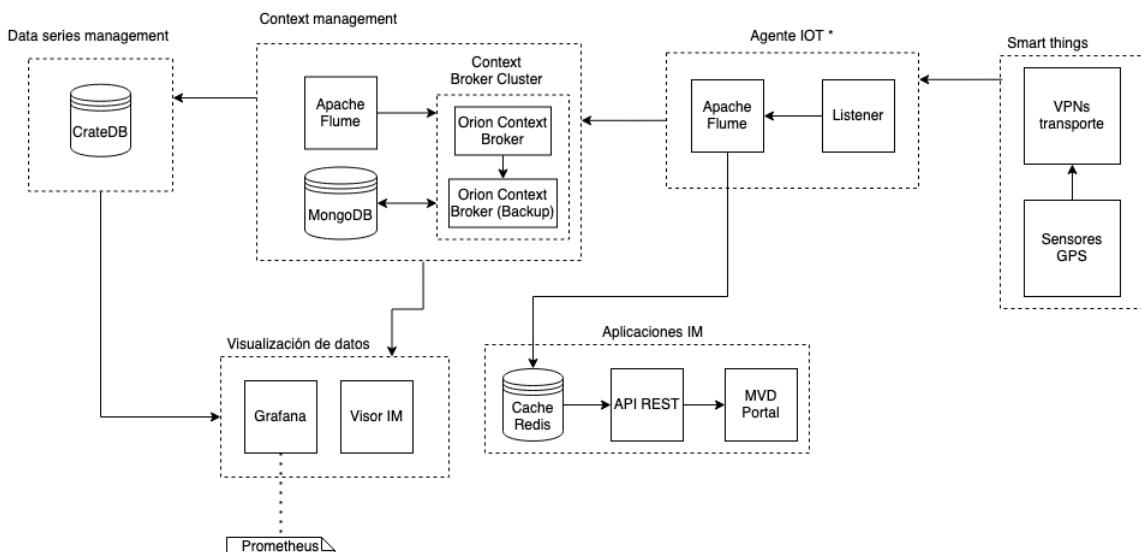


Figura 12: Diagrama de alto nivel para la arquitectura de transporte

Como se menciona anteriormente, los datos de transporte son primero recolectados

¹⁵<https://flume.apache.org/>

¹⁶<https://crate.io/>

¹⁷<https://grafana.com/>

¹⁸<https://prometheus.io/docs/visualization/grafana/>

por las diversas empresas y cooperativas de transporte, que obtienen los datos a partir de sensores de GPS a través de sus redes privadas. Luego de que procesan los datos, son enviados a la Intendencia, que los captura con un componente que denominan “Listener”. El flujo de datos es entonces regulado por una instancia de Apache Flume, la cual relega los mismos a las capas de “Context management”, y “Aplicaciones IM“. Esta ultima es la encargada de implementar las aplicaciones publicas que presentan los datos. Como se observa en la Figura 12, los datos son obtenidos de una base de datos Redis¹⁹ en lugar de ser accedidos directamente de CrateDB o de . Cuando una de estas aplicaciones haga acceso a un dato que no exista en el cache, el dato es obtenido por Apache Flume directamente de CrateDB y se actualiza luego Redis. El resto de la arquitectura es análogo al de la sección anterior.

¹⁹<https://redis.io/>

3. Análisis y solución propuesta

Este capítulo se divide en tres secciones: la sección de análisis, la sección de solución propuesta y la de selección de casos de aplicación para desarrollar el prototipo. En la primera se examinan los trabajos relevados, se identifican prácticas comunes entre ellos y se presentan las tecnologías geoespaciales consideradas. En la sección de solución propuesta se presenta la plataforma generada a través de dicho análisis. En la última sección se describen los casos de aplicación seleccionados para la implementación de los prototipos de aplicación con el fin de validar la plataforma propuesta.

3.1. Análisis

3.1.1. Estudio de trabajos relacionados

Con el objetivo de relevar trabajos desarrollados con FIWARE que incluyeran tecnologías geoespaciales en *Smart Cities*, se leyeron y resumieron algunos artículos regionales e internacionales. Por otro lado, se llevaron a cabo entrevistas con integrantes del departamento de Desarrollo Sustentable en la Intendencia de Montevideo con el fin de conocer más sobre aplicaciones desarrolladas para la ciudad. Esta sección presenta el análisis realizado sobre estos trabajos y las lecciones aprendidas para el diseño de la solución propuesta.

3.1.1.1 Intendencia de Montevideo

La arquitectura de la plataforma de la Intendencia de Montevideo es en su esencia una plataforma FIWARE, que tiene la particularidad de tener un componente encargado del manejo de datos temporales. Asimismo, la plataforma realiza la mayor parte del procesamiento de los datos geoespaciales a nivel de capa de aplicación.

Por ejemplo, dentro de las aplicaciones privadas, la Intendencia posee una aplicación que brinda estadísticas sobre los diversos reclamos que se realizan en la ciudad. Dentro de sus funcionalidades ofrece mapas de calor, gráficas del historial del volumen de reclamos por zona, entre otras. Todas estas se encuentran implementadas a nivel de la capa de “Visualización y manejo de datos”, y por tanto no se puede reutilizar la lógica que las implementa en la capa de “Servicios IM”. Es decir que para realizar gráficos o mapas de calor como los descriptos sobre otros conjuntos de datos en la parte pública de la plataforma, el código que los crea debe ser implementado de nuevo o extraído de donde está a algún componente global.

Otra aplicación que se encuentra actualmente en desarrollo, cuenta el aforo de una playa a través del procesamiento de imágenes recolectadas con un dron. Dado que esta aplicación no está terminada se desconoce su arquitectura, pero en caso de estar implementada de la misma manera que el ejemplo anterior, esta viviría en la capa “Servicios IM”. Esto quiere decir que la lógica de reconocimiento de personas también estaría dentro de esta capa. Este enfoque puede presentar ciertas desventajas dentro de las cuales se destacan la dificultad para escalar y mantener el sistema. Si la Intendencia quisiera

saber qué plazas son las más concurridas y cuál es su aforo para una aplicación interna, se debería extraer este componente o implementarlo de nuevo.

Como solución a esta desventaja, se podría agregar un centro de procesamiento de datos en el componente de “Context Management”, el cual no solo sería responsable de realizar dichos cálculos sino también de almacenarlos de forma centralizada. Este nuevo componente no tiene por qué ser muy complejo o específico, ya que eso iría en contra de su reutilización. Sin embargo, debería presentar módulos reutilizables que puedan servir como piezas de construcción genéricas de las aplicaciones.

3.1.1.2 Trabajos internacionales

Se desarrolla a continuación un resumen sobre los aspectos relevantes de los trabajos internacionales y las lecciones aprendidas sobre los mismos.

Trabajo	Tipo de sistema	Aspectos relevantes para el análisis
Trabajo en Natal	Middleware	Normalización de datos
		Base de datos geoespaciales separada
		Arquitectura modular
		Uso de APIs
Aplicación FLIWARE	FLIWARE	Presentación en capas
		Arquitectura modular
		Uso de APIs

Tabla 1: Comparación sobre trabajos relacionados internacionales.

Sobre los datos y componentes SIG: La normalización de datos presenta la ventaja de que es relativamente sencillo agregar nuevas fuentes de información dado que para su integración basta con traducir los datos al modelo desarrollado. Para el *middleware* Smart Geo Layers, se explica que la separación de las bases de datos se debe a que las consultas geográficas son más pesadas que las no geográficas, y al tratarse de una API que realiza ambas, se ven beneficios al tenerlas separadas. Sin embargo, la separación presenta algunas desventajas. En primer lugar, podrían perder la sincronización por fallas u otras razones. La segunda desventaja que se encuentra presente son los posibles problemas de *performance* al tener que hacer múltiples pedidos para poder persistir datos, en caso de que se utilice un único tipo de consulta. En el caso de la aplicación FLIWARE los datos geoespaciales se encuentran en la capa de *location* provista por los *Beacons* como se explica en el Capítulo 2. En este caso no se hacen consultas geoespaciales complejas. Para conocer la ubicación actual se consulta por el *beacon* asociado al dispositivo (el que se encuentre más cerca), y en conjunto con el componente de *machine learning* se muestran además publicidades personalizadas según la ubicación y preferencias del usuario.

Componentes separados y presentación en capas: La arquitectura modular y presentación en capas en ambos trabajos presenta la gran ventaja de que cada componente puede ser reemplazado o cambiado con relativa facilidad ya que estos se encuentran claramente segregados. Por otro lado, cada componente y capa tiene responsabilidades claras en cada caso. De esta forma, el código se vuelve más mantenable al ser posible el

trabajo en paralelo en cada uno de los componentes y se facilita la tarea de detectar los eventuales errores.

Sobre las interfaces: El uso de APIs ayuda a la mantenibilidad y hace que sea fácil extender los casos de uso para información que pueda provenir de nuevas fuentes (sea APIs de terceros u otra). También permite la fácil inserción de nuevos componentes con nuevas responsabilidades simplemente usando la forma de comunicación ya existente.

3.1.2. Análisis sobre componentes SIG

En esta sección se presenta el análisis realizado para diseñar la integración de tecnologías geoespaciales en la plataforma. Es de interés que las mismas cumplan los siguientes criterios:

- Seguir estándares OGC
- Tener bajo acoplamiento y alta cohesión
- Evitar la duplicidad de datos
- Ser fácilmente extensibles y reutilizables

A su vez las tecnologías geoespaciales deben llevar a cabo las tareas de almacenamiento, recuperación y modificación de datos, así como permitir realizar operaciones basadas en estos.

Para realizar la tarea de recuperación de la información, se decide utilizar una implementación de OGC API Features. Con dicha implementación se centraliza la información, logrando así evitar posibles problemas de mantenimiento como duplicidad, o falta de sincronización. Por otro lado, OGC API Features ofrece los datos a partir de *endpoints*, lo que hace que la información sea reutilizable por otros componentes. Por último, agregar nuevas *features* es sencillo, ya que el estándar dicta que éstas estarán disponibles en nuevos *endpoints* independientes de los ya existentes.

Otro candidato que cumple con los puntos anteriores es el estándar WFS, pero se decide a favor de OGC API Features por diversos motivos. En primer lugar toda la familia de estándares OGC API, y en particular Features, utiliza el patrón REST. Este patrón tiene como principal ventaja su alta aceptación y utilización en los sistemas actuales de la industria. FIWARE es un ejemplo de esto, ya que utiliza REST para implementar todos sus servicios. Por lo tanto, optar por OGC API Features favorece a la consistencia a lo largo de la plataforma ya que esto implica que las comunicaciones entre componentes sigan el mismo mecanismo. WFS en cambio ofrece sus servicios por medio de HTTP o SOAP²⁰. En segundo lugar, WFS utiliza para el intercambio de datos el formato XML en lugar de JSON. JSON es utilizado por los estándares OGC API, y supone una ventaja ya que es uno de los formatos más amigables y utilizados en la comunidad. En tercer lugar, en un sistema que implemente OGC API Features se pueden consultar los metadatos y las capacidades de forma intuitiva a través de páginas HTML. Con WFS dichas capacidades se exponen en un *endpoint* específico en formato XML, resultando menos amigable para un usuario. Finalmente, la comunidad OGC se encuentra más activa en lo que respecta a los nuevos estándares.

²⁰<https://www.w3.org/TR/soap12-part1/>

Por otra parte las tareas de almacenamiento, modificación y operación de datos se llevan a cabo utilizando una implementación del estándar OGC API Processes. Mediante su uso se centraliza el procesamiento de datos, teniendo así una única fuente de código que los manipula y logra hacer operaciones geoespaciales con éstos. OGC API Processes, al igual que OGC API Features, es fácil de reutilizar y de extender debido a que presenta sus servicios mediante *endpoints*. Se decide por OGC API Processes antes que WPS, por los mismos motivos que se opta por OGC API Features antes que WFS.

3.2. Solución propuesta

Dentro de esta sección se describen aspectos sobre la arquitectura y el diseño de la plataforma propuesta. Se busca que la arquitectura sea lo suficientemente genérica para que se emplee en cualquier dominio de las *Smart Cities*. A su vez se desea que los componentes utilizados sean fácilmente extensibles y reutilizables. Finalmente se busca que en la plataforma prime la simplicidad y minimalidad, evitando el *overhead* innecesario en el flujo de datos.

3.2.1. Arquitectura

En la Figura 13 se presenta un esquema a alto nivel de la arquitectura diseñada para la solución propuesta.

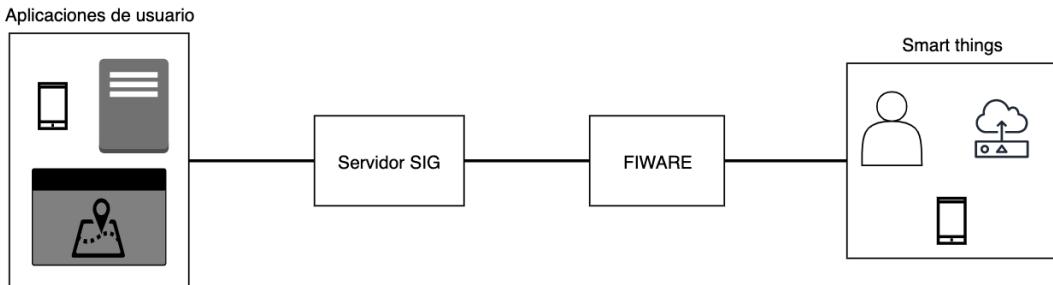


Figura 13: Arquitectura Propuesta

Como se puede observar en el diagrama la arquitectura se compone de cuatro componentes principales, que ordenados del nivel más bajo al más alto son: la capa de *Smart Things*, un componente FIWARE, un servidor SIG y la capa de aplicaciones de usuario.

En primera instancia, la arquitectura general es similar a las observadas en trabajos relacionados, ya que se encuentran en su mayoría las mismas capas vistas en esa sección. La principal diferencia en el diagrama general radica en la presencia de un servidor SIG. La razón de ser de este servidor es actuar como una instancia para procesar datos geoespaciales provenientes de FIWARE y ponerlos a disposición de posibles aplicaciones que puedan hacer uso de los mismos.

A continuación se explican con más detalle cada uno de los componentes presentados.

3.2.1.1 Smart things

En el nivel más bajo de la arquitectura se encuentra la capa de *Smart Things* o “cosas inteligentes”, compuesta principalmente por dispositivos, sensores, personas, sistemas, entre otros. Son la fuente de información principal para los sistemas IoT, y cumplen la característica de ser altamente heterogéneos. Esto significa que el mecanismo de comunicación puede variar significativamente de uno a otro, provocando la necesidad de tener herramientas que funcionen como intermediarios entre estos elementos y los sistemas.

En la arquitectura planteada, estos objetos se comunican únicamente con el componente FIWARE mediante las APIs descritas en el capítulo anterior.

3.2.1.2 FIWARE

El siguiente componente en la arquitectura es clave para transformar la solución en una plataforma inteligente con capacidades IoT. La capa FIWARE es la encargada de conectarse con los objetos inteligentes, de recibir los datos de contexto que ellos emiten y de ejecutar las acciones necesarias como consecuencia de la información obtenida.

De los variados componentes que ofrece, para la solución propuesta se toman únicamente los dos más esenciales: Context Broker y Agentes IoT.

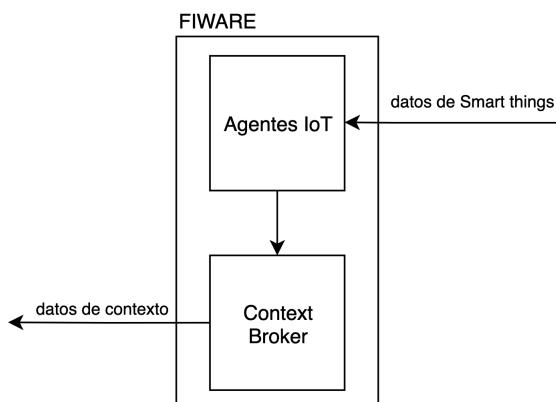


Figura 14: Componente FIWARE

Por un lado los Agentes IoT funcionan como *middleware* ya que se encargan de traducir los datos recibidos por los sensores a un formato estandarizado que luego el Context Broker entienda. Además, mediante sus interfaces, permiten la creación de *service groups* y de nuevos dispositivos en la base de datos de FIWARE, necesarios para dar de alta nuevos sensores a nivel global en la plataforma.

Por otro lado, el Context Broker es el componente que no puede faltar en un sistema basado en FIWARE. En primera instancia ofrece mediante la API NGSI v2 la posibilidad de crear suscripciones ante cambios de datos de contexto. Además, se encarga de recibir los datos traducidos por los agentes, actualizarlos y en caso de existir una suscripción sobre esa entidad, generar la notificación correspondiente.

3.2.1.3 Servidor SIG

El servidor SIG es el componente que engloba todas las funcionalidades geoespaciales de la plataforma. Consiste en un servidor que usa en su totalidad estándares OGC API, más específicamente OGC API Features y OGC API Processes. Su función principal es la de recibir pedidos de *features* geográficas, así como de la ejecución de procesos geoespaciales.

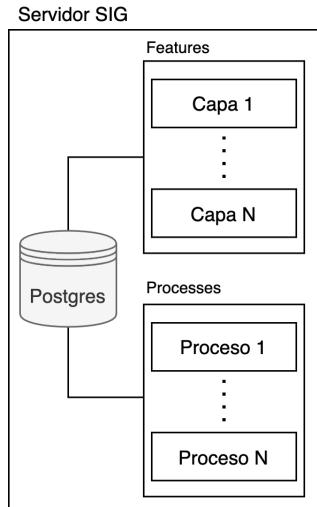


Figura 15: Componente servidor SIG

Este componente se conforma de tres subcomponentes:

- **Servicio Features.** Porción lógica del servidor SIG que implementa el estándar OGC API Features. Se encarga de exponer capas de *features* geográficas en forma de servicios Web REST.
- **Servicio Processes.** Componente del servidor que implementa el estándar OGC API Processes. Expone mediante servicios REST la capacidad de invocar geoprocessos previamente definidos.
- **Base de datos Postgres con extensión para datos geográficos.** Porción encargada de la persistencia de los datos geográficos, así como de ofrecer primitivas básicas para el cálculo de procesamiento de este tipo de datos.

La inclusión de esta capa implica tener un único centro global de procesamiento de datos geoespaciales, definido a nivel de servicios. Esto se diferencia del trabajo realizado en la Intendencia de Montevideo, donde este tipo de procesamiento se realiza del lado de las aplicaciones. Este componente por un lado favorece la escalabilidad y la mantenibilidad de la plataforma, ya que para extender las capacidades del servidor se requiere únicamente definir un proceso o capa nueva sin afectar los ya existentes. Por otro lado la inclusión de este componente facilita la reutilización, ya que permite desarrollar módulos lógicos compartidos que se puedan utilizar transversalmente por los distintos servicios definidos.

3.2.1.4 Aplicaciones de usuario

Las aplicaciones de usuario representan la capa de más alto nivel, y pueden ser cualquier sistema que haga uso de los servicios provistos por el servidor SIG. Pueden comprender desde aplicaciones Web y móvil, hasta APIs y sistemas lógicos externos que requieran de los datos de la plataforma.

3.2.2. Diseño

En esta sección se presentan una serie de diagramas que detallan cómo se efectúa la comunicación entre los distintos componentes de la arquitectura para los escenarios típicos de la plataforma.

Siguiendo un orden cronológico en las comunicaciones efectuadas entre componentes, en primera instancia se requiere hacer una configuración inicial sobre las entidades de FIWARE. La misma consiste en tres pasos: crear un *service group*, crear un *device* y crear una suscripción.

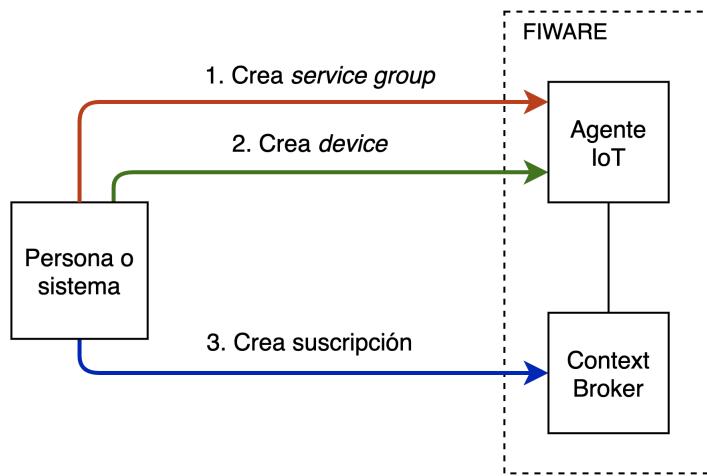


Figura 16: Inicialización en FIWARE

Estos conceptos, introducidos y detallados en la sección de FIWARE del marco teórico, se integran en la Figura 16, que explicita el orden de las invocaciones y las entidades involucradas.

Como primer paso, se debe crear un *service group* mediante la Service Group API. Éste tiene como funciones principales inicializar la configuración para la comunicación entre agente y Context Broker, agrupar un conjunto de sensores bajo un mismo tipo, levantar nuevos servicios sobre los cuales los dispositivos pertenecientes a este grupo reportarán datos, entre otros. Una vez hecho este paso, ya se pueden comenzar a registrar nuevos dispositivos en la plataforma. La configuración puede variar mínimamente según el caso de aplicación. Por ejemplo, si se quisiera dar de alta un sensor de ómnibus, en este paso se crea un nuevo *service group* indicando la URL del Context Broker, el tipo de entidad (podría ser *Vehicle*), y una clave para el grupo.

El segundo paso se basa en crear un nuevo dispositivo en FIWARE por medio de la Device API. Para la creación se requiere ingresar un identificador del sensor, así

como un nombre y un tipo que coincida con el registrado al crear el *service group*. También se especifica qué atributos estáticos y dinámicos reportará el sensor, así como el mapeo del nombre del atributo recibido por el sensor al nombre utilizado internamente por FIWARE. Este paso se realiza una vez por cada dispositivo emisor de datos en la plataforma. Siguiendo con el ejemplo anterior, el identificador podría ser el del sensor GPS a utilizar, un atributo estático podría ser la línea que representa el sensor, y un atributo dinámico podría ser la ubicación, que será el valor a reportar más adelante.

Por último, se deben crear una o más suscripciones utilizando la API NGSI v2. Estas suscripciones funcionan de manera que, al ser reportados nuevos datos sobre un determinado tipo de sensor, se ejecutan notificaciones hacia URLs previamente configuradas. Esto permite definir patrones sobre los datos de forma que solamente los nuevos datos de contexto que cumplan dicho patrón generen notificaciones sobre el objetivo. En particular para la arquitectura propuesta, las suscripciones permiten que ante nuevos datos de sensores reportados se desencadenen notificaciones que ejecuten procesos en el servidor SIG. Por ejemplo, si se quisiera enviar una notificación a un servicio para conocer la posición en tiempo real de un ómnibus, se puede crear una suscripción sobre el atributo dinámico “ubicación” que envíe los datos al servicio deseado.

Los primeros dos pasos se efectúan contra el Agente IoT, mientras que el último se realiza directamente sobre el Context Broker.

Una vez hecha la inicialización, el sistema ya se encuentra listo para recibir datos de los distintos emisores. La Figura 17 detalla el flujo de comunicación para este caso.

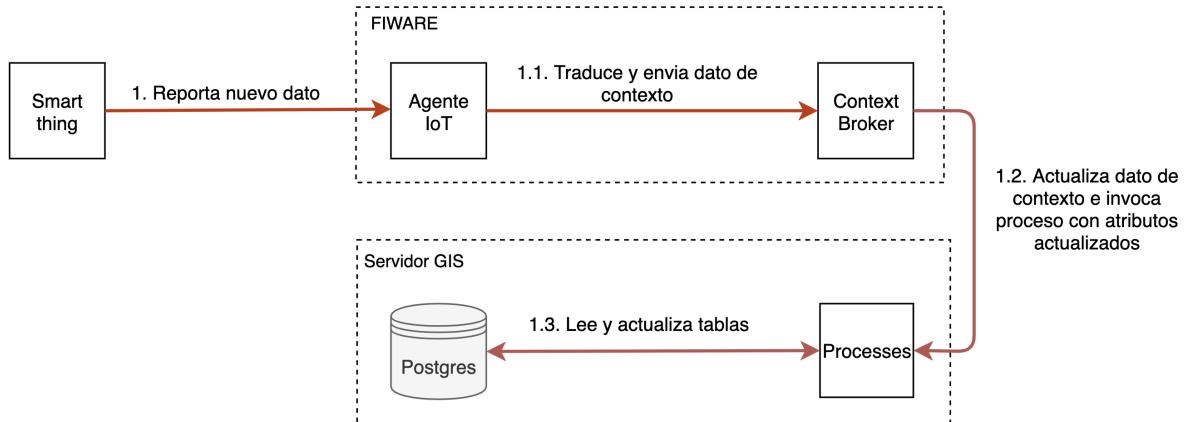


Figura 17: Comunicación entre componentes para nuevo dato reportado

Luego de que el *Smart Thing* genera el dato, el Agente IoT lo recibe. Éste componente realiza la traducción del dato recibido por medio del protocolo IoT en cuestión, al formato definido previamente en la inicialización. Luego el agente envía los datos traducidos al Context Broker, quien recibe el pedido y actualiza en su base de datos local los valores para el dispositivo. Por último, a causa de la suscripción creada en el diagrama anterior, el Context Broker desencadena la notificación correspondiente invocando a uno de los procesos del servicio API Processes. Dicho proceso realiza las lecturas y escrituras requeridas en la base de datos Postgres. En el caso del ejemplo anterior, el nuevo dato sería una nueva ubicación, y el proceso podría ser un servicio que reporta ubicaciones en tiempo real o guarda un histórico de los datos.

Por último, interesa representar el flujo de comunicación entre las aplicaciones de usuario y el servidor SIG.

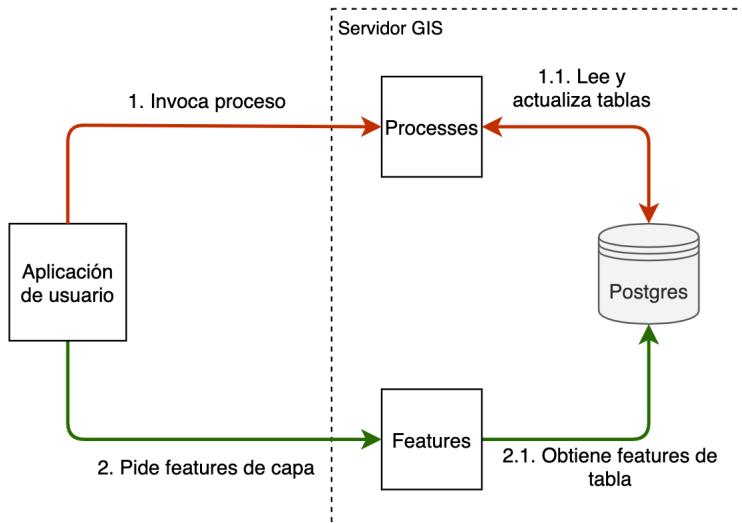


Figura 18: Comunicación entre Aplicaciones de usuario y servidor SIG

Como se observa en la Figura 18, las aplicaciones de usuario pueden efectuar dos operaciones distintas con el servidor SIG.

La primera es la capacidad de invocar procesos sobre el servicio *Processes*. Éste lee de la base los datos necesarios para ejecutar el proceso, y de ser necesario realiza las actualizaciones correspondientes.

La segunda es la posibilidad de acceder a distintas capas de *features* mediante el servicio *Features*. Una vez que la aplicación hace el pedido, el servicio obtiene los datos necesarios de la base de datos. Siguiendo el ejemplo anterior, la aplicación podría ser un servicio que consuma y muestre las posiciones de un ómnibus en tiempo real o permita hacer consultas históricas sobre su recorrido.

3.3. Casos de aplicación seleccionados para desarrollar el prototipo

El principal objetivo de los casos de aplicación es validar la plataforma introduciendo casos que requieran operaciones SIG específicas no provistas por FIWARE por defecto. El objetivo secundario para la selección es elegir aplicaciones no implementadas en Montevideo, pero que puedan utilizar datos y sensores que ya se encuentran disponibles. Finalmente, gracias al relevamiento del trabajo realizado por la Intendencia de Montevideo y la forma en la que dividen el uso de su plataforma según el tipo de sensor, se decide pensar en dos casos de aplicación. El primero basado en sensores móviles (i.e. que cambian su ubicación en un período corto de tiempo) y el segundo en sensores fijos. Además de lo anterior, pensar en sensores móviles y fijos implica seleccionar distintos modelos de datos provistos por FIWARE. Esto permite explorar distintos dominios para implementar los casos de aplicación.

Como consecuencia surge la aplicación de desvíos para sensores móviles, que identifica si un ómnibus se desvía de su recorrido establecido. Por otro lado, al pensar en sensores fijos y cómo combinar los datos de los mismos para entregar un mayor valor al usuario, se llega a la funcionalidad de *ranking* de playas. La aplicación para este caso permite obtener un *ranking* de playas según las preferencias del usuario, utilizando la información disponible proveniente de los sensores. Con respecto a los modelos de datos de FIWARE utilizados, en el caso de la aplicación de desvíos de ómnibus se utiliza *Vehicle* para los sensores de GPS. Por otro lado, para el caso de la aplicación de *ranking* se utiliza el modelo *Device* para modelar los distintos sensores en una playa.

4. Implementación de la plataforma

En este capítulo se explican los detalles de implementación de los componentes presentados en el capítulo anterior. Con el fin de guiar la explicación, se utiliza como ejemplo un ómnibus que reporta su ubicación a través de un sensor GPS.

4.1. Configuración FIWARE

Se presentan las acciones de creación de sensores y cómo se reportan los datos al componente FIWARE para el ejemplo de ómnibus reportando datos en el siguiente diagrama.

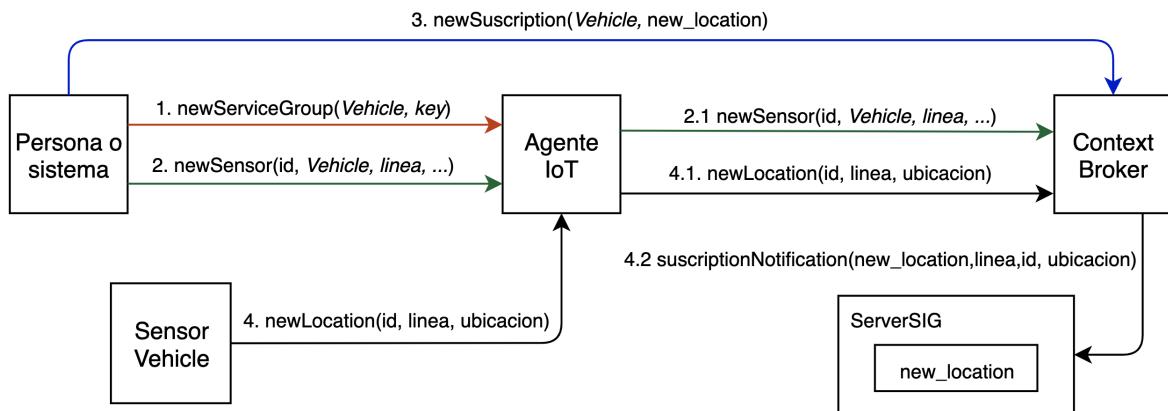


Figura 19: Flujos necesarios para que un sensor de ómnibus reporte posiciones

En esta sección se explica la configuración del componente de FIWARE para que la plataforma se encuentre lista para recibir datos. En el caso del ejemplo guía, el sensor reportará datos que siguen el modelo *Vehicle* de FIWARE. Como ilustra la Figura 19, se deben crear un *service group*, un sensor *Vehicle* y una suscripción. Para el desarrollo de la plataforma y de aquí en adelante, se asume el uso de la implementación Orion del Context Broker por ser la más documentada y utilizada en la comunidad.

4.1.1. Service group

El primer paso es crear un *service group* para el tipo de sensor sobre el cual se desean reportar datos.

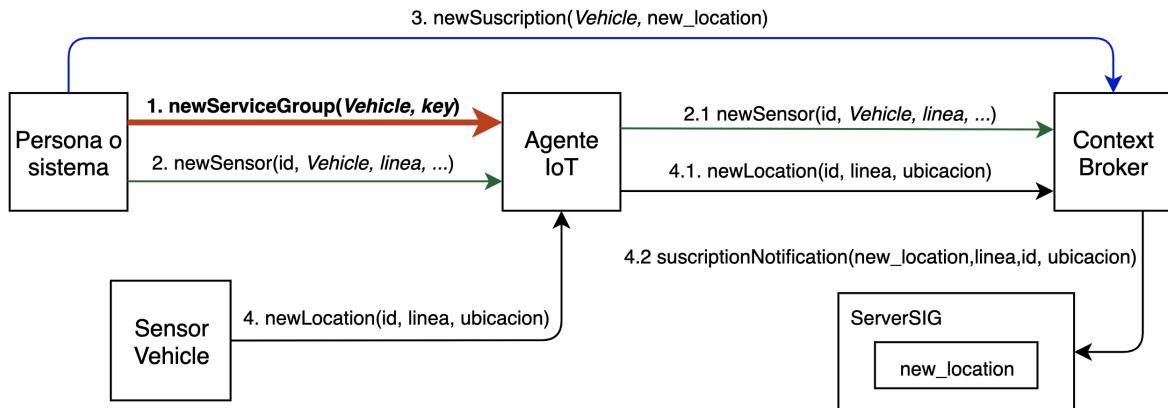


Figura 20: Flujo para la creación de un *service group*

Un *service group* define un *endpoint* al que un conjunto determinado de sensores enviarán datos. Esto permite gestionar sensores de forma diferenciada según su tipo, permitiendo que dispositivos que miden variables distintas reporten datos a *endpoints* diferentes dentro del Agente IoT, los que a su vez pueden redirigir los datos a distintos Context Brokers. Por ejemplo, puede ser deseable que sensores de tipo *Vehicle* reporten a un Context Broker, mientras que sensores de otro tipo reporten a uno diferente.

Para el ejemplo en cuestión, el *payload* se vería como el siguiente:

```

services: [
  {
    apikey: "a0b093d6d433",
    cbroker: http://contextBroker:1026,
    entity_type: "Vehicle",
    resource: "/iot/json"
  }
]

```

Código 6: Payload para la creación del Service Group

En este caso se está definiendo un nuevo servicio sobre la ruta `/iot/json` para reportar datos de sensores de tipo *Vehicle*. A su vez se define una clave bajo el campo `apikey` usada para autenticarse a dicho servicio. Esta clave se envía como parámetro en la ruta de los pedidos de los sensores. Por ejemplo, los sensores de ómnibus que se crean bajo este grupo realizarán pedidos a la siguiente url: `http://agenteIoT:7896/iot/json ?k=a0b093d6d433`. Finalmente, en el payload de creación del Service Group, se especifica la URL del Context Broker al que se redirigirán los datos.

4.1.2. Alta de sensor

El siguiente paso es crear el sensor en el sistema de FIWARE para poder reportar datos en el futuro.

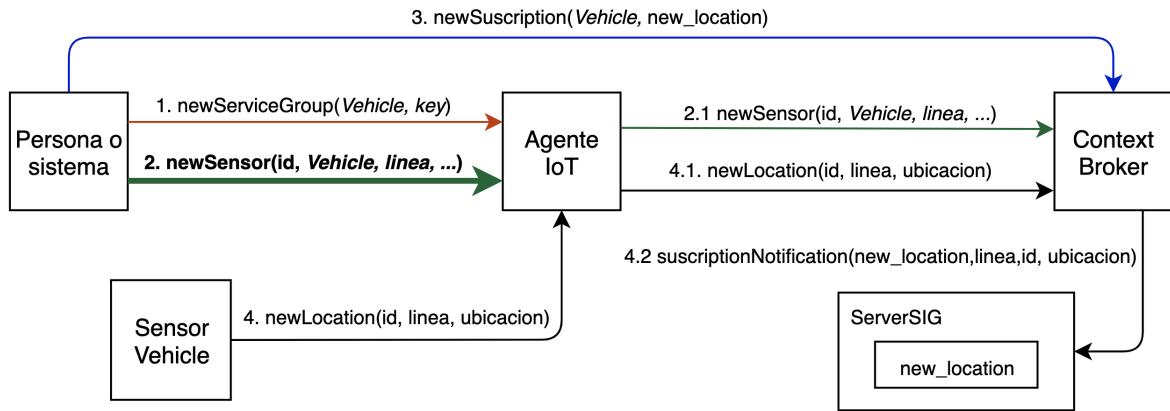


Figura 21: Flujo para la creación de un sensor

La creación del sensor en el Agente IoT permite informarle qué datos son los que van a estar siendo reportados, y también permite agregar información extra relativa al sensor. Siguiendo el ejemplo, se presenta a continuación el *payload* necesario para crear el sensor en el agente.

```

{
    device_id: "Bus1",
    entity_name: "urn:ngsi-ld:Bus:1",
    entity_type: "Vehicle",
    static_attributes: [
        { name: "vehicleType", type: "Text", value: "bus" },
        { name: "category", type: "Text", value: "public" },
        { name: "heading", type: "Text", value: "Parque Rodó" },
        { name: "fleetVehicleId", type: "Text", value: "405" },
        { name: "serviceProvided", type: "Text", value: "urbanTransit" },
        { name: "areaServed", type: "Text", value: "Montevideo" },
        { name: "serviceStatus", type: "Text", value: "onRoute" }
    ],
    attributes: [
        { object_id: "location", name: "location", type: "geo:point" }
    ]
}

```

Código 7: Payload para la creación del sensor

Los primeros campos representan el identificador y nombre que se utilizarán en

el Context Broker para el dispositivo. Los campos especificados en `static_attributes` dependen del tipo de entidad con la que se esté trabajando. En este caso los valores se definen suponiendo que el ómnibus en cuestión pertenece a la linea 405 y tiene como destino Parque Rodó en la ciudad de Montevideo. Por último, en `attributes` se definen los campos variables que reportará el sensor. En este caso basta con agregar la localización bajo el nombre `location`.

4.1.3. Suscripción FIWARE

Para que las nuevas lecturas reportadas sean procesadas por el servidor SIG, es necesario crear una suscripción personalizada que considere los datos necesarios a reportar, así como la información de los procesos del servidor SIG a los que se quiere invocar. Este flujo está representado por el punto 3 del diagrama. El diagrama de la figura ilustra que, a diferencia de los pasos anteriores donde las operaciones se realizan utilizando las APIs del agente, esta configuración debe realizarse invocando la NGSI API del Context Broker.

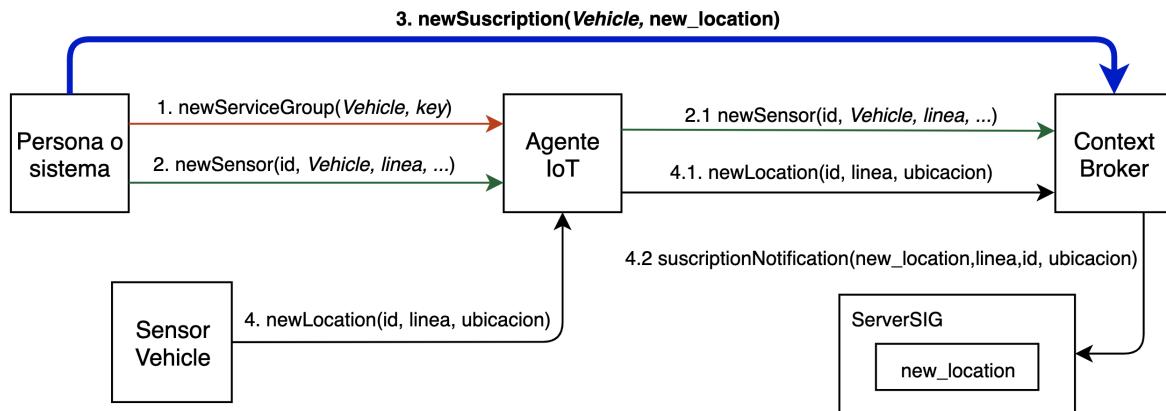


Figura 22: Flujo para la creación de una suscripción

El *payload* de la suscripción a crearse es el siguiente.

```
{
  description: "Notify of changes in a bus location
    value",
  subject: {
    entities: [{ "idPattern": "Bus.*",
      "type": "Vehicle" }],
    condition: {
      attrs: ["location"]
    }
  },
  notification: {
    httpCustom: {
      url: "http://serverSIG:8080/processes/new_location
        /jobs",
    }
  }
}
```

```

    headers: {
        "Content-Type": "application/json"
    },
    method: "POST",
    payload: { "%7B%20%22location%22%3A%20${location}%20%7D" }
},
attrs: ["fleetVehicleId", "heading"],
attrsFormat: "keyValues",
metadata: ["dateCreated", "dateModified"]
},
throttling: 0
}

```

Código 8: Payload para la creación de una suscripción

Se puede observar que la suscripción se divide en cuatro claves: `description`, `subject`, `notification` y `throttling`.

En la primera clave el campo a definir es `description`, que se trata de un *string* que contiene una descripción de los datos que se reportarán con la suscripción. Por ejemplo, una descripción puede ser “Cambios de ubicación para vehículos”.

La clave `subject` corresponde a un objeto JSON el cual tendrá toda la información sobre qué entidades van a estar disparando las notificaciones de esta suscripción. En el ejemplo de guía, se quiere hacer una suscripción solamente para los sensores del tipo vehículo que sean ómnibus. Para lograr esto se deben definir los campos `type` y `idPattern`. En el campo `type` se especifica el tipo de sensor que generará notificaciones, que en este caso es *Vehicle*. Por otro lado el campo `idPattern` toma una expresión regular, la cual es útil para acotar el espacio de sensores del tipo *Vehicle*. En este ejemplo como se quieren los ómnibus se busca que el patrón `Bus.*` esté presente dentro del identificador del sensor. Por esto mismo la creación de los sensores de la sección anterior deben tener en cuenta qué patrones se usarán en `idPattern` para incluirlos en los identificadores. Finalmente con el campo `condition` se enumeran los atributos del sensor para los cuales se enviará una notificación en caso de ser modificados o creados. En el ejemplo, solamente se incluye `location` ya que solo interesa notificar al servidor GIS de los cambios en la posición del sensor.

La tercera clave del objeto de la suscripción es `notification` y en ella se incluye toda la información pertinente de la notificación. Dado que se está utilizando una notificación personalizada, se debe agregar dentro de la clave `httpCustom` el método HTTP y la dirección a la que el Context Broker deben realizar los pedidos. El primer dato a definir es la URL a la que se enviará la notificación. Como muestra la Figura 22, en el ejemplo guía se desea que ante nuevas ubicaciones del sensor el Context Broker haga un pedido al proceso *new_location* del servidor SIG. Por esto mismo la URL es `http://serverSIG:8080/processes/new_location/jobs`. En `headers` y `methods` se indica que `headers` y método HTTP utilizará el Context Broker en el momento de realizar una notificación.

Por otro lado, se asigna el contenido a enviar en el campo `payload` de `notification`.

En el ejemplo se envía un objeto JSON como indica el *header*. Este JSON se escribe como un *string* y debe estar codificado con el mismo protocolo con el que se codifican las URLs (ver caracteres reservados en RFC URI²¹). Cualquiera de los atributos del sensor se pueden incluir en cualquier parte de este JSON, y se deben incluir como \${atributo}. Los atributos del sensor que se envíen que no forman parte de **condition** deben ser agregados dentro del campo **attrs**. Asimismo se pueden incluir metadatos en el JSON, en cuyo caso éstos deben estar declarados también en el campo **metadata**. En el ejemplo se muestra un mensaje simple que representa el objeto JSON {“location”: \${location}}.

La cuarta y última clave es **throttling**, que admite un número entero. Este valor representa la mínima cantidad de tiempo en segundos que debe existir entre dos notificaciones consecutivas. Por ejemplo en caso de ser 1, esto significa que luego de enviarse una notificación se espera un segundo hasta que se habilite la emisión de una nueva. Los intentos de notificación que ocurran antes del segundo, se ignoran y no se emiten. Este campo es útil cuando se quiere evitar saturar al objetivo de la suscripción. En el presente ejemplo el **throttling** vale 0, lo que significa que se omite esta ventana de tiempo y las notificaciones se emiten ni bien los vehículos reportan sus valores de localización.

4.2. Servidor SIG

En esta sección se cubren los detalles de implementación del servidor SIG de la plataforma.

Para el desarrollo de este componente, se decide utilizar la herramienta Pygeoapi como implementación de los estándares OGC API Features y OGC API Processes. Pygeoapi es un servidor de código libre construido completamente en Python. Cuenta con actividad reciente en su repositorio y es utilizado en ambientes de producción en diversos sistemas existentes. Por ejemplo, actualmente está siendo utilizado por el servicio meteorológico canadiense [54][55].

Utilizar una única aplicación que implementa ambos estándares tiene como ventaja que el modulo SIG se encuentra homogeneizado. Es decir que se tienen ambas implementaciones (*Features* y *Processes*) en el mismo lenguaje de programación, en la misma instancia de servidor y con los mismos patrones de diseño.

La documentación Pygeoapi²² estipula que para inicializar el servidor se debe configurar un archivo de formato YAML que contiene cuatro secciones: **server**, **logging**, **metadata** y **resources**. Siguiendo el ejemplo presentado en la sección anterior, el archivo tendría una configuración similar a la siguiente:

```
server:  
  bind:  
    host: 0.0.0.0  
    port: 8080  
  url: http://localhost:8080  
  mimetype: application/json; charset=UTF-8  
  (...)
```

²¹<https://tools.ietf.org/html/rfc3986>

²²<https://docs.pygeoapi.io/en/stable/configuration.html>

```
logging:
    level: ERROR

metadata:
    identification:
        title: Servidor SIG
        description: servidor que expone capas de datos y
                      procesos geoespaciales
    (...)

resources:
    recorridos:
        type: collection
        title: Recorridos
        description: recorridos de líneas de ómnibus de
                      Montevideo
    (...)

    providers:
        - type: feature
          name: PostgreSQL

new_location:
    type: process
    processor:
        name: new_location
```

Código 9: YAML de configuración de Pygeoapi

Dentro de la sección de `server` se debe incluir toda la configuración del servidor, como dirección y puertos en la que se escuchan pedidos, codificación, el tipo de datos manejado por el servidor, etc. Luego en la sección de `logging` se configura la ruta del archivo en donde se escriben los *logs*, y la naturaleza de los mismos (sólo errores, sólo información, etc). Dentro de `metadata` se define la información del servidor como nombre, naturaleza de los datos, datos de contacto de quienes lo mantienen, entre otros. Por último, en `resources` se listan los recursos que provee el servicio. En este campo se debe listar los procesos que implementa el servidor, así como las capas de *features* (denominadas *collections* en el estándar API Features) que se exponen. En las siguientes secciones se explica cómo se agregan estos servicios al servidor.

4.2.1. Features

Las *features* expuestas por OGC API Features se mapean directamente con tablas disponibles en una base de datos. Por lo que el primer paso para disponer de dichas *features* es cargar las tablas con la información que corresponda, o dejar su esquema vacío en caso de que no se cargue información pre-existente en la base de datos. Siguiendo el ejemplo de las secciones anteriores, se podrían cargar datos sobre los recorridos y paradas

de ómnibus.



Figura 23: Carga de datos y *features* expuestas

Una vez cargadas las tablas necesarias se puede proceder a exponer las *features* como recursos. Como se menciona anteriormente Pygeoapi requiere que para exponer dichas capas es necesario modificar el archivo YAML de configuración. Si se quisiera exponer una capa que represente las paradas de ómnibus, se debería agregar la siguiente entrada dentro de `resources`:

```

recorridos:
  type: collection
  title: recorridos
  description: Recorridos de ómnibus en Montevideo
  keywords:
    - recorridos
  links:
    - type: text/html
      rel: canonical
      title: information
      (...)

  extents:
    spatial:
      bbox: [-56.43, -34.70, -56.02, -34.94]
      crs: http://www.opengis.net/def/crs/OGC/1.3/
          CRS84
      (...)

  providers:
    - type: feature
      name: PostgreSQL
      data:
        host: localhost
        port: 5432
        (...)

      id_field: ID
      table: Recorridos
      geom_field: geom
  
```

Código 10: Configuración para agregar los recorridos como una capa

Toda la información se define bajo el nombre de la *collection*, que en este caso es `recorridos`. Esto representa con qué nombre queda expuesta la capa. Los siguientes

campos a completar se tratan de información correspondiente a la *collection*, como puede ser su nombre, descripción, palabras clave, etc. La información bajo la clave **spatial** es necesaria para definir el *bounding box*²³ en el que se deben encontrar los datos, y además se puede especificar el sistema de referencia utilizado. Por defecto la plataforma utiliza el sistema CRS84 ya que facilita el uso de GeoJSON en otros componentes.

En la sección de **providers** se indica la información de la base de datos en donde se encuentra la tabla que se interesa exponer. Para esto se requiere agregar información pertinente para acceder a la misma como se indica en el ejemplo. Por último, es necesario indicar el nombre de la tabla, qué atributo se utiliza como identificador en la misma y qué atributo corresponde a la geometría de las entidades que guarda. En el ejemplo se expone la tabla *Recorridos*, cuyo identificador es ID y la geometría se representa con la columna **geom** que en este caso es del tipo LineString.

4.2.2. Procesos

Para implementar nuevos procesos es necesario agregar un nuevo archivo en la carpeta `pygeoapi/process` que herede de la clase `BaseProcessor` definida por defecto por Pygeoapi. Para asegurarse de tener los campos necesarios y que el proceso sea ejecutado correctamente, lo más conveniente es utilizar el esqueleto del proceso que se incluye por defecto llamado `hello_world.py`. En el ejemplo guía, se precisa un proceso que almacene la nueva localización del ómnibus, la cual llega gracias a la suscripción personalizada configurada en la sección anterior. Para implementar el proceso se debe agregar un nuevo archivo `new_location.py` con el siguiente contenido:

```
import logging

from pygeoapi.process.base import BaseProcessor

LOGGER = logging.getLogger(__name__)

# Definición de entradas y salidas del proceso
PROCESS_METADATA = { ... }

# Clase que hereda de BaseProcessor
class NewLocationProcessor(BaseProcessor):
    # Inicialización del proceso
    def __init__(self, provider_def):
        BaseProcessor.__init__(self, provider_def,
                              PROCESS_METADATA)

    # Lógica del proceso
    def execute(self, data):
        (...)

        return outputs
```

²³Un *bounding box* es un área representada por un rectángulo, definida por dos pares de coordenadas.

```
# Cómo se expone el proceso
def __repr__(self):
    return "<NewLocationProcessor> {}".format(
        self.name)
```

Código 11: Esqueleto del proceso new_location

Dentro de este archivo, se debe primero definir las entradas del proceso en la constante `PROCESS_METADATA`. Pygeoapi utilizará esta constante para validar las entradas al proceso, brindando una capa de seguridad sobre el sistema. Los campos que se deben definir en esta constante para luego exponer y utilizar correctamente el proceso son `"id"`, `"inputs"` y `"outputs"` que corresponden al identificador del proceso y la especificación de entrada y salida respectivamente.

Las entradas a los procesos de Pygeoapi deben ser objetos JSON. Siguiendo el ejemplo, en la creación de la suscripción se envía la localización del ómnibus en un objeto JSON bajo la clave `"location"`. El valor del campo `"inputs"` en la constante `PROCESS_METADATA` del proceso `new_location` sería el siguiente:

```
"inputs": [{
    "id": "location",
    "title": "location",
    "input": {
        "literalDataDomain": {
            "dataType": "string"
        }
    },
    "minOccurs": 1,
    "maxOccurs": 1
}],
```

Código 12: Ejemplo de declaración de las entradas del proceso new_location

La primera definición es el campo `id`, que debe coincidir con lo especificado en la suscripción de FIWARE de la sección anterior para que los valores se puedan validar correctamente. Luego se pueden definir distintos tipos de datos en `"dataType"`. Como el Context Broker enviará los datos en formato `string`, se utiliza este valor para indicar el tipo de la entrada. Por último, con `"minOccurs"` y `"maxOccurs"` se define la cantidad de veces que debe encontrarse el input en el `payload` con el que se llama al proceso. Por ejemplo si ambos son 1, la entrada debe aparecer una vez.

El siguiente paso es implementar la lógica propia del proceso que se ejecutará cuando éste sea invocado. Para esto se debe implementar la función `execute(self, data)` que recibe los datos enviados en la variable `data`. El siguiente extracto de código ejemplifica la implementación de la función `new_location` la cual inserta la localización obtenida en la base de datos y la retorna para indicar que ejecutó correctamente.

```
def execute(self, data):
    outputs = [
        "id": "ubicacion",
        "value": data["location"]]
```

```
    }]
    insert_location_in_database(data["location"])

    return outputs
```

Código 13: Función new_location

Por último, es necesario exponer el nuevo proceso para lo cual se lo debe agregar a la lista de procesos en el archivo `pygeoapi/plugin.py` dentro del objeto "process". Siguiendo el ejemplo anterior, esta sección del código se vería de la siguiente forma:

```
(...) ,
"process": {
    "NewLocation": "
        pygeoapi.process.new_location.NewLocationProcessor
    "
}
```

Código 14: Exposición del proceso en el archivo plugin.py

La clave dentro de "process" corresponde al nombre del proceso, que se utilizará más adelante. Luego el valor indica la ruta de dirección de la clase que implementa el proceso. La misma se compone del elemento más general al más específico. Primero se navega al módulo `process` de `pygeoapi`, se accede al archivo `new_location` y se indica que la clase es `NewLocationProcessor`.

Finalizando, se expone en el YAML de configuración el proceso de ejemplo de la siguiente manera:

```
new_location:
    type: process
    processor:
        name: NewLocation
```

Código 15: Configuración para la adición del proceso

El valor de la clave `name` dentro de `processor` debe coincidir con el nombre definido en el archivo `plugin.py`.

4.2.3. Módulos de ayuda implementados para procesos

En el curso del proyecto se implementaron dos módulos de ayuda para las operaciones con datos geoespaciales y operaciones CRUD con la base de datos. Esto se hace para cumplir con el bajo acoplamiento y alta cohesión en el código. Para usarlas se deben importar al inicio del archivo correspondiente al proceso.

```
from . import postgis_helper
from . import db_helper

# Ejemplo de uso
```

```
(...)  
postgis_helper.geom_from_geojson(location)
```

Código 16: Ejemplo de importación de los módulos

A continuación se incluyen las definiciones de las funciones disponibles para cada módulo.

4.2.3.1 Módulo SIG

Las funciones se implementan haciendo consultas a una base de datos con la extensión PostGIS²⁴ compartida entre los *endpoints* para procesos y los *endpoints* para *features*.

- `geom_from_geojson(location)`: Devuelve la geometría a partir de las coordenadas pasadas como parámetro. Para el cálculo se utilizan las operaciones `geometry ST_GeomFromGeoJSON(text geomjson)` y `text ST_AsText(geometry g1)` de PostGIS.
- `intersects_predicate(geom1, geom2)`: Devuelve un booleano que indica si las geometrías se intersectan. Se utiliza la operación `boolean ST_Intersects(geometry geomA, geometry geomB)` de PostGIS.
- `get_geom(tableName, whereClause)`: Devuelve la geometría, para una entidad que se busca con la cláusula *where* pasada como parámetro. Se utiliza la operación `text ST_AsText(geometry g1)` de PostGIS.
- `buffer(geom, bufferRadius)`: Devuelve la geometría resultado de hacer un *buffer*²⁵ de diámetro *bufferRadius* sobre la geometría indicada. Se utilizan las operaciones `geometry ST_Buffer(geometry g1, float radius_of_buffer)` y `text ST_AsText(geometry g1)` de PostGIS.
- `distance(geometry g1, geometry g2)`: Devuelve la distancia entre las geometrías *g1* y *g2*. Se utiliza la operación `float ST_Distance(geometry g1, geometry g2)` de PostGIS.

Todas las operaciones de PostGIS utilizadas para la implementación del módulo se encuentran explicadas en detalle en la documentación oficial de PostGIS [56].

4.2.3.2 Módulo DB

Por otro lado, se implementa el módulo DB para concentrar las funciones relacionadas a operaciones con la base de datos, fuera de los datos geoespaciales. El módulo cuenta con las siguientes funciones.

- `create_new_record(columnArray, tableName, attributesArray)`: Inserta un nuevo registro a la tabla de nombre *tableName* con los valores *attributesArray* para las columnas *columnArray*.

²⁴<https://postgis.net/>

²⁵Devuelve todos los puntos cuya distancia a la geometría es menor o igual a la indicada

- **update_record(tableName, setClause, whereClause)**: Actualiza un registro de la tabla de nombre *tableName* que se encuentra con la cláusula *whereClause* con lo indicado por la cláusula *setClause*.
- **is_new_record(whereClause, tableName)**: Devuelve un booleano que indica si el registro que se puede encontrar en la tabla de nombre *tableName* con la cláusula *whereClause* ya existe.
- **select_fields_from_table(tableName, whereClause, fields)**: Devuelve los campos indicados con *fields* para uno o más registros encontrados con la cláusula *whereClause* en la tabla de nombre *tableName*.

Todas las funciones se implementan ejecutando una consulta en lenguaje SQL²⁶ en la base PosgreSQL del servidor SIG.

²⁶<https://www.postgresql.org/docs/9.3/sql.html>

5. Aplicación de gestión de sensores

En este capítulo se presentan aspectos sobre las funcionalidades, arquitectura, diseño e implementación de la aplicación de gestión de sensores.

5.1. Introducción

El propósito de esta aplicación es proveerle a un usuario administrador una herramienta para manejar los sensores que se pueden encontrar en un sistema, así como formas de visualizar información relevante de los mismos. También se encarga de la configuración inicial relacionada a la creación de *service groups*, suscripciones y demás para que los usuarios no deban hacerlo de forma manual.

El segundo rol que cumple este sistema es el de simular datos. Si bien los datos simulados son compatibles con los que se podrían encontrar en servicios abiertos, en el marco de los prototipos y con el fin de validar la plataforma, las lecturas son simuladas.

A continuación se muestran detalles sobre las funcionalidades, arquitectura, diseño e implementación de la misma.

5.2. Funcionalidades

La aplicación de gestión se trata de una aplicación Web con un mapa que permite visualizar los sensores existentes.

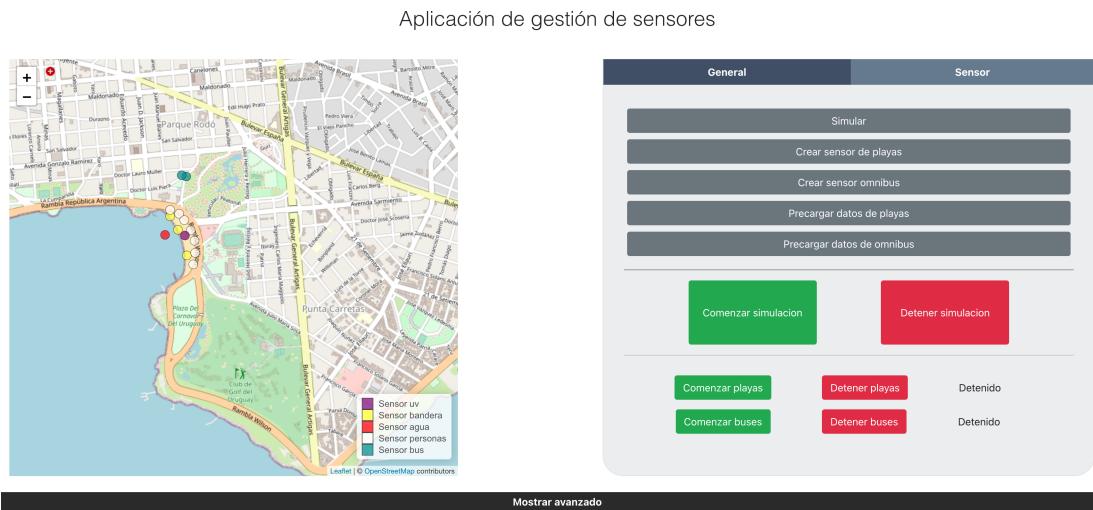


Figura 24: Aplicación de gestión de sensores

Además de lo anterior, la aplicación cuenta con algunas funciones básicas para crear, editar y borrar sensores que se describen a continuación.

5.2.1. Crear sensores

Para crear sensores existen dos posibilidades, se pueden crear manualmente o pueden ser creados utilizando la funcionalidad de precarga, que se encarga de crear múltiples sensores de los tipos disponibles. En el marco del proyecto hay dos disponibles: el sensor de playa y el sensor de ómnibus.

En caso de crear un sensor de playa manualmente se dispone de las siguientes opciones.



Figura 25: Aplicación de gestión de sensores - Creación de sensor de playa

Para el caso de un sensor de ómnibus únicamente se da la opción de marcar si el sensor estará activo y qué conjunto de datos usar para simular el desvío

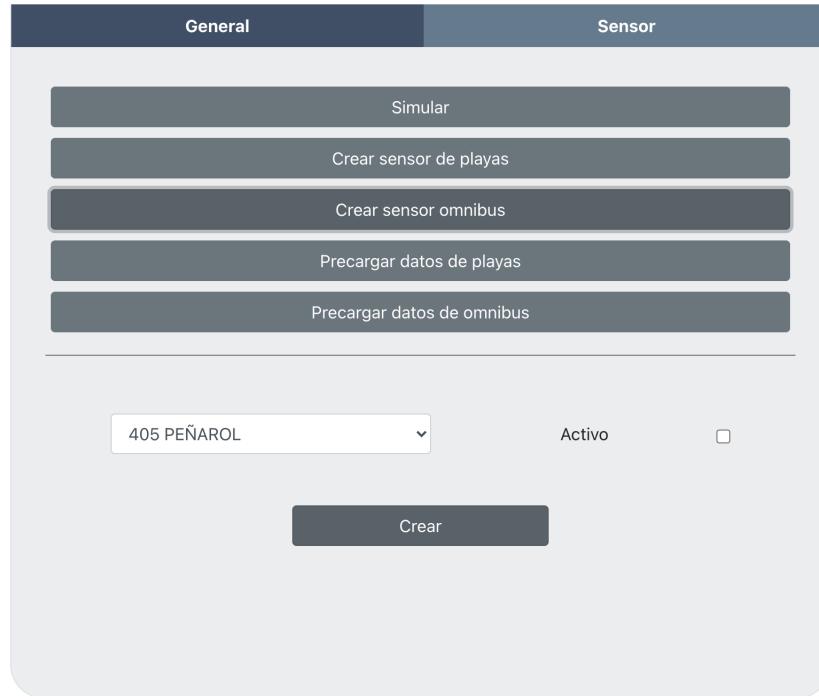


Figura 26: Aplicación de gestión de sensores - Creación de sensor de ómnibus

5.2.2. Editar sensores

Para editar campos sobre sensores, se puede seleccionar uno en el mapa lo que provoca el despliegue del panel mostrado en la Figura 27.

The screenshot shows a detailed edit form for a sensor. It has two tabs at the top: 'General' and 'Sensor'. The 'Sensor' tab is active. The form contains several input fields: 'ID:' with value '37', 'Tipo de sensor:' with value 'uv', 'Minimo:' with value '1' in a dropdown, 'Maximo:' with value '11' in a dropdown, 'Activo:' with a checked checkbox, 'Fijo:' with an unchecked checkbox, and 'Valor Fijo:' with an empty input field. At the bottom is a large dark grey button labeled 'Actualizar'.

Figura 27: Aplicación de gestión de sensores - Edición de sensor

Para el caso de los sensores de playa se pueden editar los valores de “Mínimo”, “Máximo”, “Activo”, “Fijo” y “Valor Fijo”, los cuales serán descritos más adelante en la sección de implementación. En el caso de los sensores de ómnibus solamente se puede editar si el sensor se encuentra activo o no.

5.2.3. Simular datos

Para simular datos se puede elegir producir datos para ambos tipos de sensores o para cada uno individualmente, como se muestra en la Figura 28. El algoritmo de simulación también esta descrito en la sección de implementación del presente capítulo.

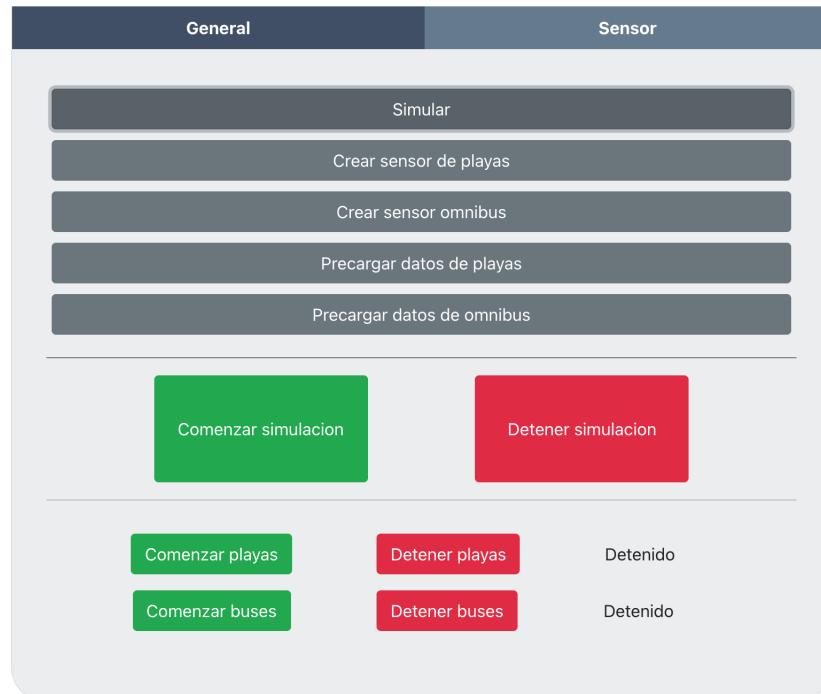
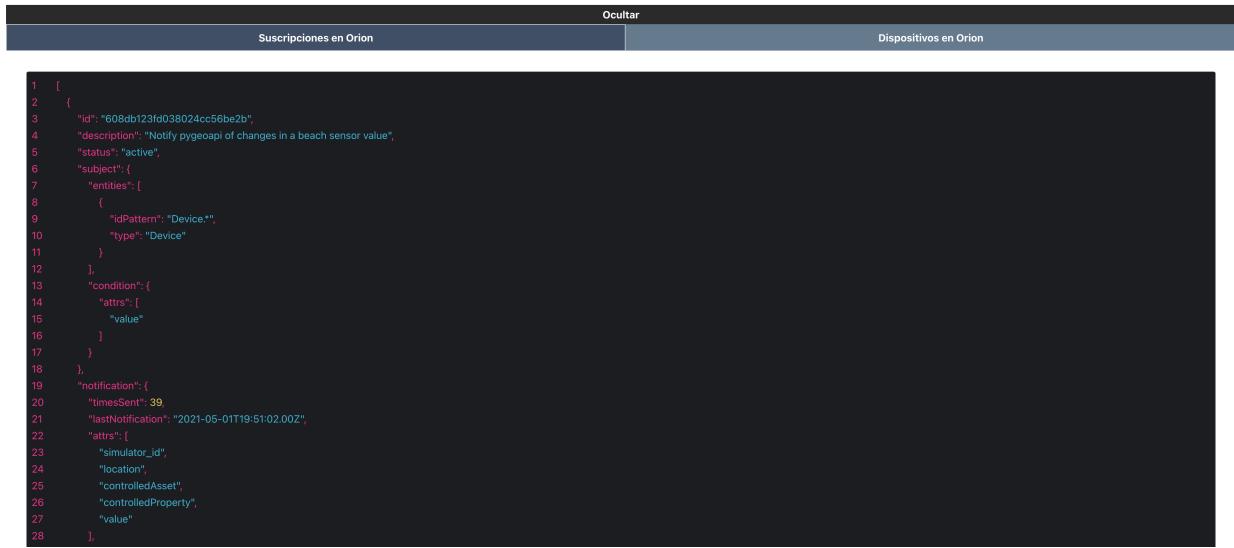


Figura 28: Plataforma Sensores - Simulación de datos

5.2.4. Uso avanzado

Existe la posibilidad de consultar por la suscripción y dispositivos disponibles en el Context Broker. Esto se muestra al seleccionar el botón de “Mostrar avanzado”, y la respuesta consiste en objetos JSON que representan los sensores existentes.

En la primera pestaña se muestran las suscripciones disponibles, con todos los datos correspondientes.



The screenshot shows a web interface with two tabs at the top: "Suscripciones en Orion" (left) and "Dispositivos en Orion" (right). The "Suscripciones en Orion" tab is active. Below the tabs, there is a dark panel containing a JSON-like code snippet representing a subscription. The code includes fields such as "id", "description", "status", "subject", "entities", "condition", "notification", "timesSent", "lastNotification", and "attrs". The "attrs" field lists attributes like "simulator_id", "location", "controlledAsset", "controlledProperty", and "value".

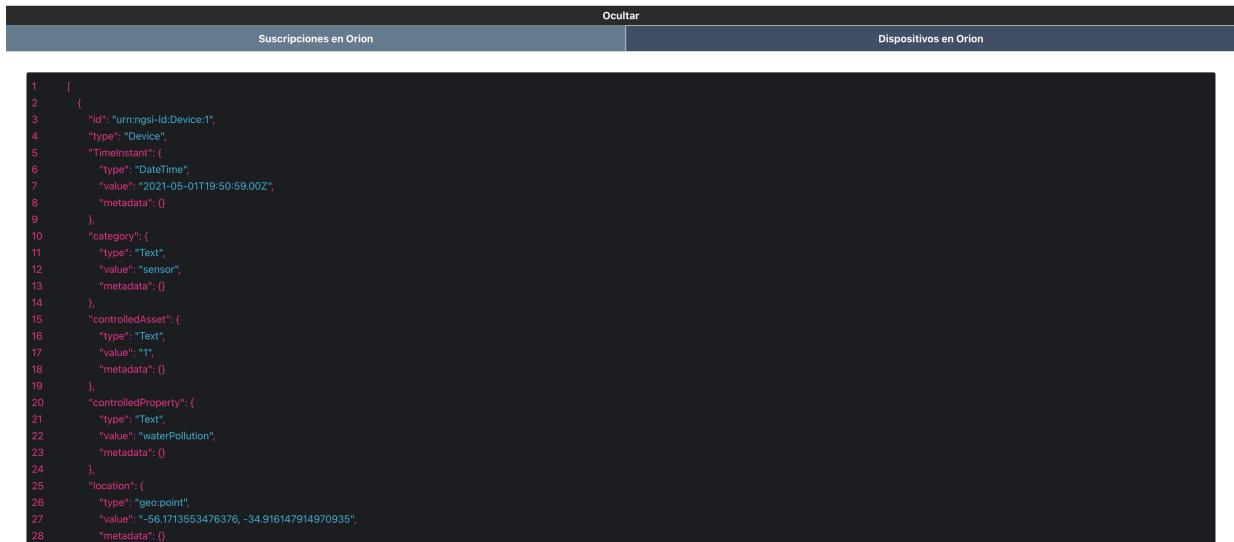
```

1 [
2   {
3     "id": "608db123fd038024cc56be2b",
4     "description": "Notify pygeoapi of changes in a beach sensor value",
5     "status": "active",
6     "subject": {
7       "entities": [
8         {
9           "idPattern": "Device",
10          "type": "Device"
11        }
12      ],
13      "condition": {
14        "attrs": [
15          "value"
16        ]
17      }
18    },
19    "notification": {
20      "timesSent": 39,
21      "lastNotification": "2021-05-01T19:51:02.00Z",
22      "attrs": [
23        "simulator_id",
24        "location",
25        "controlledAsset",
26        "controlledProperty",
27        "value"
28      ],
29    }
30  }
31 ]

```

Figura 29: Aplicación de gestión de sensores - Suscripciones al Context Broker

En la segunda pestaña se encuentra disponible la lista de dispositivos que tiene el Context Broker con sus respectivos atributos.



The screenshot shows a web interface with two tabs at the top: "Suscripciones en Orion" (left) and "Dispositivos en Orion" (right). The "Dispositivos en Orion" tab is active. Below the tabs, there is a dark panel containing a JSON-like code snippet representing a device. The code includes fields such as "id", "type", "Timeinstant", "category", "controlledAsset", "controlledProperty", and "location". The "category" field has a "value" of "sensor". The "controlledAsset" field has a "value" of "1". The "controlledProperty" field has a "value" of "waterPollution". The "location" field has a "value" of "-56.1713553476376, -34.916147914970935".

```

1 [
2   {
3     "id": "urmgssi-Id:Device:1",
4     "type": "Device",
5     "Timeinstant": {
6       "type": "DateTime",
7       "value": "2021-05-01T19:50:59.00Z",
8       "metadata": {}
9     },
10    "category": {
11      "type": "Text",
12      "value": "sensor",
13      "metadata": {}
14    },
15    "controlledAsset": {
16      "type": "Text",
17      "value": "1",
18      "metadata": {}
19    },
20    "controlledProperty": {
21      "type": "Text",
22      "value": "waterPollution",
23      "metadata": {}
24    },
25    "location": {
26      "type": "geopoint",
27      "value": "-56.1713553476376, -34.916147914970935",
28      "metadata": {}
29    }
30  }
31 ]

```

Figura 30: Aplicación de gestión de sensores - Sensores en el Context Broker

5.3. Arquitectura

La aplicación de gestión de sensores se encuentra compuesta por una aplicación Web *frontend* y un servidor *backend* dónde se encuentra la mayor parte de la lógica de la aplicación.

El esquema de la Figura 31 muestra las tecnologías utilizadas para cada parte, y cómo se relacionan entre ellas.

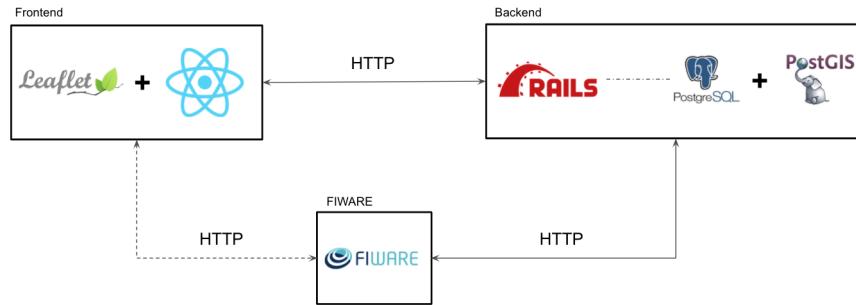


Figura 31: Arquitectura + Tecnologías de aplicación de gestión de sensores

El servidor *backend* provee los *endpoints* necesarios para manejar las operaciones CRUD de los sensores, y se encarga de la simulación de datos. También se encarga de las comunicaciones con los Agentes IoT y el Context Broker para dar de alta sensores y suscripciones, registrar nuevas medidas y obtener información sobre ellos.

Para el desarrollo del servidor se utiliza Ruby on Rails²⁷ que se basa en el patrón MVC (Model View Controller) [57]. Es por esto que los pedidos al servidor se atienden de manera sencilla a través de los controladores, y los accesos a la base de datos se realizan a través de la interfaz de los modelos. Además, resulta fácil agregar nuevos modelos y controladores para nuevos casos de aplicación, ya que existen comandos que crean todas las partes pertinentes automáticamente.

Además de lo anterior, se cuenta con una base de datos utilizada principalmente para la lógica de simulación. La misma cuenta con la extensión PostGIS²⁸ para poder almacenar datos geográficos.

El servidor *frontend* se encarga de mostrar los sensores creados y provee la interfaz para el resto de las operaciones CRUD de los sensores. Asimismo, para las funcionalidades dentro de “Mostrar avanzado”, el *frontend* también se comunica con algunos componentes de FIWARE para obtener datos sobre los dispositivos, entre otros.

Para el desarrollo del *frontend* se utiliza React²⁹ y en particular la biblioteca Leaflet³⁰ para mostrar el mapa y los sensores en el mismo.

5.4. Diseño

La comunicación para la mayoría de los casos de uso provistos por la aplicación se inicializa del lado del componente Web, que hace un pedido a un *endpoint* provisto por el servidor *backend*.

A continuación se muestran los diagramas de comunicación para los casos de uso de esta aplicación.

²⁷<https://rubyonrails.org/>

²⁸<https://postgis.net/>

²⁹<https://es.reactjs.org/>

³⁰<https://leafletjs.com/>

5.4.1. Creación de sensores

El diagrama de la Figura 32 corresponde a la comunicación entre los componentes relevantes al crear un sensor nuevo.

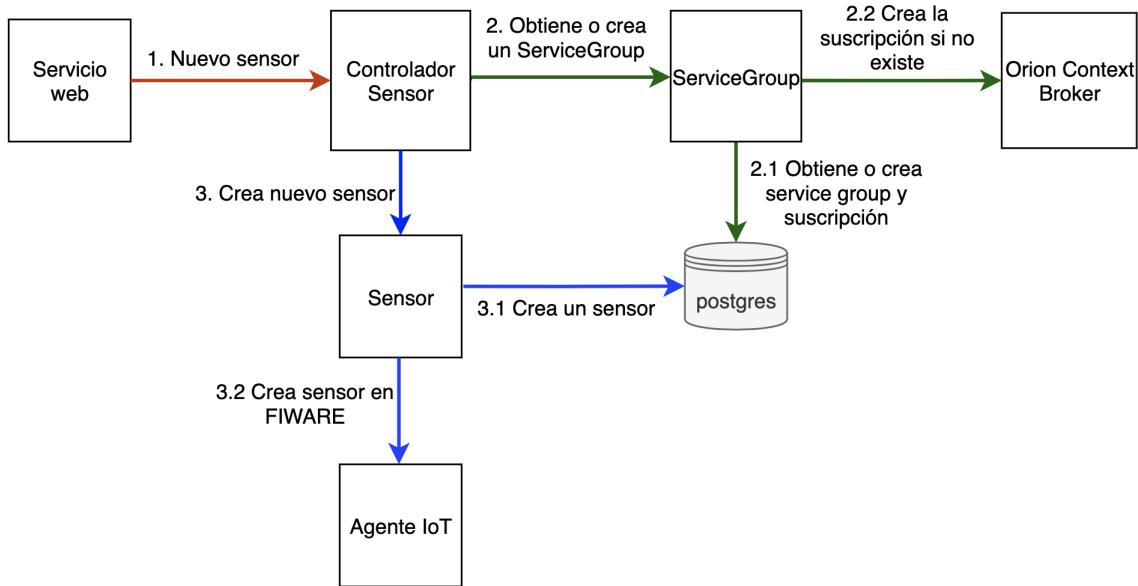


Figura 32: Diagrama de comunicación para crear sensores

Un usuario llena el formulario presentado en las Figuras 25 y 26, con los valores que tendrá el sensor. Luego se envía la petición y el servidor pasa a crear el *service group* al que pertenece el sensor en caso de que este no haya sido creado. A través de esta creación se realiza también un pedido a Orion para crear la suscripción del tipo de sensor con un *payload* personalizado el cual se adapta a lo requerido por el servidor SIG. Por último se crea el sensor con los parámetros ingresados por el usuario en una base de datos local y se realiza un POST al *endpoint* provisto por el Agente IoT para crear nuevos dispositivos con esta información.

5.4.2. Simulación de datos

A continuación se muestra un diagrama de comunicación realizado por la plataforma para llevar a cabo esta funcionalidad.

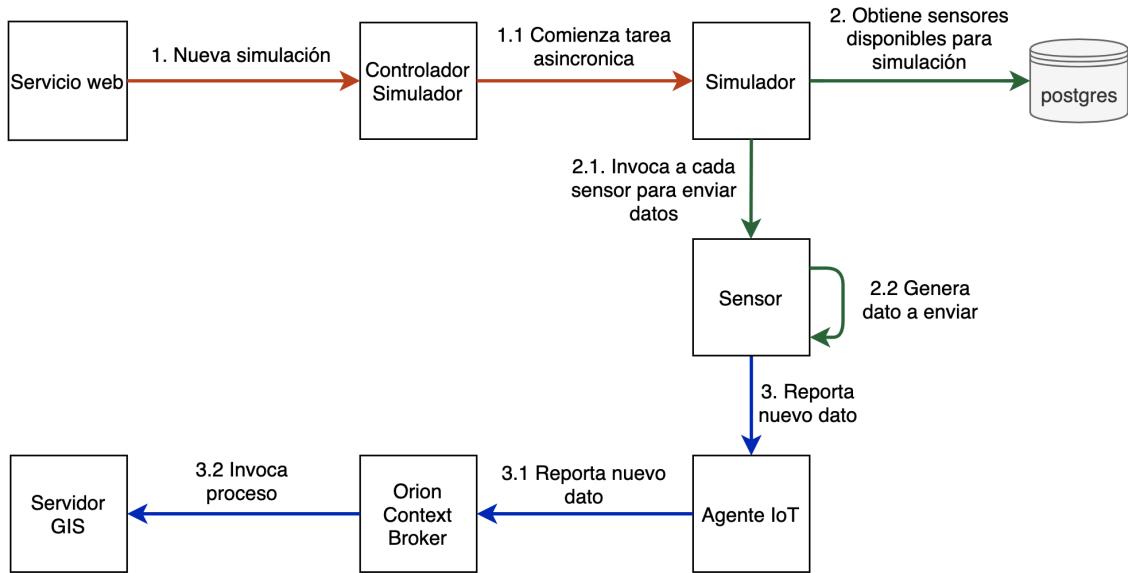


Figura 33: Diagrama de comunicación para simular datos

Primero un usuario indica qué tipo de sensor desea simular, ya sea playas u ómnibus. El siguiente paso es crear una tarea asíncrona que se auto invoca cada cierto período de tiempo, la cual se implementa con la gema³¹ de Ruby *Resque*³². Luego, en dicha tarea se crea un hilo nuevo por cada sensor, el cual se encarga de generar y enviar un nuevo dato al Agente IoT. Para finalizar un proceso de simulación, es necesario saber el *id* de la tarea que se crea con *Resque*. Dicho identificador es almacenado en el sistema en el momento de creación de la misma.

5.5. Implementación

5.5.1. Representación de datos

La aplicación de gestión de sensores cuenta con las siguientes clases para resolver los procesos antes descritos.

³¹Una gema es un paquete o librería del lenguaje Ruby

³²<https://github.com/resque/resque>

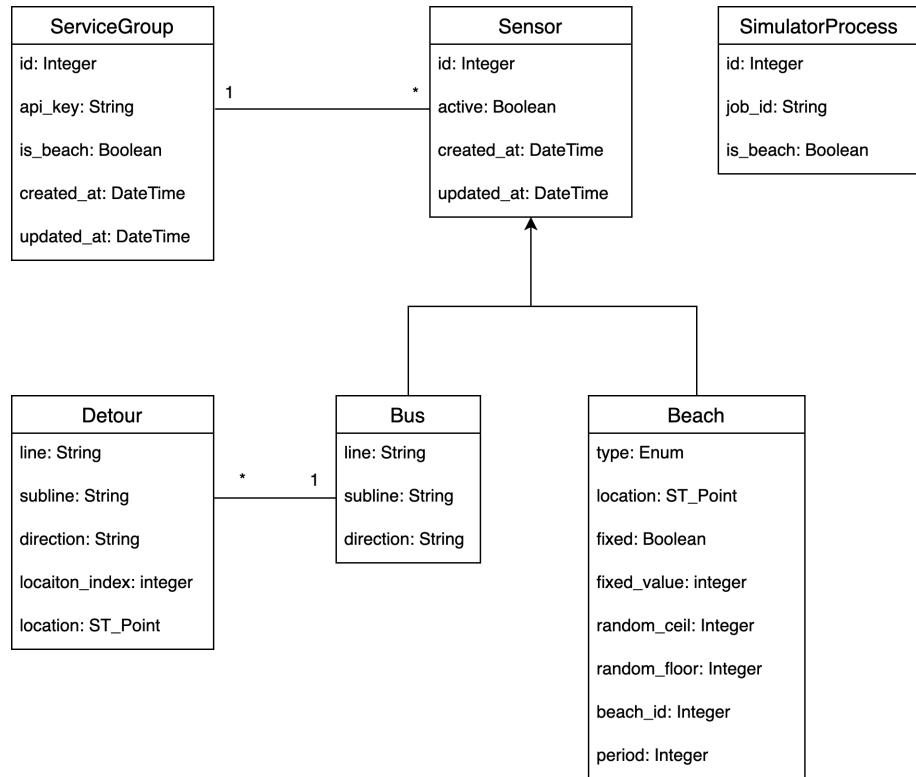


Figura 34: Modelo de datos de aplicación de gestión de sensores

La clase principal representada en el diagrama es el Sensor. La misma representa una entidad sensor que será responsable de simular sus propios datos y de enviarlos al Agente IoT de FIWARE. La entidad sensor posee los atributos de *id* y *active*, que representan un identificador dentro del sistema y un booleano el cual determina si el sensor debe reportar datos (es decir, se encuentra activo).

De esta entidad heredan dos, *Beach* que representa los sensores de playa y *Bus* para los sensores de tipo de ómnibus. Los atributos de los sensores de playa son los siguientes:

- **type:** Es un enumerado que representa el tipo de medida que el sensor captura. Los valores que puede tomar dicho enumerado son:
 - **agua:** Determina si el agua está habilitada para baños o no.
 - **bandera:** Representa la bandera indicada por guardavidas en las playas.
 - **uv:** Mide el índice de radiación UV.
 - **personas:** Mide el ingreso y el egreso de personas mediante movimiento.
- **beach_id:** Es un número que identifica a qué playa pertenece el sensor.
- **random.ceil** y **random.floor:** Determinan cuál es el rango de números al azar que va a estar generando el sensor.
- **period:** Intervalo de tiempo entre reporte de datos tomado en segundos.
- **fixed y fixed value:** Valor fijo si se desea que el sensor reporte siempre el mismo valor.

- **localization:** Posición geográfica del sensor.

La siguiente tabla describe los tipos de sensores, qué valores pueden reportar y cuál es su representación en el sistema.

Tipo de sensor	Valores posibles	Representación en el sistema
UV	1..11	1..11
Banderas	Verde, Amarillo, Rojo	0, 1, 2
Bañabilidad	No Permitida, Permitida	0, 1
Personas	-N..N	-N..N

Tabla 2: Representación de los dominios de los tipos de sensores.

A excepción de los valores que representan el número de personas en una playa, el resto de los sensores reportan datos con el mismo formato utilizado por el servicio de playas desarrollado por la Intendencia de Montevideo³³. Cabe destacar que los valores negativos de personas significan que las personas egresaron de la playa, mientras que los positivos indican el ingreso de las mismas.

Los atributos del sensor *Bus* son simplemente identificadores de qué recorrido se está simulando. Asimismo, el sensor de ómnibus se relaciona con una entidad denominada *Detour*, que representa el desvío del ómnibus. Ésta contiene también los mismos tres atributos que representan el recorrido al que pertenecen, pero se agrega un atributo extra el cual se denomina *location_index*. Este índice se utiliza para identificar el orden en el cual las localizaciones deben ser leídas para simular el recorrido del ómnibus.

En el diagrama también se encuentran las entidades *ServiceGroup* y *Simulator-Process*. La primera cumple con el rol de determinar si una suscripción a Orion ya fue realizada, además de indicar el cuerpo de la misma. La segunda, tiene un atributo denominado *job_id* que apunta al *id* del proceso en el cual la tarea asíncrona es ejecutada. Es importante destacar que esta clase no es la que ejecuta el proceso de simulación, sino que se instancian para saber que procesos están corriendo y su naturaleza.

5.5.2. Modelos de datos usados para sensores de playa y ómnibus

FIWARE da la posibilidad de crear dispositivos usando modelos de datos predefinidos para distintos dominios. La aplicación de gestión de sensores, cumple con esta convención, al utilizar dos modelos de datos ofrecidos al crear los sensores con el Agente IoT. Dado que los casos de aplicación son desvío de ómnibus y *ranking* de playas (para el cual se necesitan sensores de diversos tipos) se hace uso de los modelos *Vehicle* y *Device* respectivamente.

A continuación se incluyen dos ejemplos de los payloads en formato JSON enviados al Agente IoT para crear sensores de ambos tipos, comenzando con el tipo *Vehicle*.

```
payload = {
    devices: [
        {
            device_id: "Vehicle#{ID}",
            ...
        }
    ]
}
```

³³<https://montevideo.gub.uy/areas-tematicas/cultura-y-tiempo-libre/playas>

```

entity_name: "urn:ngsi-ld:Vehicle:#{ID}",
entity_type: "Vehicle",
static_attributes: [
    { name: "vehicleType", type: "Text", value: "bus"
    },
    { name: "category", type: "Text", value: "public"
    },
    { name: "heading", type: "Integer", value: "#{SENTIDO}"},
    { name: "fleetVehicleId",
        type: "Text",
        value: "#{LINEA}-#{SUBLINEA}-#{ID}" },
    { name: "serviceProvided", type: "Text", value:
        "urbanTransit" },
    { name: "areaServed", type: "Text", value:
        "Montevideo" },
    { name: "serviceStatus", type: "Text", value:
        "onRoute" }
],
attributes: [
    { object_id: "location", name: "location", type:
        "geo:point" }
]
]
}
]
}

```

Código 17: Ejemplo de payload para la creación de sensores de tipo Vehicle

Dentro de los valores de las diferentes claves del objeto JSON se pueden apreciar diferentes variables con la siguiente sintaxis: `#{variable}`. La variable `ID` se corresponde con el `id` que se genera automáticamente para cada sensor en la aplicación. Las variables `LINEA`, `SUBLINEA` y `SENTIDO` son las que se utilizan más adelante para poder identificar el recorrido al que corresponde el sensor. Cabe destacar que estos últimos tres no son generados, son ingresados por el usuario cuando se realiza la creación del sensor.

Por otro lado, para el caso de los sensores de playa el objeto empleado es el siguiente:

```

payload = {
    devices: [
        {
            device_id: "Device#{ID}",
            entity_name: "urn:ngsi-ld:Device:#{ID}",
            entity_type: "Device",
            static_attributes: [
                { name: "simulator_id", type: "Text", value: "#{ID}" },
                { name: "location", type: "geo:point", value:

```

```
        "#{LON}, #{LAT}"} ,  
        { name: "controlledAsset", type: "Text", value:  
          ID_PLAYA } ,  
        { name: "category", type: "Text", value: "sensor"  
          } ,  
        { name: "controlledProperty",  
          type: "Text",  
          value: "#{TIPO_DE_SENSOR}" }  
    ] ,  
    attributes: [  
      { object_id: "value", name: "value", type: "Text"  
        }  
    ]  
}  
]  
}
```

Código 18: Ejemplo de payload para la creación de sensores de tipo Device

ID nuevamente se corresponde al valor generado automáticamente en la aplicación al crear un sensor. Las variables LON y LAT corresponden a la longitud y latitud respectivamente, asociadas a la ubicación del sensor. Dentro del modelo de datos de *Device*, no sólo se utilizaron algunos atributos del mismo, sino que también se aprovechó el campo *controlledProperty* para indicar el tipo de sensor mediante la variable TIPO_DE_SENSOR. Para cumplir con el estándar, se realiza la siguiente correspondencia entre los valores posibles que ofrece FIWARE con los tipos de sensores de playa de la aplicación:

Tipo de sensor de playa	Tipo FIWARE
Personas	<i>occupancy</i>
UV	<i>solarRadiation</i>
Bañabilidad	<i>waterPollution</i>
Banderas	<i>weatherConditions</i>

Tabla 3: Correspondencia entre tipo de sensor de playa y tipo FIWARE.

Por último, el atributo **value** representa la lectura que toma cada sensor. El valor depende de cada sensor.

5.5.3. Algoritmo de simulación

El algoritmo de simulación empleado por cada sensor es sencillo. Los sensores de tipo *Bus* siguen una ruta predefinida dentro de una línea determinada. Para lograr esto, se utiliza el campo *location_index* el cual ordena los puntos de un recorrido de menor a mayor, por lo que una entidad sensor de ómnibus sólo debe realizar el siguiente comportamiento:

```
localizacion ← localizacion_actual(detour_asociado)
next_location_index ← location_index_actual(detour_asociado) + 1
reportar_localizacion()
detour_asociado ← Detour.find_by(location_index : next_loaction_index)
```

Algoritmo 1: Simulación de recorridos

Por otro lado, los sensores de playa simulan datos al azar que responden a una variable aleatoria de distribución uniforme dentro del rango ingresado en el momento de su creación. Cabe destacar que el sensor de personas sin embargo tiene ciertas particularidades y por esto destaca ante los demás. Se tiene que considerar la cantidad de personas de la playa para no exceder cierta capacidad. Asimismo, los valores de mínimo y máximo se van ajustando en el tiempo para simular el comportamiento de llegada de gente y salida que se podría observar en la realidad. El comportamiento es el siguiente:

```
nuevo_valor ← generar_numero_azar_dentro_rango()
if nuevo_valor + cantidad_actual <= capacidad then
    reportar_valor(nuevo_valor)
else
    reportar_valor(0)
end if
ajustar_maximo_minimo(tiempo)
```

Algoritmo 2: Simulación de contador de personas

6. Aplicaciones de usuario implementadas

En este capítulo se presentan las aplicaciones de usuario desarrolladas para validar la viabilidad de la plataforma. Con estas se busca poner a prueba las capacidades de la integración entre los estándares de OGC API y FIWARE.

Como fue descrito en el Capítulo 3 estas aplicaciones son viables de implementarse en Montevideo ya que están basadas en la infraestructura y sensores existentes. Asimismo, implementan funcionalidades que resultan útiles para el ciudadano común. Por un lado se tiene una aplicación que alerta al usuario de desvíos de ómnibus, mientras que por otro lado se tiene una aplicación de playas la cual brinda entre otros datos el aforo de las mismas.

En el presente capítulo se describen las funcionalidades principales de las aplicaciones y los aspectos importantes sobre su arquitectura, diseño e implementación. Finalmente se presentan conclusiones sobre las ventajas y desventajas del uso de la plataforma como base de construcción para las aplicaciones.

6.1. Aplicación de desvíos de ómnibus

Esta aplicación permite al usuario consultar por desvíos de las distintas líneas del sistema de transporte metropolitano de Montevideo. Se trata de una interfaz Web en la que se muestran las paradas de los ómnibus. El usuario al seleccionar una de estas, es informado de cuáles de las líneas que pasan por esa parada tienen algún desvío de su recorrido esperado. El usuario luego puede visualizar una animación del ómnibus en movimiento sobre los puntos de desvío registrados. Los datos vienen de los sensores de GPS de los vehículos, los que reportan distintas ubicaciones a lo largo del tiempo. La plataforma registra a partir de procesos geoespaciales si estas posiciones están desviadas de la línea a la que pertenecen.

6.1.1. Funcionalidades

La aplicación consiste en una interfaz Web con un mapa donde se muestra la información asociada a paradas de ómnibus y las líneas que pasan por ellas. Para cada línea se puede identificar también si presenta desvíos en su recorrido, y en caso de tener se permite visualizarlos en el mapa. La Figura 35 muestra el estado inicial de la aplicación.



Figura 35: Aplicación de desvíos

La aplicación cuenta con dos funcionalidades principales: ver desvíos en el mapa y ver desvíos dinámicamente. Las mismas se describen en las secciones 6.1.1.1 y 6.1.1.2 respectivamente.

6.1.1.1 Ver desvíos en mapa

La primera funcionalidad se puede observar al seleccionar una parada en el mapa. Como consecuencia, se abre un *popup* con botones representando a las líneas de ómnibus que pasan por ella, como se ve en la Figura 36. A su vez, los botones presentan colores distintos dependiendo de si hay desvíos o no. En caso de haber desvíos para una línea, el botón se muestra de color naranja junto con un símbolo de advertencia (línea 405 con destino Peñarol en la figura). De lo contrario, se verá de color turquesa con un símbolo de verificación (línea 192 con destino Parque Rodó).

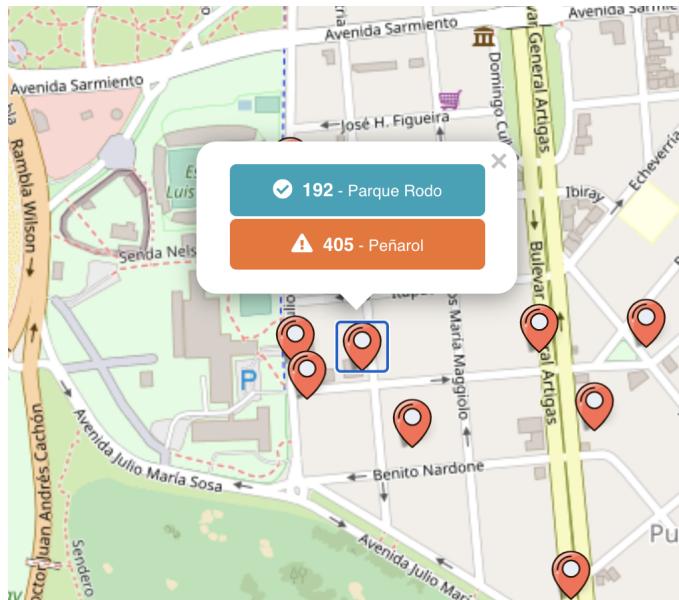


Figura 36: Aplicación de desvíos - Ver parada

Luego, al seleccionar una línea existen dos posibilidades dependiendo del color del botón. Si el botón correspondiente a la línea es turquesa y por lo tanto no presenta desvíos, se muestra el mensaje “No hay desvíos para esta línea”.

En caso de que sí se hayan registrado desvíos para dicha línea, al seleccionar el botón se muestra la advertencia representada en la Figura 37.

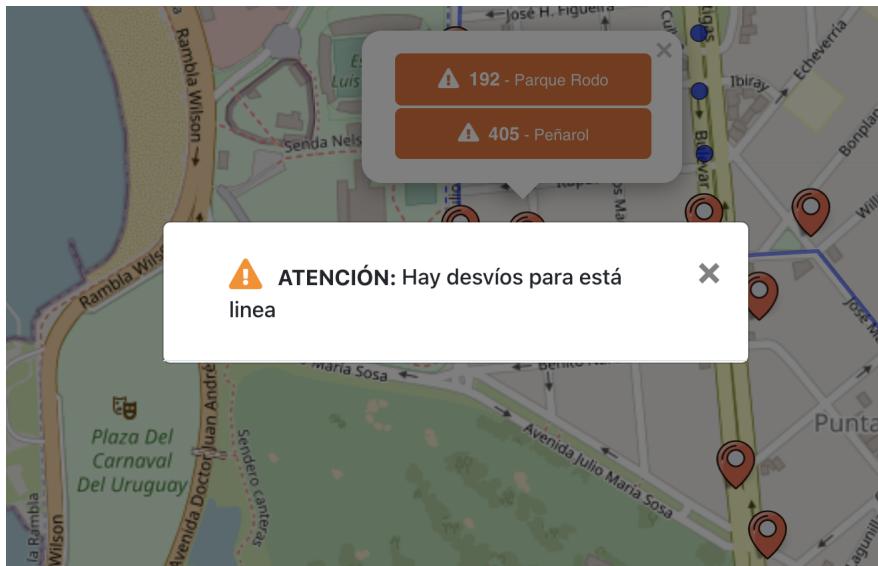


Figura 37: Aplicación de desvíos - Hay desvíos

Al cerrar el mensaje con la advertencia, se observan nuevas geometrías en el mapa. En primer lugar, se muestra una línea representando el recorrido esperado del ómnibus desviado. En segundo lugar, se incluyen también los puntos de desvío del vehículo en cuestión así como la fecha y hora en que se reportó cada punto de desvío. Ver Figura 38.

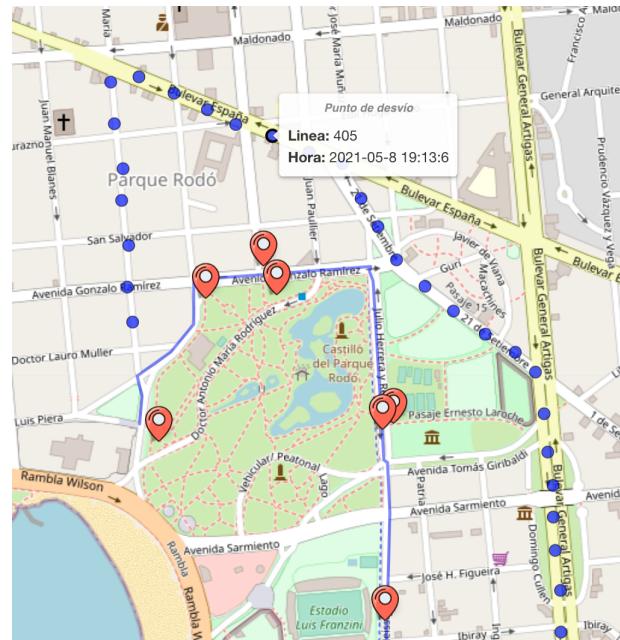


Figura 38: Aplicación de desvíos - Puntos de desvío

6.1.1.2 Ver desvío dinámicamente

En caso de existir un desvío para un ómnibus en particular, se puede observar una línea representando el recorrido esperado en el mapa. Al seleccionar esta geometría, se despliega el *popup* de la Figura 39 que incluye información relativa al vehículo y un botón con el texto “Ver desvío”.

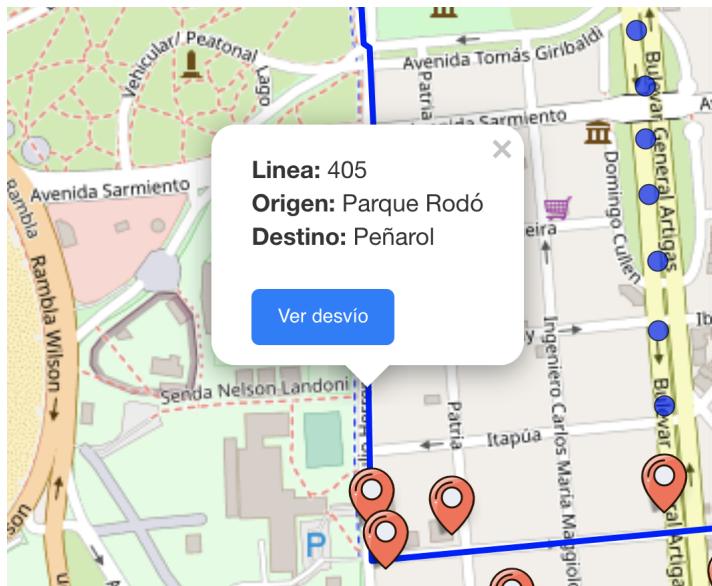


Figura 39: Aplicación de desvíos - Opción ver desvío

Este botón brinda la posibilidad de ver una animación del ómnibus en movimiento, transitando sobre los puntos de desvío en el mapa. Al seleccionarlo se abre la ventana mostrada en la Figura 40, que incluye un mapa contenido dos líneas. Por un lado una

Línea azul que indica el recorrido esperado, y por otro lado una línea roja representando el desvío del ómnibus. Esta última a su vez es recorrida por una figura 3D de un ómnibus en color verde al iniciarse la animación.

Esta ventana presenta además un reloj en la esquina inferior izquierda, con un botón para iniciar la animación.



Figura 40: Aplicación de desvíos - Ver desvío en movimiento

6.1.2. Arquitectura

Las aplicaciones desarrolladas para probar las capacidades de la plataforma se basan en sensores móviles (Aplicación de desvíos) y sensores con ubicación fija (Aplicación de *ranking*) como fue explicado anteriormente. En ambos casos, la arquitectura propuesta para las aplicaciones Web implementadas es similar a la utilizada para el sistema de gestión de sensores, ya que se basan en usar un conjunto de *endpoints* provistos en este caso por el servidor SIG.

Para la aplicación Web de desvío, se utilizan únicamente las capas de *features* del servidor SIG (implementadas con OGC API Features) para obtener los datos relacionados relevantes al caso, que se describen en la siguiente sección. Por otro lado, se utiliza un proceso para procesar los desvíos (implementado con OGC API Processes). Para la implementación del proceso se cuenta con los módulos de ayuda módulo SIG y módulo DB descritos en la sección 4.2.3.

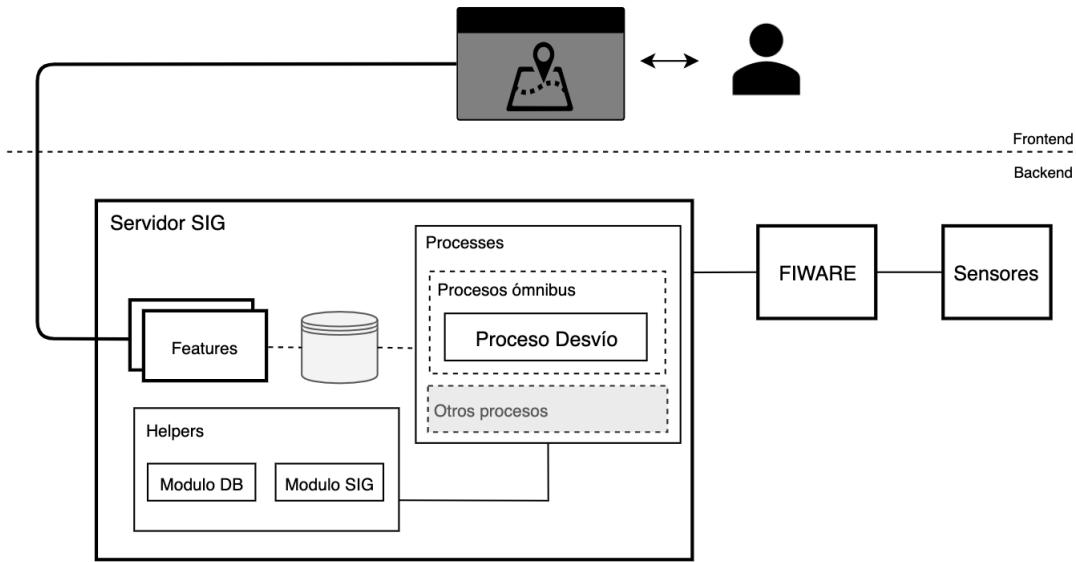


Figura 41: Arquitectura Aplicación Desvíos

6.1.2.1 Features

Las capas de *features* geográficas expuestas por el servidor SIG para el caso de desvíos se corresponden con las tablas almacenadas en la base de datos con extensión PostGIS utilizadas para este caso de aplicación.



Figura 42: Capas de *features* expuestas para aplicación de desvíos

Éstas son:

- **Paradas:** representa todas las paradas de ómnibus existentes en la ciudad. Su geometría es *Point*.
- **Desvíos:** contiene todos los puntos de desvío existentes para todas las líneas. Su geometría es *Point*.
- **Recorridos:** representa todos los recorridos existentes para cada línea de ómnibus. Su geometría es *LineString*.

- **ParadasSublineas:** capa de relación entre Paradas y Recorridos que representa todas las líneas que pasan por una parada de ómnibus. Su geometría es LineString.

Las capas de *Paradas*, *Recorridos* y *ParadasSublineas* se obtienen del Catálogo Nacional de Datos Abiertos³⁴ para los recorridos establecidos y paradas de ómnibus en Montevideo. La capa de *Desvíos* en cambio es creada específicamente para este proyecto, y sus atributos y datos se detallan en la sección 6.1.4.2 del presente capítulo.

6.1.2.2 Proceso de desvío

La finalidad del proceso de desvío es saber si una nueva lectura de posición corresponde a un desvío cuando es comparado con un recorrido conocido. En caso de que lo sea, el proceso persiste los datos como un desvío en la base de datos.

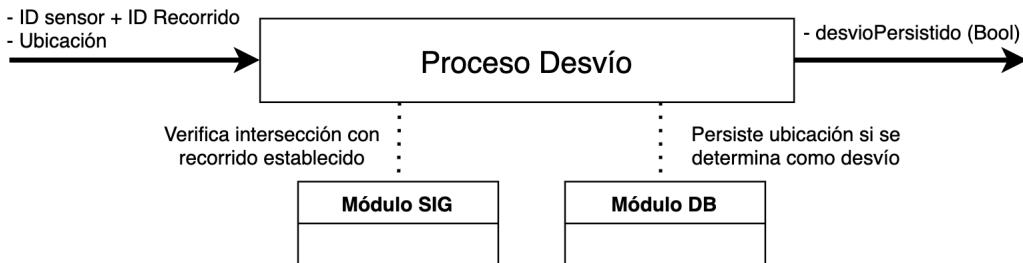


Figura 43: Arquitectura Proceso Desvío

Como se indica en el diagrama, las entradas necesarias para procesar un desvío son las siguientes:

- **ID sensor:** Identificador del sensor en la base de datos de FIWARE.
- **ID recorrido:** Conjunto de atributos que identifican un recorrido.
- **Ubicación:** Punto en formato GeoJSON y tiempo en que se realizó la medida.

El proceso verifica si la distancia de la ubicación recibida al recorrido definido para la línea está dentro de los parámetros esperados o si se trata de un desvío. Esto se hace utilizando las funciones geoespaciales de *buffer* e intersección del módulo SIG. En primer lugar se calcula un *buffer* de radio 25 metros tomando como centro la ubicación recibida. Como resultado se genera el círculo rojo representado en la Figura 44. Luego se calcula la intersección entre la geometría resultante del *buffer* y la geometría que representa el recorrido esperado del ómnibus (línea azul en la figura). En caso de que la intersección sea vacía, se resuelve que el punto representa un desvío. En caso contrario, se determina que no es desvío.

³⁴<https://catalogodatos.gub.uy/>

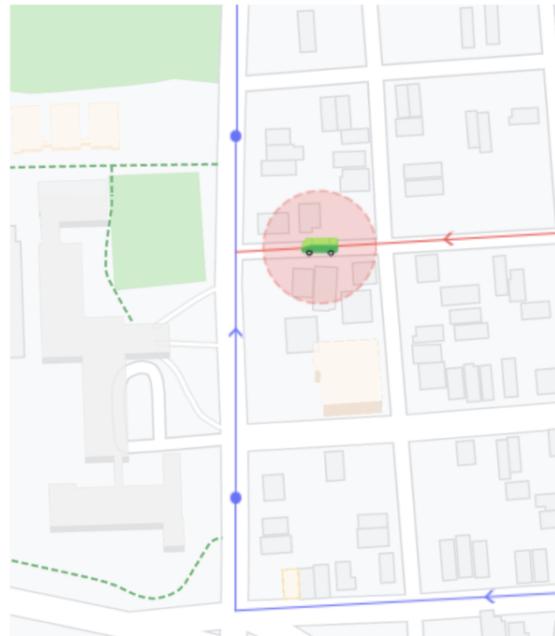


Figura 44: Determinación de desvío de vehículo

Si se determina que es un desvío, se registra el punto, junto con la hora de su medida en la base de datos con la ayuda de las operaciones del módulo DB. Finalmente, el proceso devuelve el booleano *desvioPersistido* que indica si la lectura realizada se trata de un desvío o no, y por ende si el mismo fue persistido como tal.

6.1.3. Diseño

Se presenta en la Figura 45 un diagrama de comunicación entre los componentes principales involucrados para la aplicación de desvíos.

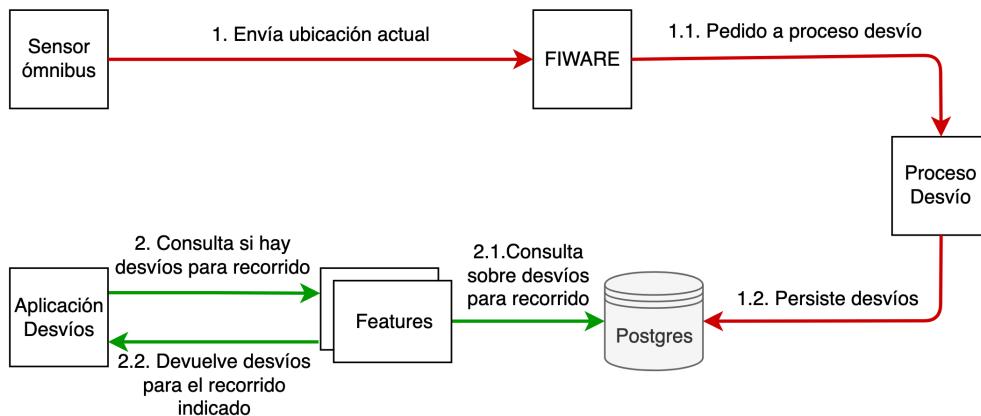


Figura 45: Diagrama de comunicación de aplicación de desvíos

En el diagrama se aprecian en rojo las acciones relacionadas al flujo del registro de desvíos en el sistema. En esta secuencia se observan los pasos de reporte de datos por

parte del sensor (paso 1), cómo estas posiciones son enviadas al proceso desvío (paso 1.1) y cómo eventualmente se guardan los puntos de desvío en la base de datos (paso 1.2).

Luego, en verde, se representan las acciones que se desencadenan al utilizar la aplicación Web de desvíos. Se engloban las funcionalidades que llevan a que la aplicación Web consulte al servidor SIG, específicamente buscando las *features* de desvíos (pasos 2 y 2.1). En caso de que existan desvíos para el recorrido indicado por el usuario, los mismos se devuelven como respuesta a la primera solicitud (paso 2.2).

6.1.4. Implementación

6.1.4.1 Tecnologías utilizadas

El siguiente diagrama contiene las tecnologías utilizadas para la implementación de la aplicación de desvíos.

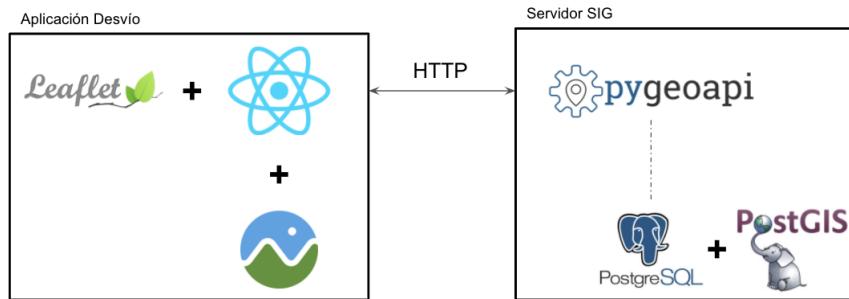


Figura 46: Tecnologías utilizadas para aplicación de desvíos

La aplicación Web se implementa con React³⁵, y se utilizan las variantes de las librerías Leaflet³⁶ y Cesium³⁷ de dicho *framework*.

En cuanto al servidor SIG, se utiliza la herramienta Pygeoapi para la implementación de los estándares OGC API Features y OGC API Processes, en conjunto con una base de datos PostgreSQL con extensión PostGIS.

Leaflet

Leaflet es una biblioteca de JavaScript para desarrollar aplicaciones Web y móviles con mapas interactivos, que funciona de manera eficiente en todas las principales plataformas de escritorio y móviles. Permite montar mapas de forma sencilla en una aplicación Web, así como cargarlos de *features* y capas de datos geoespaciales en distintos formatos, y agregarles distintos estilos y funcionalidades [58]. En esta aplicación se utiliza para mostrar e interactuar con las *features* geográficas como fue presentado en las funcionalidades.

³⁵<https://es.reactjs.org/>

³⁶<https://leafletjs.com/>

³⁷<https://cesium.com/>

Cesium

CesiumJS es una librería JavaScript de código abierto para construir globos 3D y mapas 2D a ser visualizados en un navegador Web. Entre sus distintas funcionalidades, se destaca principalmente por facilitar el manejo de datos geoespaciales que varían con el tiempo, permitiendo desarrollar aplicaciones Web interactivas [59]. En particular Cesium permite generar animaciones mediante el uso de objetos en formato JSON llamados documentos CZML [60].

Cesium es utilizado para poder desplegar la animación que simula al vehículo en movimiento transitando sobre los puntos de desvío (ver Figura 40). Para este caso, el documento CZML consiste en una lista de dos objetos JSON. El primero representa las configuraciones globales, como puede ser la configuración del reloj observado en la parte inferior izquierda de la Figura 40. El segundo objeto representa a la escena que se quiere visualizar. En esta se describen las coordenadas de los puntos a recorrer en la animación, en que momento se recorren y las propiedades del modelo 3D que realizará la animación.

6.1.4.2 Modelo de datos

Para la aplicación de desvíos se utilizan datos de ómnibus sacados del Catálogo Nacional de Datos Abiertos ofrecidos por la Intendencia de Montevideo. En el siguiente esquema se excluyen algunos atributos que están presentes en los datos originales, pero no son relevantes para la aplicación propiamente dicha.

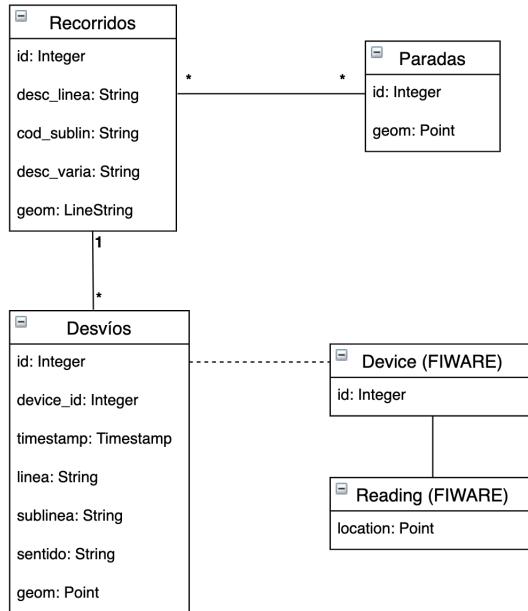


Figura 47: Modelo de datos aplicación de desvíos

Para mantener compatibilidad con los datos abiertos disponibles, se mantienen los nombres de los atributos para recorridos y paradas. El mapeo entre atributos es el siguiente:

- **desc_linea** se representa como **linea** en la tabla **desvíos**.

- `cod_sublin` es `sublinea`.
- `desc_varia` se corresponde con `sentido`.

De esta forma, cada dato ingresado en la tabla de desvíos se puede asociar con un recorrido. Por otro lado, se utilizan los atributos `device_id`, `timestamp` y `geom` para representar el sensor que reporta los datos, en qué momento lo hizo y cuál fue la ubicación que se determinó como desvío respectivamente. Los mismos están mapeados directamente a lo que reportan y guardan los componentes FIWARE.

6.2. Aplicación de ranking de playas

Esta aplicación permite al usuario consultar en tiempo real distintas características que presenta una playa, como puede ser su aforo o índice UV. Asimismo, brinda la posibilidad de obtener un *ranking* de playas en base a un conjunto de preferencias configurables por el usuario. De esta forma, la aplicación ayuda a los usuarios a tomar la decisión sobre cuál es la mejor playa de acuerdo a sus intereses. Por ejemplo, un ciudadano que deseé concurrir a una playa en particular puede conocer a través de la aplicación si el agua está apta para bañarse, o exactamente cuántas personas se encuentran en la playa en ese momento. De fondo los datos vienen de los diversos sensores colocados en las playas, los cuales reportan valores que se reflejan en tiempo real en la interfaz de la aplicación.

6.2.1. Funcionalidades

La aplicación de Ranking de playas consiste en una interfaz Web que contiene un mapa donde se pueden ver los sensores de las playas y consultar por sus valores en tiempo real, como fue explicado anteriormente. También cuenta con un panel que le permite al usuario ingresar el conjunto de preferencias para obtener así la lista ordenada de playas acorde a las mismas. El estado inicial de la aplicación se muestra en la Figura 48.



Figura 48: Aplicación de *ranking* de playas

6.2.1.1 Visualización de lecturas en tiempo real

Una de las funcionalidades principales que ofrece la aplicación es la posibilidad de ver información actualizada sobre las playas. Se permite ver las últimas lecturas reportadas por los sensores, así como observar información global relativa a cada playa. A su vez, todas las variables medidas por los sensores se actualizan en tiempo real en la interfaz a medida que éstos van reportando nuevos datos.

Cuando el usuario mueve el cursor sobre un sensor en particular, aparece un *popup* que muestra la información actualizada del mismo. En la Figura 49 se ejemplifica esta funcionalidad para el caso de un sensor de radiación ultravioleta en la playa Ramírez.



Figura 49: Aplicación de *ranking* de playas - Información de sensor

Algo similar sucede cuando se coloca el cursor sobre una de las playas. En ese caso, también aparece un *popup* con información estática y dinámica de la playa. Dentro de los datos dinámicos se encuentra la cantidad de personas que están en la playa en ese instante, así como la densidad de personas por metro cuadrado. Ver Figura 50.



Figura 50: Aplicación de *ranking* de playas - Información de playa

A su vez, tanto la geometría de la playa en el mapa como el *popup* de información avanzada cambian de color según la cantidad de gente que tiene la misma en ese instante. Los colores posibles son verde, naranja y rojo en orden ascendente de densidad de personas. La Figura 51 muestra esta funcionalidad para el caso de tres playas con distintos niveles de densidad.



6.2.1.2 Cálculo de *ranking*

La segunda funcionalidad provista por la aplicación es la posibilidad de obtener un *ranking* de las mejores playas para un usuario particular. Para esto, el usuario utiliza el panel de preferencias presentado en la Figura 52. Este panel muestra distintas variables de interés, como por ejemplo que la playa tenga poca cantidad de personas, o la distancia del usuario a la playa, la cual es calculada en base a la localización actual del usuario. Para cada una de las variables de interés, el usuario puede seleccionar un *emoji* que represente el nivel de importancia que le quiera dar a dicha variable. Luego, las preferencias ingresadas por el usuario son consideradas al momento de calcular cuáles son las mejores playas para él.



Figura 52: Aplicación de *ranking* de playas - Preferencias del usuario

Luego de procesar las preferencias del usuario, se muestra en la interfaz el *ranking* de playas en forma de acordeón como se ve en la Figura 53. Al seleccionar una playa en el resultado, se permite observar además los valores promedio de cada variable de la playa utilizados para el cálculo del *ranking*.

#1	Buceo	
👤	Densidad (personas por m ²):	0.309
☀️	Radiación UV:	11
🚩	Bandera:	amarilla
🏊	Bañabilidad:	permitida
📍	Distancia (km):	4.017

#2	Pocitos
----	---------

#3	Ramirez
----	---------

Figura 53: Aplicación de *ranking* de playas - Resultado

6.2.2. Arquitectura

Para la aplicación de *ranking* de playas, en lo que respecta a la obtención y visualización de capas de *features* geográficas relativas a playas, se utilizan los *endpoints* correspondientes ofrecidos por el servicio *Features* del servidor SIG (implementadas con OGC API Features). Además, se definen dos procesos dentro del servicio *Processes* de este servidor (implementados con OGC API Processes).

El proceso de *ranking* recibe las preferencias del usuario y su ubicación, y devuelve una lista ordenada de playas. Para calcular el *ranking* de playas, se utilizan las lecturas realizadas por los diversos sensores. Para contar con la información de los sensores en la base de datos es necesario tener un proceso que sea capaz de actualizar dichas lecturas, así como dar de alta nuevos sensores que se asocien a la playa. Dicho proceso se llama *Upsert* como se muestra en la figura a continuación. Para la implementación de ambos procesos se utilizan los módulos de ayuda módulo SIG y módulo DB descritos en la sección 4.2.3.

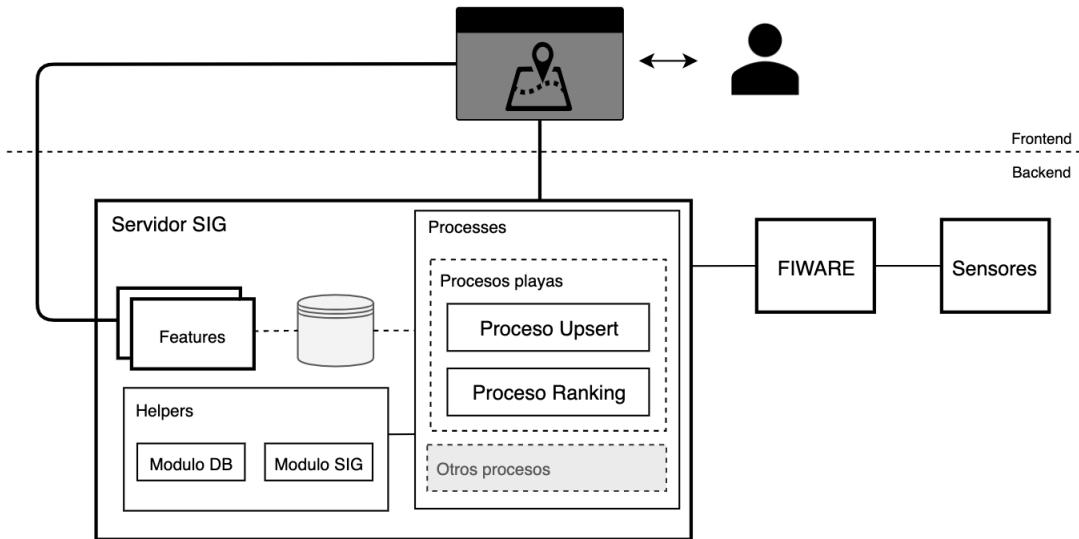


Figura 54: Arquitectura de aplicación de *ranking* de playas

6.2.2.1 Features

Las capas ofrecidas por el servidor SIG mediante OGC API Features para la aplicación de *ranking* de playas también se corresponden con las tablas utilizadas en la base de datos.

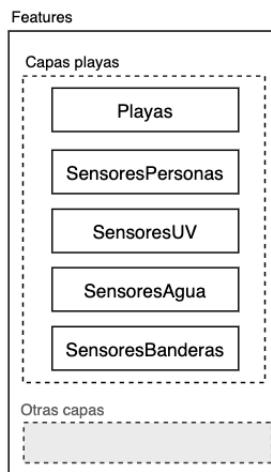


Figura 55: Capas de *features* expuestas para aplicación de *ranking* de playas

Éstas son:

- **Playas:** representa las playas en la ciudad. Su geometría es **Polygon**.
- **SensoresPersonas:** modela los sensores que cuentan la cantidad de personas ingresadas y egresadas de una playa. Su geometría es **Point**.
- **SensoresUV:** contiene los sensores que miden la radiación UV. Su geometría es **Point**.
- **SensoresAgua:** representa los sensores que miden la calidad del agua. Su geometría es **Point**.
- **SensoresBanderas:** sensores que indican el color de la bandera en una sección de la playa. Su geometría es **Point**.

Como se detalla a lo largo del Capítulo 5 los datos en las aplicaciones de usuario son simulados. Esto permite que los sensores utilizados no se correspondan con dispositivos de la realidad. Siguiendo esta línea, las capas de *SensoresPersonas*, *SensoresUV*, *SensoresAgua* y *SensoresBanderas* fueron creadas específicamente para este proyecto, y representan sensores ficticios. En cuanto a las playas, no existen datos abiertos disponibles con sus geometrías, por lo que se creó manualmente una capa con un subconjunto de playas de Montevideo.

6.2.2.2 Proceso Upsert sensor

El proceso de Upsert sensor se encarga de insertar un sensor en la base de datos, en caso de que no se encuentre creado. Cuando ya existe, se actualiza en la base de datos el campo correspondiente a la última medida registrada.

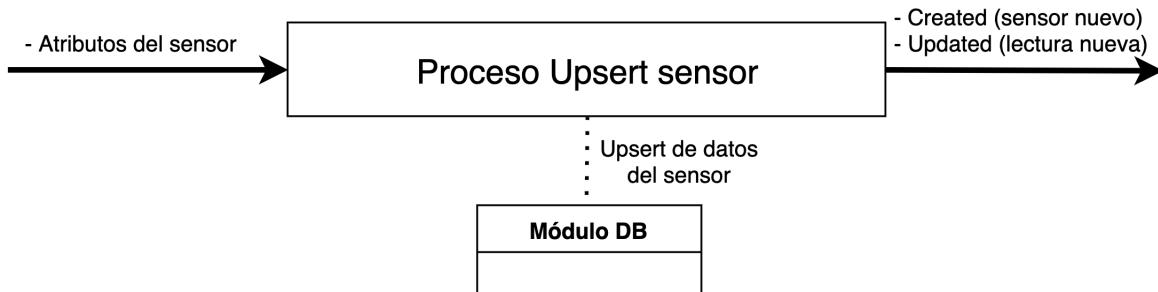


Figura 56: Arquitectura Proceso Upsert sensor

La única entrada que recibe este proceso es el conjunto de atributos del sensor. Este conjunto puede variar dependiendo del tipo de sensor. La salida del proceso simplemente indica si se trató de una operación de inserción o de modificación de la lectura.

6.2.2.3 Proceso ranking

El proceso de *ranking* se encarga de calcular y devolver una lista ordenada de playas en base a las preferencias recibidas por parte del usuario.

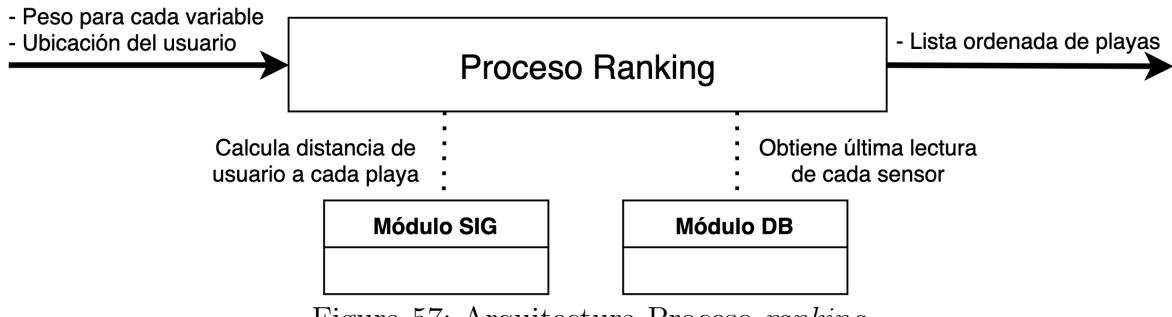


Figura 57: Arquitectura Proceso *ranking*

La entrada del proceso se compone de dos elementos. El primero es un conjunto de pesos representados por números del 1 al 3, que se corresponden con las preferencias ingresadas por el usuario en el panel de la Figura 52. El segundo elemento es la ubicación actual del usuario.

En primer lugar, el proceso de *ranking* calcula la distancia del usuario a cada playa existente en la base de datos utilizando operaciones del módulo SIG. Luego, lee todas las últimas lecturas reportadas por cada sensor utilizando el módulo DB. Por último, el proceso combina los pesos recibidos en la entrada con la distancia y los valores obtenidos de la base de datos para calcular un puntaje para cada playa. Dicho puntaje es un número real, por lo que permite ordenar las playas en orden descendente sobre dicho valor, donde la mejor playa para el usuario se ubica en la primera posición y la peor en la última. Esta lista ordenada es luego devuelta por el proceso.

6.2.3. Diseño

Se presenta en la Figura 58 un diagrama de comunicación que representa la interacción entre los componentes del sistema para los casos de uso de la aplicación de playas.

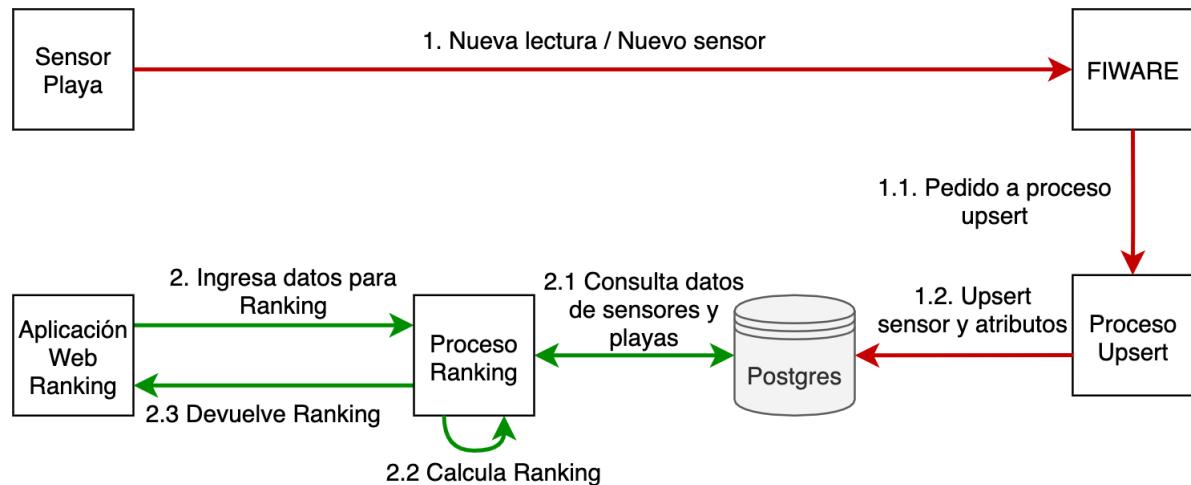


Figura 58: Diagrama de comunicación de *ranking* de playas

Como se explica en la sección 6.1.3 de Diseño para la aplicación de desvíos, el paso

1 “Nueva lectura / Nuevo sensor” engloba varias acciones sobre el componente FIWARE que se detallan en la sección 3.2.2.

Para esta aplicación el proceso al que reporta el Context Broker es el *Upsert*, que recibe los datos actualizados de un sensor (paso 1.1). Al ejecutar el proceso se hace precisamente el *upsert* del sensor (*insert* si no existe, *update* si ya existe) reflejado en el paso 1.2.

Luego, se representan las acciones que se desencadenan cuando los usuarios usan la funcionalidad de *ranking* desde la aplicación Web. Se invoca al proceso de igual nombre con las preferencias ingresadas por el usuario para cada variable (paso 2). El proceso procede a obtener las lecturas actualizadas de los sensores para cada playa, así como otros datos pertinentes sobre las mismas (paso 2.1). Luego se calcula el *ranking* (paso 2.2), y finalmente se devuelve al usuario (paso 2.3).

6.2.4. Implementación

6.2.4.1 Tecnologías utilizadas

El diagrama de la Figura 59 incluye las tecnologías utilizadas para la implementación de la aplicación de *ranking* de playas.

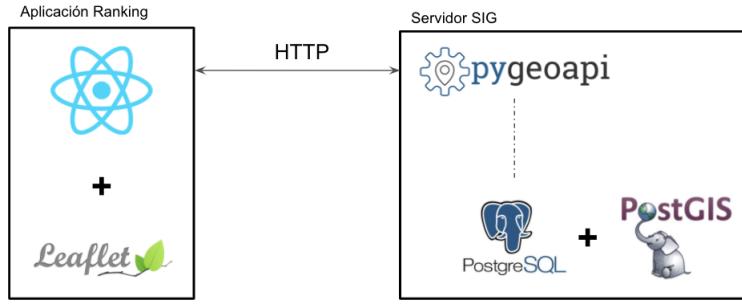


Figura 59: Tecnologías usadas para aplicación de *ranking*

Al igual que la aplicación de desvíos, esta aplicación Web se implementó con el framework React y la librería Leaflet. En cuanto al servidor SIG, se utiliza la misma instancia que para la aplicación de desvíos, y por lo tanto las mismas tecnologías.

6.2.4.2 Modelo de datos

Para la implementación de este caso de aplicación se utilizan principalmente dos modelos de datos, el de playas y el de sensor

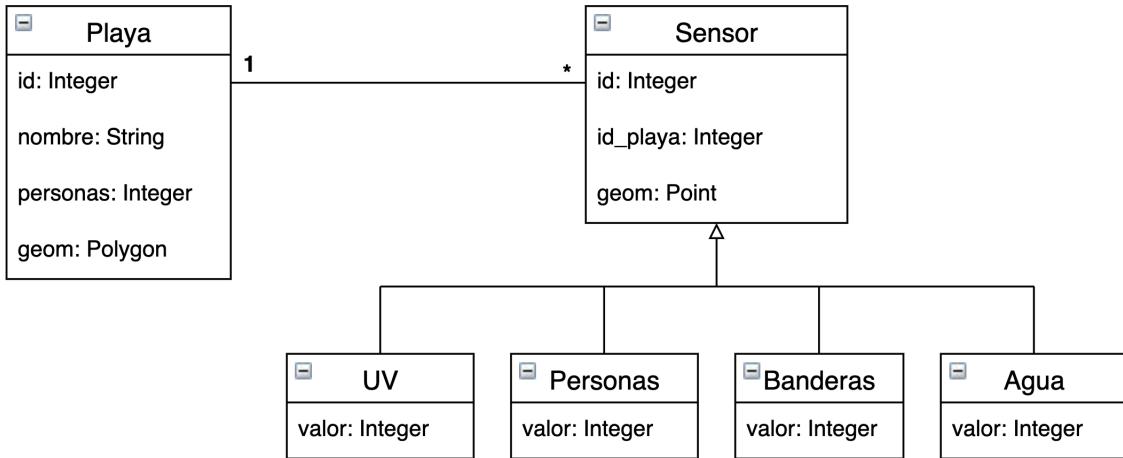


Figura 60: Modelo de datos playas y sensores

Las playas se identifican a través de un identificador numérico, que se asocia a las distintas tablas existentes para cada sensor. Todos los sensores tienen los mismos campos, aunque la diferencia entre ellos radica en el dominio de valores que pueden tomar las lecturas reportadas. Por ejemplo, el rango de radiación ultravioleta suele comprender valores enteros de entre 1 y 11. En cambio, la bañabilidad indicada por un sensor de agua puede ser 1 si está permitida, y 0 si no.

Cabe destacar que el número de personas presente como atributo en el modelo de playas representa la suma de los valores de sensores de personas de cada una. Dicho número se actualiza cada vez que se obtiene una nueva lectura.

6.2.4.3 Proceso *ranking*

Como fue explicado en la sección de Arquitectura del presente capítulo, el proceso de *ranking* recibe los pesos relativos (asociados a las preferencias del usuario) para cada tipo de sensor y la ubicación actual del usuario. Mediante la combinación de los datos de la entrada del proceso y el conjunto de las lecturas de los sensores para cada playa se calcula el *ranking*. A continuación se presenta el pseudocódigo del mismo.

```

pesos ← inputUsuario[pesos]
ubicacionUsuario ← inputUsuario[ubicacion]
listaOrdenada ← []
for all playas do
    puntaje ← calcularPuntaje(playa, pesos, ubicacionUsuario)
    listaOrdenada ← insertarOrdenado(playa, puntaje)
end for
  
```

Algoritmo 3: Cálculo de *ranking*

La función representada como `puntaje = calcularPuntaje(playa, pesos, ubicacionUsuario)` utiliza funciones del módulo DB para obtener las lecturas de los sensores y utiliza la función para calcular la distancia que se encuentra en el módulo SIG. Luego de normalizar los valores y pesos, se calcula el puntaje de cada playa.

6.2.4.4 Aforo de playas

En lo que respecta a la aplicación Web de *ranking* de playas, interesa destacar una sección lógica de la implementación en particular. Al observar tanto una playa específica en el mapa como su información avanzada, se discrimina con un color distinto dependiendo del aforo actual de la misma (ver Figura 51).

La lógica en sí para esta funcionalidad no tiene mayor complejidad, ya que consiste en una secuencia de sentencias **if** para decidir si el aforo es bajo, mediano o alto.

```

densidad ← cantidadPersonas/areaPlaya
if densidad ≤ α then
    aforo ← BAJO
else if densidad ≤ β then
    aforo ← MEDIO
else
    aforo ← ALTO
end if

```

Algoritmo 4: Cálculo de aforo

La clave en este caso radica en elegir correctamente los valores α y β (con $\alpha < \beta$) para que la distinción entre un caso y otro tenga sentido. La correcta elección de estos parámetros es una tarea compleja la cual es motivo de diversas investigaciones [61] y se considera por fuera del alcance del presente proyecto. Por este motivo se asume una realidad simplificada en base a algunas premisas.

Por un lado se asume que la distancia mínima aceptada entre dos personas es de 2 metros. Por otro lado se toma como unidad base para la división del espacio al metro cuadrado. Esto se debe a que el área de una playa se almacena en dicha unidad, y por consiguiente el aforo se mide en personas por metro cuadrado. Con estas bases, si se modela a una persona como un punto que se ubica exactamente en el centro de un cuadrado, entonces dicha figura tiene que tener como mínimo un lado de 2 metros para satisfacer la distancia entre personas.

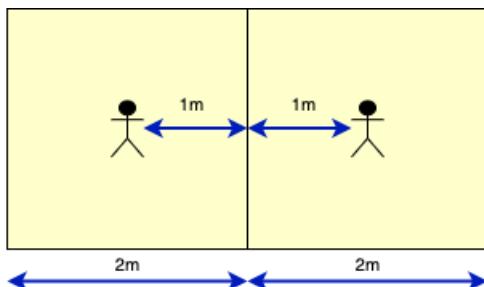


Figura 60: Distancia mínima entre personas en una playa

Por lo tanto si debe haber $1 \text{ persona}/4 \text{ m}^2$, entonces el límite de densidad para considerar que el aforo de una playa es alto debe ser $0.25 \text{ personas}/\text{m}^2$, por lo que β debe ser 0.25. Para calcular α como límite de aforo medio, simplemente se toma el punto

medio entre 0 y β y se lo redondea a dos cifras significativas. Como resultado se obtiene $\alpha = 0.13$.

6.3. Conclusiones

Como se menciona al principio de este capítulo, el objetivo principal del desarrollo de estas aplicaciones es probar la viabilidad de la plataforma propuesta. Las aplicaciones se basan en el uso de sensores móviles y sensores fijos, y datos geoespaciales de Montevideo que conforman estándares propuestos por FIWARE para ciudades inteligentes. Estos factores hacen que la prueba realizada sea lo más realista posible.

La flexibilidad ofrecida en el componente de procesos permitió la implementación de módulos reutilizables los cuales presentan varias ventajas. No solo se mejora la mantenibilidad de la plataforma al eliminar el código duplicado sino que también se simplifica la implementación de nuevas aplicaciones. Este último punto se explica debido a que los módulos sirven como bloques de construcción genéricos que son fácilmente reutilizables por nuevos sistemas que se quieran integrar.

Las aplicaciones se benefician de la integración entre FIWARE y el servidor SIG ya que cuentan con un único componente específico para hacer consultas geoespaciales. Por ejemplo, la aplicación de desvíos sólo debe consultar por los desvíos existentes para una línea, y no realizar una consulta geoespacial para calcularlos. De la misma manera, se tiene una única fuente de verdad de los datos de la realidad, lo que logra consistencia entre las instancias de las aplicaciones. Otra ventaja es que el servidor que provee las features es el mismo que el que provee las operaciones geoespaciales, lo que simplifica el código de las aplicaciones de usuario. Por ejemplo se tiene una menor cantidad de componentes distintos comunicándose entre sí, y por lo tanto una menor cantidad de comunicaciones diferentes ocurriendo a la vez.

7. Gestión del proyecto

Se detalla en esta sección la metodología de trabajo que siguió el equipo, así como un cronograma a alto nivel de las actividades llevadas a cabo durante la realización del proyecto.

7.1. Cronograma de actividades

En el siguiente diagrama se presenta el cronograma programado al principio del proyecto.

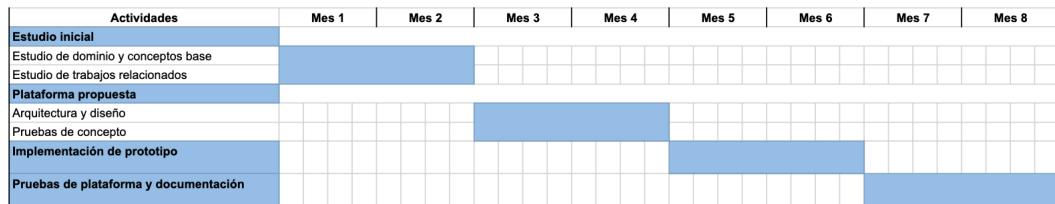


Figura 61: Diagrama Gantt de actividades por mes

En el diagrama se muestran las actividades por grupo y los meses de duración aproximados para los mismos.

- **Meses 1 y 2:** Estudio de dominio e investigación sobre trabajos relacionados.
- **Meses 3 y 4:** Diseño de una propuesta de plataforma y su arquitectura, pruebas de concepto sobre tecnologías relacionadas a la plataforma.
- **Meses 5 y 6:** Implementación de la plataforma, así como prototipos de aplicaciones.
- **Meses 7 y 8:** Pruebas de plataformas y tareas de documentación.

En el siguiente esquema se presentan las actividades realizadas durante el proyecto.

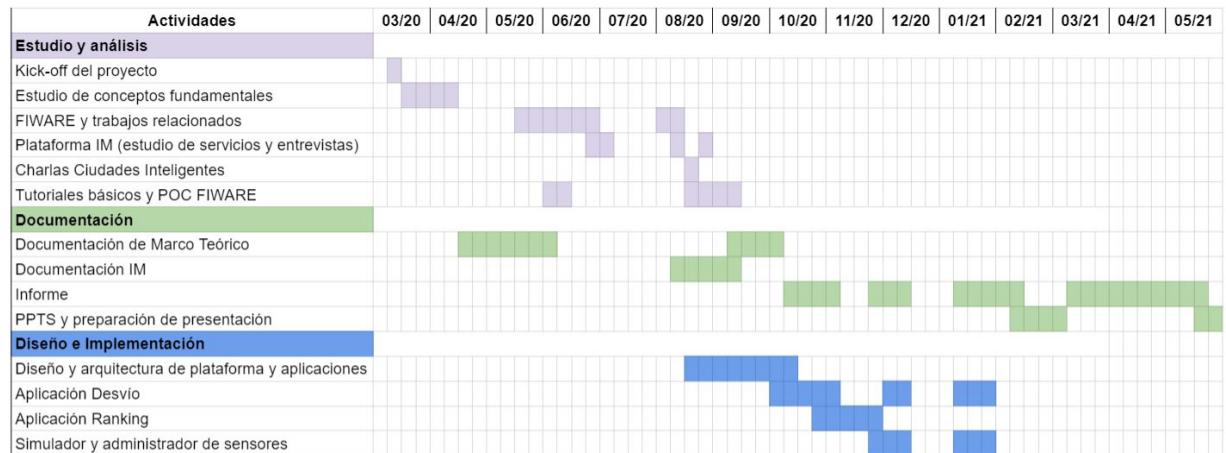


Figura 62: Diagrama Gantt de actividades por mes

Estudio y análisis

En este grupo se encuentran todas las actividades realizadas para entender conceptos relevantes al marco del proyecto, relevamiento de trabajos relacionados y descubrimiento de requerimientos para las aplicaciones implementadas. Dentro de dichas actividades se destacan las relacionadas a la Intendencia de Montevideo que constaron de entrevistas con personas del área de Desarrollo Sustentable para conocer las aplicaciones existentes en Montevideo. Por otro lado, se asistió virtualmente a las charlas sobre Ciudades Inteligentes, impulsadas por la Intendencia de Montevideo, en las que se conocieron trabajos realizados en otros departamentos o países.

Como se muestra en el diagrama, la mayor parte de estas actividades fueron llevadas a cabo en la primera mitad del año 2020.

Por último, previo a empezar las primeras tareas de implementación, se realizaron tutoriales y pruebas de concepto de las tecnologías utilizadas.

Documentación

Se incluyen en este grupo todas las tareas relacionadas a documentaciones realizadas durante el proyecto. Nuevamente, se destaca particularmente el trabajo realizado con la Intendencia de Montevideo donde se realizó un esquema de la arquitectura con la que cuentan para las diversas aplicaciones presentadas en la sección 2.2.3. Como es observable en el diagrama, se hicieron distintas tareas de documentación a lo largo de los meses, pero mayormente al final del proyecto.

Implementación

Se consideran tareas de implementación tanto las pruebas de concepto realizadas de distintas tecnologías, el desarrollo de las aplicaciones presentadas en el Capítulo 6, así como las actividades de verificación y validación llevadas a cabo para las mismas. En el diagrama se puede observar que existe más de una instancia en la que se trabajó sobre el desarrollo de una misma aplicación, esto se debe al proceso iterativo que se siguió para el desarrollo de la mayor parte del proyecto. Se incluyen más detalles sobre el mismo en la siguiente sub-sección.

7.2. Metodología de trabajo

Se brinda un resumen general de la metodología de trabajo seguida por el equipo tanto para la organización del trabajo, así como la forma en la que se implementaron las aplicaciones.

Objetivos bi-semanales

Durante la mayor parte del tiempo transcurrido para el desarrollo del proyecto, el equipo tuvo reuniones con los tutores cada dos semanas de una hora de duración apro-

ximadamente. En dichas reuniones se validaban conceptos, ideas o partes del desarrollo realizado cuando correspondía. Al final de cada reunión, el equipo junto con los tutores definían nuevos objetivos de lo que se quería lograr antes de la próxima reunión. Esto hizo que el trabajo fuera más manejable y al tener objetivos claros y concisos se pudo percibir y medir el avance del proyecto de manera efectiva.

Desarrollo iterativo

Una de las maneras en las que se lograban los objetivos bi-semanales era haciendo iteraciones sobre las aplicaciones o documentación sobre el que el equipo estuviera trabajando. No sólo se llevó a cabo el desarrollo en partes pequeñas, sino que también existieron distintas versiones de las mismas funcionalidades.

Un ejemplo de esto es la aplicación de gestión de sensores, que tuvo varias versiones a lo largo del proyecto. En su primer versión empezó siendo un script que simplemente mandaba datos para la aplicación de desvíos. Luego en una segunda iteración, consistió en un script que recibía parámetros para los dos casos de aplicación. Por último en su versión final se llegó a una aplicación Web con distintas funcionalidades. Lo mismo ocurrió con el resto de las aplicaciones.

El desarrollo iterativo permitió poder validar cada sección del código antes de pasar a las siguientes funcionalidades que surgieron durante el proyecto.

Organización de tareas

Desde el inicio del proyecto, se utilizó un tablero de Trello³⁸ para llevar una bitácora de tareas pendientes, realizadas y qué persona del equipo las realizó. Dentro del tablero se agregaron y utilizaron las siguientes columnas:

- **Things To Do (Implementación):** En esta columna se incluían todas las actividades de implementación antes mencionadas. Las tarjetas de esta columna se mantenían priorizadas según los objetivos bi-semanales que el equipo tuviera en el momento.
- **Things To Do (Informe):** En esta columna se encontraban las tarjetas pendientes relacionadas a distintas tareas relacionadas al informe. Al igual que la anterior, se mantenían priorizadas según los objetivos.
- **Doing:** En esta columna se encontraban todas las tarjetas en las que se estuviera trabajando en ese momento.
- **Done:** Por último, se encuentran en esta columna las tarjetas relacionadas a las tareas que se realizaron.

El flujo de trabajo consistía en reuniones semanales en las que el equipo priorizaba las tareas pendientes según lo que se hubiese discutido con los tutores.

Luego cada integrante del equipo era libre de asignarse una tarjeta y pasarlala a la columna **Doing** para indicar que la misma estaba en progreso y evitar el trabajo duplicado.

³⁸<https://trello.com/es>

Por último, previo a la siguiente reunión con los profesores el equipo se reunía para organizar lo que se presentaría y validaría en la siguiente reunión revisando la columna **Done**.

8. Conclusiones y trabajos futuros

En este capítulo se presentan en primer lugar las conclusiones del proyecto, y en segunda instancia un listado de los elementos más relevantes para estudiar como trabajo a futuro de esta investigación.

8.1. Conclusiones

Este proyecto investigó la integración de tecnologías geoespaciales en plataformas de *Smart Cities* basadas en FIWARE.

Para llevar a cabo la investigación, se relevaron y analizaron tres trabajos relacionados realizados con la plataforma IoT FIWARE que incluyeran tecnologías geoespaciales, en el marco de *Smart Cities*. Se estudió un trabajo internacional realizado en Italia, otro regional realizado en Brasil y uno local hecho por la Intendencia de la ciudad de Montevideo, Uruguay. Durante el análisis se identificaron elementos comunes en las arquitecturas, así como buenas prácticas que funcionaron como guía para diseñar la plataforma propuesta.

Luego, se diseñó y propuso una plataforma que integra FIWARE con tecnologías geoespaciales basadas en estándares de OGC, con el objetivo de potenciar aplicaciones en el contexto de *Smart Cities*. La misma define una arquitectura y los flujos de comunicación necesarios entre sus componentes, para desplegar sistemas que resuelvan problemas de diversos dominios.

Se desarrolló también un prototipo de la plataforma propuesta, que consistió en la integración del Orion Context Broker y un Agente IoT de FIWARE con componentes desarrollados para el prototipo. Uno de estos componentes es un servidor geoespacial implementado utilizando la herramienta Pygeoapi. El otro consiste en una aplicación Web encargada del manejo de sensores a bajo nivel y de la simulación de los datos.

Con el fin de validar la viabilidad de la plataforma propuesta, se eligieron para implementar dos casos de aplicación para cubrir el caso de sensores en movimiento, y el escenario de sensores fijos que reportan datos a lo largo del tiempo. Para dichos casos, se desarrollaron dos aplicaciones Web orientadas al ciudadano correspondientes a los casos de aplicación elegidos, utilizando como base los datos abiertos disponibles de la Intendencia de Montevideo. Una de ellas se encarga de identificar desvíos de ómnibus en tiempo real con respecto a su trayectoria. La otra aplicación es responsable de ofrecer información actualizada sobre cada playa de la ciudad, y de permitir obtener un *ranking* de las mejores de acuerdo a las preferencias del usuario.

8.2. Trabajos futuros

A continuación, se presentan posibles líneas de trabajo para continuar con la investigación realizada en este proyecto.

Integración con sistema CEP

Una alternativa posible para el procesamiento de datos en la plataforma es utilizar un GE como Perseo³⁹. Se trata de un *software* de Complex Event Processing (CEP) para procesar datos de forma masiva. Asimismo, trabajar con un CEP permite definir reglas y lógica detrás de la lectura de los datos en tiempo real. Un caso de aplicación podría ser el de detectar una tendencia de aumento de tráfico en cierta zona, para lo cual se deben procesar no sólo los datos reportados en un momento dado, sino que también considerar una porción de datos históricos.

Plataformas IoT alternativas.

Resulta interesante investigar otras plataformas IoT disponibles similares a FIWARE. Para las alternativas seleccionadas, se podría estudiar la viabilidad de su integración con tecnologías geoespaciales para resolver problemas de *Smart Cities*, y desarrollar prototipos que implementen los casos de aplicación seleccionados. Como resultado de lo anterior, se podría realizar un estudio comparativo entre las plataformas según distintos criterios de interés. Algunos ejemplos de plataformas disponibles son Cisco Kinetic⁴⁰ y AWS IoT⁴¹.

Generalización de la plataforma.

Una posible mejora a realizar es implementar la aplicación de manejo de sensores de una forma más genérica, de manera que permita agregar nuevos modelos de datos de FIWARE que se correspondan con nuevos tipos de sensores. Siguiendo la misma línea se podría desarrollar una herramienta que permita crear aplicaciones de usuario desde cero y configurar su interfaz y fuentes de datos a gusto, de forma que puedan utilizar los sensores definidos previamente en la aplicación de manejo.

Pruebas de la plataforma con datos reales

En el marco del proyecto se utilizaron mayoritariamente datos simulados. Un posible trabajo a realizar puede ser utilizar la plataforma con sensores reales. Otra posibilidad puede ser integrar la plataforma con sistemas existentes, como puede ser el de la Intendencia de Montevideo.

Mejoras sobre la plataforma.

Si bien la solución propuesta contempla casos de aplicación típicos en una plataforma que gestiona sensores, como es el acceso en tiempo real a datos reportados por dispositivos, existen funcionalidades adicionales que pueden ser interesantes para agregar. Por ejemplo, podría ser útil que la plataforma almacene un histórico de los datos

³⁹<https://perseo.readthedocs.io/en/latest/>

⁴⁰<https://www.cisco.com/c/en/us/solutions/industries/smart-connected-communities/kinetic-for-cities.html>

⁴¹<https://aws.amazon.com/es/iot/>

para su posterior acceso y visualización. De la misma manera, podría ser interesante la inclusión de controles de integridad y de acceso a los datos así como la adición de funcionalidades para la autenticación de los usuarios. Por último, se podrían realizar pruebas de carga que permitan identificar problemas en la *performance*, así como GEs u otros componentes que los resuelvan.

9. Referencias

- [1] UN. Un - world urbanization prospects. <https://www.un.org/development/desa/en/news/population/2018-revision-of-world-urbanization-prospects.html>. Último Acceso: 22 de Mayo 2021.
- [2] Vito Albino, Umberto Berardi, and Rosa Maria Dangelico. Smart cities: Definitions, dimensions, performance, and initiatives. *Journal of Urban Technology*, 22(1):3 – 21, 2015.
- [3] Saber Talari, Miadreza Shafie-Khah, Pierluigi Siano, Vincenzo Loia, Aurelio Tommasetti, and João PS Catalão. A review of smart cities based on the internet of things concept. *Energies*, 10(4):421, 2017.
- [4] FIWARE Foundation. Fiware foundation. <https://www.fiware.org/foundation/>. Último Acceso: 21 de Septiembre 2020.
- [5] George Percivall. Ogc smart cities spatial information framework. Technical report, OGC, 2015.
- [6] Arthur Souza, Jorge Pereira, Juliana Oliveira, Claudio Trindade, Everton Cavalcante, Nelio Cacho, Thais Batista, and Frederico Lopes. A data integration approach for smart cities: The case of natal. In *2017 International Smart Cities Conference (ISC2)*, pages 1–6. IEEE, 2017.
- [7] Lorenzo Carnevale, Antonino Galletta, Maria Fazio, Antonio Celesti, and Massimo Villari. Designing a fiware cloud solution for making your travel smoother: The fiware experience. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 392–398. IEEE, 2018.
- [8] Vladimir Zwass. Information system. <https://www.britannica.com/topic/information-system>, 2017. Último Acceso: 27 de Agosto 2020.
- [9] GIS First, GIS Second, and GIS Third. Geographic information systems as an integrating technology: context, concepts, and definitions. <http://gisweb.massey.ac.nz/topic/webreferencesites/whatgis/texaswhatisgis/texas/intro.htm>, 1996. Último Acceso: 30 de Noviembre 2020.
- [10] David M Mark, Nicholas Chrisman, Andrew U Frank, Patrick H McHaffie, and John Pickles. The gis history project. http://www.ncgia.buffalo.edu/ncgia/gishist/bar_harbor.html. Último Acceso: 27 de Agosto 2020.
- [11] Paul Bolstad. *GIS Fundamentals: A First Text on Geographic Information Systems*. Eider Press White Bear Lake, Minnesota, The address, 5 edition, 2016.
- [12] QGIS. Projected coordinate reference systems. https://docs.qgis.org/2.14/es/docs/gentle_gis_introduction/coordinate_reference_systems.html#projected-coordinate-reference-systems. Último Acceso: 18 de Marzo 2021.
- [13] Mark M Macomber. World geodetic system 1984. Technical report, DEFENSE MAPPING AGENCY WASHINGTON DC, 1984.
- [14] Introduction to gis. https://geogra.uah.es/patxi/gisweb/GISModule/GIST_Vector.htm. Último Acceso: 27 de Agosto 2020.

- [15] Oscar Lozano Cañadilla. Localización de áreas prioritarias de actuación para la ubicación de cortafuegos usando software libre gis: gvsig, sextante y postgis. 2008.
- [16] Environmental Systems Research Institute. What is raster data? <https://desktop.arcgis.com/en/arcmap/10.3/manage-data/raster-and-images/what-is-raster-data.htm>. Último Acceso: 20 de Noviembre 2020.
- [17] Open Geospatial Consortium. About ogc. <https://www.ogc.org/about>. Último Acceso: 27 de Agosto 2020.
- [18] Lupp M. Encyclopedia of gis. https://doi.org/10.1007/978-0-387-35973-1_903. Último Acceso: 30 de Noviembre 2020.
- [19] Open Geospatial Consortium. Web map service. <https://www.ogc.org/standards/wms>. Último Acceso: 18 de Marzo 2021.
- [20] Open Geospatial Consortium. Web feature service. <https://www.ogc.org/standards/wfs>. Último Acceso: 18 de Marzo 2021.
- [21] Open Geospatial Consortium. Web processing service. <https://www.ogc.org/standards/wps>. Último Acceso: 18 de Marzo 2021.
- [22] Open Geospatial Consortium. Ogc api. <https://ogcapi.ogc.org/>. Último Acceso: 27 de Agosto 2020.
- [23] Open Geospatial Consortium. Ogc api features. <https://www.ogc.org/standards/ogcapi-features>. Último Acceso: 30 de Noviembre 2020.
- [24] Open Geospatial Consortium. Ogc api features: Part 1 - core. <https://docs.opengeospatial.org/is/17-069r3/17-069r3.html>. Último Acceso: 30 de Noviembre 2020.
- [25] Open Geospatial Consortium. Ogc api features: Part 1 - core. <https://portal.ogc.org/files/91976>. Último Acceso: 30 de Noviembre 2020.
- [26] Open Geospatial Consortium. Ogc api features: Part 1 - core. <http://docs.opengeospatial.org/DRAFTS/19-079.html>. Último Acceso: 30 de Noviembre 2020.
- [27] Open Geospatial Consortium. Ogc api features: Part 1 - core. <http://docs.opengeospatial.org/DRAFTS/20-002.html>. Último Acceso: 30 de Noviembre 2020.
- [28] Open Geospatial Consortium. Ogc api processes. <https://ogcapi.ogc.org/processes/>. Último Acceso: 29 de Enero 2021.
- [29] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
- [30] Kevin Ashton et al. That ‘internet of things’ thing. *RFID journal*, 22(7):97–114, 2009.
- [31] Pallavi Sethi and Smruti R Sarangi. Internet of things: architectures, protocols, and applications. *Journal of Electrical and Computer Engineering*, 2017, 2017.
- [32] OASIS Open Europe Foundation. Mqtt. <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>. Ultimo Acceso: 18 de Marzo 2021.

- [33] IETF. Http. <https://tools.ietf.org/html/rfc2616>. Ultimo Acceso: 7 de Febrero 2021.
- [34] JSON RFC. Json. <https://tools.ietf.org/html/rfc7159>. Ultimo Acceso: 22 de Septiembre 2020.
- [35] FIWARE Foundation. Fiware ultralight 2.0. <https://fiware-iotagent-ul.readthedocs.io/en/latest/usermanual/>. Último Acceso: 21 de Septiembre 2020.
- [36] Jasmin Guth, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Lukas Reinfurt. Comparison of iotplatform architectures: A field study based on a reference architecture. In *2016 Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE, Nov 2016.
- [37] FIWARE Foundation. Fiware catalogue. <https://www.fiware.org/developers/catalogue/>. Último Acceso: 18 de Marzo 2021.
- [38] FIWARE. Fiware wirecloud doc. <https://wirecloud.readthedocs.io/en/stable/>. Ultimo Acceso: 22 de Septiembre 2020.
- [39] FIWARE. Fiware stream oriented generic enabler. <https://kurento.readthedocs.io/en/stable/>. Ultimo Acceso: 20 de Marzo 2021.
- [40] FIWARE. Fiware keyrock doc. <https://fiware-idm.readthedocs.io/en/latest/>. Ultimo Acceso: 22 de Septiembre 2020.
- [41] FIWARE Foundation. Fiware-ngsi v2 specification. <https://fiware.github.io/specifications/ngsiv2/stable/>. Ultimo Acceso: 10 de Diciembre 2020.
- [42] Definición srs epsg:4326. <https://epsg.io/4326>. Último Acceso: 15 de Diciembre 2020.
- [43] GeoJson RFC. Geojson. <https://tools.ietf.org/html/rfc7946>. Ultimo Acceso: 22 de Septiembre 2020.
- [44] FIWARE Foundation. Traffic north of the iot agent - ngsi interactions. <https://iotagent-node-lib.readthedocs.io/en/latest/northboundinteractions/index.html>. Ultimo Acceso: 14 de Diciembre 2020.
- [45] FIWARE Foundation. Iot agent api - architecture. <https://iotagent-node-lib.readthedocs.io/en/latest/architecture/index.html>. Ultimo Acceso: 14 de Diciembre 2020.
- [46] FIWARE Foundation. Iot agent api. <https://iotagent-node-lib.readthedocs.io/en/latest/api/index.html>. Ultimo Acceso: 14 de Diciembre 2020.
- [47] Telefónica. Iot agent provision api documentacion. <https://telefonicaiotiotagents.docs.apiary.io/#>. Ultimo Acceso: 14 de Diciembre 2020.
- [48] FIWARE Foundation. Fiware data models. <https://www.fiware.org/developers/data-models/>. Último Acceso: 22 de Septiembre 2020.
- [49] FIWARE Foundation. Fiware data models guidelines. <https://fiware-datamodels.readthedocs.io/en/latest/guidelines/index.html>. Último Acceso: 22 de Septiembre 2020.
- [50] FIWARE Foundation. Fiware device data model. <https://fiware-datamodels.readthedocs.io/en/latest/device/index.html>.

readthedocs.io/en/latest/Device/Device/doc/spec/index.html. Ultimo Acceso: 10 de Diciembre 2020.

- [51] FIWARE Foundation. Transportation harmonized data models. <https://fiware-datamodels.readthedocs.io/en/latest/Transportation/doc/introduction/index.html>. Ultimo Acceso: 13 de Diciembre 2020.
- [52] FIWARE Foundation. Fiware vehicle data model. <https://fiware-datamodels.readthedocs.io/en/latest/Transportation/Vehicle/Vehicle/doc/spec/index.html>. Ultimo Acceso: 13 de Diciembre 2020.
- [53] FIWARE. Fiware pep proxy wilma doc. <https://fiware-pep-proxy.readthedocs.io/en/latest/>. Ultimo Acceso: 22 de Septiembre 2020.
- [54] Geopython. pygeoapi. <https://github.com/geopython/pygeoapi>.
- [55] Meteorological Service of Canada. msc-pygeoapi. <https://github.com/ECCC-MSC/msc-pygeoapi>.
- [56] Postgis - documentación. <https://postgis.net/documentation/>. Último Acceso: 14 de Diciembre 2020.
- [57] A. Syromiatnikov and D. Weyns. A journey through the land of model-view-design patterns. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 21–30, 2014.
- [58] Vladimir Agafonkin. Leaflet - an open-source javascript library for mobile-friendly interactive maps. <https://leafletjs.com/>. Ultimo Acceso: 3 de Febrero 2021.
- [59] Cesium. Cesiumjs - 3d geospatial visualization for the web. <https://cesium.com/cesiumjs/>. Ultimo Acceso: 3 de Febrero 2021.
- [60] Inc. Analytical Graphics. Czml guide. <https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/CZML-Guide>. Ultimo Acceso: 3 de Febrero 2021.
- [61] Víctor Yepes Piqueras. Método simplificado de cálculo del aforo de las playas en tiempos de coronavirus. 2020.

Glosario

API *Application Programming Interface.* Es una interfaz de software que define las interacciones entre dos o más aplicaciones. 5, 6, 11–15, 17, 21–23, 25, 28–34, 40, 42, 43, 73, 78, 79

Context Broker Componente provisto por FIWARE que maneja los datos de contexto dentro de una aplicación o plataforma. 12, 13, 17, 21–23, 25, 31, 33, 34, 38, 40, 41, 46, 54–56, 82, 91

CQL *Common Query Language.* Lenguaje formal que tiene el fin de representar consultas posibles sobre un sistema. 6

CRUD *Create, Read, Update, Delete.* Acrónimo para las 4 operaciones básicas sobre una entidad. 12, 16–18, 47, 56

CZML Formato de codificación basado en JSON para la representación de una escena gráfica dinámica en el tiempo. 74

DNS Domain name system. Es un sistema jerárquico descentralizado de nombrado de sistemas o computadoras en una red. 22

endpoint Punto de entrada en una interfaz de software o dispositivo en la Web. 22, 23, 29, 30, 38, 48, 56, 57, 69, 78

feature Entidad geográfica. 3, 5, 6, 29, 30, 32, 35, 42–44, 48, 69, 70, 73, 78, 79, 85

FIWARE Iniciativa *open source* que define estándares para el manejo de datos de contexto dentro de plataformas IoT. 1–3, 11–13, 15–19, 21–24, 27, 29–31, 33–38, 46, 56, 59, 60, 62, 71, 75, 82, 85, 91, 92

framework Plataforma para el desarrollo de aplicaciones de software. 73, 82

Generic Enabler Conjuntos de componentes provistos por FIWARE. 11

GeoJSON Formato para codificar datos geoespaciales dentro de un JSON. 15, 16, 19, 45, 71

HTTP *Hyper Text Transfer Protocol.* Protocolo de capa de aplicación diseñado para el intercambio de documentos de hipertexto. 5, 6, 8, 13, 24, 29, 41

IoT *Internet of Things.* Refiere a la idea de la interconexión y cooperación de dispositivos que se conectan entre sí mediante Internet. 1, 2, 6–11, 13, 17, 19, 22–25, 31, 34, 38, 39, 56–60, 91, 92

JSON *Javascript Object Notation.* Es un lenguaje para la codificación de datos para que sean leíbles tanto por máquinas como por personas. 8, 9, 13, 14, 16, 29, 41, 42, 46, 54, 60, 61, 74

layer Conjunto de datos (ya sean *features* u otros) de un sistema de información geográfico. También conocido como capa. 3

metadato Datos sobre los datos. 6, 14, 15, 29, 42

middleware Programa que provee servicios a otros programas que los sistemas operativos no proveen. 20, 21, 28, 31

NGSI *Next Generation Service Interface.* Interfaz provista por FIWARE para manejar datos de contexto. 12–17, 31, 34, 40

OGC *Open Geospatial Consortium.* Organización internacional sin fines de lucro enfocada en el desarrollo de estándares abiertos para la comunidad geoespacial. 2, 5, 6, 29, 30, 32, 42, 43, 73, 78, 79, 91

open source Software que es de libre acceso para su modificación o distribución. 11

overhead Carga extra sobre una operación o procesamiento de datos. 30

REST *Representational state transfer.* Es un estilo arquitectónico para la provisión de diferentes datos y servicios por parte de un sistema Web. 6, 13, 21–23, 29, 32

SIG Sistema de Información Geográfica. Es un conjunto de componentes responsables de mantener, almacenar y proveer datos geográficos. 2–4, 24, 30, 32–35, 40–42, 49, 57, 69, 70, 73, 78, 79, 81–83, 85

Simple Location Format Formato utilizado para representar las figuras geométricas básicas. 15, 16

Smart City Ciudad que utiliza *smart things* para desarrollar y promover prácticas sustentables. 10, 21

Smart Thing Dispositivo o sensor que reporta, almacena o recibe datos a una o varias plataformas IoT. 7, 22, 24, 30, 31, 34

SRC Sistema de Referencia de Coordenadas. Es un sistema de coordenadas utilizado para localizar entidades geográficas. 3, 6

TCP *Transmission Control Protocol.* Protocolo de capa de transporte diseñado para la comunicación de dos entidades en la Web. 8

TIC Tecnología de la Información y Comunicación. 1, 10

ultralight Es un formato de codificación de datos para que sean leíbles tanto por máquinas como por personas. Tiene la particularidad de utilizar una cantidad reducida de caracteres haciéndolo un formato liviano. 9, 13

URL *Uniform Resource Locator.* Coloquialmente conocido como dirección de red, es el identificador de un recurso Web. 5, 33, 34, 38, 41, 42

XML *Extensible Markup Language.* Es un lenguaje para la codificación de documentos para que sean leíbles tanto por máquinas como por personas. 5, 29