



Universidad
de Cádiz

Escuela Superior
de Ingeniería

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

**SISTEMA DE MONITORIZACIÓN EN
TIEMPO REAL DEL TRÁFICO,
APARCAMIENTO Y POLUCIÓN EN LAS
CIUDADES**

AUTOR: MANUEL CANO CRESPO

Cádiz, enero 2023



Universidad
de Cádiz

Escuela Superior
de Ingeniería

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

**SISTEMA DE MONITORIZACIÓN EN
TIEMPO REAL DEL TRÁFICO,
APARCAMIENTO Y POLUCIÓN EN LAS
CIUDADES**

TUTOR: GUADALUPE ORTIZ BELLOT

COTUTOR: JUAN BOUBETA PUIG

AUTOR: MANUEL CANO CRESPO

Cádiz, enero 2023

Agradecimientos

A mi madre y a mi padre, por haberme puesto todas las facilidades y haberme dado una educación de la que puedo sentirme orgulloso, sin ellos nada de esto hubiera sido posible. Sobre todo a mi madre, sin su insistencia y su apoyo, y sin los mensajes de “¿Dónde estás? Deberías de estar estudiando”, seguramente no me encontraría ahora en este punto.

A mi hermano Martín y también a la familia que elegí, mis amigos, porque les debo mucho de lo que soy, por estar cuando uno apenas está, y porque cuando tanto la vida como la carrera se veían como una cuesta arriba, me impulsaron hacia arriba y subieron la cuesta conmigo.

Por supuesto a Guadalupe y a Juan, por su implicación y compromiso en este proyecto. Y a todos los profesores que, aparte de como profesión, entienden la docencia como una vocación y un servicio a la sociedad, y que me han enseñado tanto durante toda mi etapa como estudiante.

También a los estupendos compañeros, algunos ya amigos, con los que he compartido tanto tiempo en la universidad que, huyendo de la espiral de competitividad a la que nos empuja este sistema, no han dudado en ayudarme y permitirme aprender de ellos, y me han hecho mucho más llevaderos todos estos años.

Y, por último y no menos importante, a la educación pública, de la que debemos sentirnos orgullosos, sin que ello signifique ser complacientes y dejar de defenderla desde cualquier terreno para seguir ampliando el derecho a una universidad pública e inclusiva, que no deje a nadie atrás y garantice la igualdad de oportunidades.

Índice general

1. Visión general del proyecto	13
1.1. Motivación	13
1.2. Objetivos.....	14
1.3. Alcance	14
1.4. Visión general del documento.....	14
1.5. Estandarización del documento	15
1.6. Acrónimos.....	15
1.7. Definiciones	16
2. Contexto del proyecto.....	17
2.1. Descripción general del proyecto	17
2.2. Características del usuario	17
2.3. Modelo de ciclo de vida	18
2.4. Tecnologías	18
2.4.1. Esper	18
2.4.2. Mule ESB.....	19
2.4.3. RabbitMQ	19
2.4.4. API REST	19
2.4.5. MySQL.....	19
2.4.6. Android SDK.....	19
2.5. Lenguajes	20
2.5.1. Java	20
2.5.2. SQL.....	20
2.5.3. Esper EPL	20
2.6. Herramientas	20
2.6.1. Anypoint Studio	20
2.6.2. Android Studio.....	21
2.6.3. Eclipse IDE.....	21
2.6.4. nITROGEN	21
2.6.5. Balsamiq	21
2.6.6. Postman.....	21
3. Planificación	22
3.1. Iniciación del proyecto.....	22
3.2. Iteraciones del proceso de desarrollo	22
3.2.1. Diseño de patrones.....	23
3.2.2. Implementación de los flujos en Mule	23
3.2.3. Desarrollo de la estructura de App.....	24
3.2.4. Mejora del proyecto Mule y desarrollo de la API REST	24
3.2.5. Mejora de la aplicación Android y conexión con la API	24

3.3.	Diagrama de Gantt	26
4.	<i>Análisis del sistema</i>	27
4.1.	Especificación de requisitos	27
4.1.1.	Requisitos de información	27
4.1.2.	Requisitos funcionales	27
4.1.3.	Diagrama de casos de uso	28
4.1.4.	Descripción de casos de uso	29
4.1.5.	Diagramas de secuencia	33
4.2.	Garantía de calidad	35
4.2.1.	Seguridad	35
4.2.2.	Interoperabilidad	35
4.2.3.	Operabilidad	36
4.2.4.	Transferibilidad	36
4.2.5.	Eficiencia	36
4.2.6.	Mantenibilidad	36
4.3.	Gestión del presupuesto	36
5.	<i>Diseño</i>	38
5.1.	Arquitectura física	38
5.1.1.	Módulo de entrada	38
5.1.2.	Módulo de procesamiento	39
5.1.3.	Módulo de salida	39
5.2.	Arquitectura lógica	40
5.2.1.	Módulo de entrada	40
5.2.2.	Módulo de procesamiento	41
5.2.3.	Módulo de salida	41
5.3.	Esquema de la base de datos	42
5.4.	Diseño de la aplicación Android	42
5.4.1.	Diseño de la navegación	43
5.4.2.	Diseño de la interfaz de usuario	44
6.	<i>Implementación</i>	45
6.1.	Esquemas y patrones de eventos	45
6.1.1.	Calidad del aire	45
6.1.2.	Aparcamiento	47
6.1.3.	Tráfico	49
6.2.	Flujos de Mule ESB	52
6.3.	Aplicación de Android	61
6.3.1.	Visualización de eventos	61
6.3.2.	Mapa de aparcamientos libres	64
7.	<i>Entrega del producto</i>	68
8.	<i>Procesos de soporte y pruebas</i>	69
8.1.	Gestión y toma de decisiones	69
8.2.	Gestión de riesgos	69
8.2.1.	Análisis de riesgos	69

8.2.2.	Plan de contingencia.....	70
8.3.	Verificación y validación del software	70
8.3.1.	Patrones de eventos.....	70
8.3.2.	Mule ESB.....	72
8.3.3.	API REST.....	74
8.3.4.	Aplicación móvil.....	76
8.3.5.	Pruebas de integración.....	77
9.	Conclusiones y trabajo futuro	79
9.1.	Valoración del proyecto.....	79
9.2.	Cumplimiento de los objetivos propuestos.....	79
9.3.	Trabajo futuro	80
	Bibliografía.....	83
A.	Manual de instalación	85
i.	Aplicación móvil.....	85
ii.	RabbitMQ.....	86
iii.	nITROGEN	86
iv.	Base de datos	86
v.	Anypoint Studio y proyecto Mule.....	87
B.	Manual de usuario.....	89
C.	Boceto de la aplicación	93

Índice de figuras

Figura 1. Diagrama de Gantt del proyecto	26
Figura 2 Diagrama de casos de uso App	¡Error! Marcador no definido.
Figura 3. Diagrama de secuencia “Mule ESB”	33
Figura 4. Diagrama de secuencia "Ver detalles de evento"	34
Figura 5. Diagrama de secuencia "Ver mapa de aparcamientos libres"	35
Figura 6. Arquitectura física.....	38
Figura 7. Arquitectura lógica.....	40
Figura 8. Esquema de la base de datos	42
Figura 9. Diagrama de navegación de la App	43
Figura 10. Esquema “AirQualityEvent”	45
Figura 11. Patrón "NitrogenDioxideAlert"	46
Figura 12. Patrón "SulfurDioxideAlert"	46
Figura 13. Patrón "OzoneAlert"	46
Figura 14. "AveragePollutantLevels"	46
Figura 15. "PollutionYellowAlert"	47
Figura 16. Patrón "PollutionOrangeAlert"	47
Figura 17. Patrón "PollutionRedAlert"	47
Figura 18. Esquema "ParkingEvent"	48
Figura 19. Patrón "FreeParkingSpot"	48
Figura 20. Patrón "TotalFreeParkingSpots"	48
Figura 21. Patrón "OccupiedParkingSpot"	48
Figura 22. Patrón "TotalOccupiedParkingSpots"	48
Figura 23. Patrón "AverageOccupation"	49
Figura 24. Patrón "MaxOccupationHour"	49
Figura 25. Esquema "TrafficJamEvent"	49
Figura 26. Patrón "SpeedOver15"	50
Figura 27. Patrón "SpeedBelow15"	50
Figura 28. Patrón "PossibleTrafficJam"	50
Figura 29. Patrón "CurrentTrafficJam"	50
Figura 30. Patrón "TotalDayTrafficJams"	51
Figura 31. Patrón "AverageSpeedZoneHour"	51
Figura 32. Patrón "AverageSpeedZoneDay"	51
Figura 33. Patrón "TotalVehiclesZoneHour"	51
Figura 34. Patrón "AverageVehiclesZoneDay"	52
Figura 35. Flujo 1 Mule ESB "Recepción y tratamiento de eventos simples"	52
Figura 36. Flujo 2 Mule ESB "Despliegue de patrones en el motor CEP"	53
Figura 37. Flujo 3 Mule ESB "Almacenamiento y envío de eventos complejos"	54
Figura 38. Configuración VM Endpoint	55
Figura 39. Flujo 4 Mule ESB "Creación de servicio REST"	56
Figura 40. Configuración componente HTTP	57
Figura 41. Configuración componente REST	57
Figura 42. Método API "sayIsWorking"	57
Figura 43. Método API "addEvent"	58

Figura 44. Método API "getNumberOfEvents"	58
Figura 45. Método API "getEventosAparcamiento"	59
Figura 46. Clase de retorno de API "Event"	59
Figura 47. Parámetros conexión MySQL	60
Figura 48. Método API "getEventoAparcamiento"	60
Figura 49. Extracción de parámetro en el método onCreate	61
Figura 50. Inicialización y configuración de RecyclerView	61
Figura 51. Petición a la API REST de los eventos de tráfico	62
Figura 52. Obtención y tratado de los atributos de los eventos de tráfico	63
Figura 53. Preparación del mapa de aparcamientos libres	65
Figura 54. Obtención del estado de cada plaza de aparcamiento	66
Figura 55. Inserción en el mapa de marcadores para cada aparcamiento libre	67
Figura 56. Escenario creado en Esper Notebook	71
Figura 57. Eventos complejos generados en Esper Notebook	72
Figura 58. Logger de recepción de eventos simples	72
Figura 59. Logger de despliegue de esquema sobre motor CEP	73
Figura 60. Logger de despliegue de patrón sobre motor CEP	73
Figura 61. Logger de generación de evento complejo en motor CEP	73
Figura 62. Logger de consumo de evento complejo por flujo de Mule ESB	73
Figura 63. Mensaje recibido en cola de RabbitMQ	73
Figura 64. Eventos complejos almacenados en Base de Datos	74
Figura 65. Prueba de método getNumberOfEvents en Postman	75
Figura 66. Prueba de método getAparcamientos en Postman	76

Índice de tablas

Tabla 1. RF-01 - Seleccionar tipo de evento	29
Tabla 2. RF-02 - Ver lista de eventos complejos	30
Tabla 3. RF-03 - Ver información detallada de un evento complejo	31
Tabla 4. RF-04 - Consultar mapa de aparcamientos libres.....	32
Tabla 5. RF-05 - Consultar mapa de atascos	33
Tabla 6. Presupuesto hardware	37
Tabla 7. Presupuesto mano de obra	37
Tabla 8. Presupuesto total.....	37
Tabla 9. Riesgos analizados	69
Tabla 10. Plan de contingencia.....	70

1. Visión general del proyecto

En este capítulo se ofrecerá una visión general del proyecto: la motivación que llevó a desarrollarlo, los objetivos que pretende cumplir y el alcance del mismo; así como una visión general de la estructura y los contenidos del presente documento, los estándares que sigue y los acrónimos usados en él, y por último, las definiciones de los conceptos más relevantes del proyecto.

1.1. Motivación

Sin duda el mayor reto que tiene la sociedad por delante es la lucha contra el cambio climático que amenaza con la desaparición de nuestro planeta tal y como lo conocemos. Ya empezamos a ver en nuestro día a día las consecuencias de este: veranos con temperaturas récord, aumento de los fenómenos meteorológicos extremos, sequías históricas, etc.

Consecuentemente, debemos abordar esta problemática desde todos los frentes, y uno de los principales activos en esta lucha deben ser los ayuntamientos de las ciudades. Se debe apostar desde ya por una movilidad y una industria sostenible que hagan de la ciudad una ciudad respirable y habitable para todos.

Cuando hablamos de “Smart cities” o ciudades inteligentes, la reducción de los niveles de contaminación en las ciudades debe ser uno, sino el principal, de los focos donde dirigir los avances tecnológicos. Las tecnologías de la información, y en concreto, lo que atraviesa este proyecto, el procesamiento de eventos complejos, tienen mucho que aportar en el proceso de coordinar, mejorar y automatizar (en la medida de lo posible), la detección y actuación ante situaciones que perjudiquen al medio ambiente.

La Bahía de Cádiz y la Bahía de Algeciras son dos de los diez focos de más contaminación de Andalucía [1], según el informe “La calidad del aire en el Estado español durante 2021” publicado por Ecologistas en Acción[2]. Es evidente por tanto que debemos tomar medidas urgentes para mejorar la calidad del aire que respiramos en nuestras ciudades.

Tomando como referencia la ciudad de Cádiz, en la que vivo, podemos ver claramente que el principal problema, y en el que el ayuntamiento tiene puesto el foco desde hace años, es la movilidad. El gran volumen de vehículos que circula a todas horas por la ciudad emite gases que perjudican al medio ambiente y a nuestra propia salud. Además, la circulación se ve perjudicada por atascos y conductores buscando constantemente aparcamiento, lo que hace que se generen emisiones contaminantes que de no producirse estas situaciones no se generarían.

Con todo esto en mente he desarrollado mi sistema, para monitorizar los niveles de polución del aire y detectar de manera automática situaciones que perjudiquen al medio ambiente y a la salud de la gente, así como ayudar a la búsqueda de aparcamiento y a la detección de atascos, de forma que la actuación para solucionar estas situaciones sea casi inmediata.

1.2. Objetivos

El objetivo principal es diseñar e implementar una arquitectura software que permita tratar una serie de datos relacionados con el tráfico en la ciudad, el aparcamiento y la contaminación atmosférica, mediante un sistema, que lea los datos de una cola de mensajería y los procese en tiempo real con un motor de procesamiento de eventos complejos (CEP).

El sistema deberá ser persistente, almacenando el registro de eventos complejos detectados, de manera que se pueda acceder posteriormente a la información para actuar ante ella.

Además, se desarrollará una aplicación Android que permita visualizar situaciones de interés detectadas de una manera amigable para el usuario. Esta aplicación, aparte de mostrar los detalles de los eventos ocurridos, deberá ubicar geográficamente en un mapa plazas de aparcamientos libres y atascos que hay actualmente en la ciudad.

1.3. Alcance

La aplicación Mule podrá procesar cualquier dato que reciba a través del bróker RabbitMQ siempre que los datos sigan un formato JSON predefinido por el desarrollador de la aplicación. Aunque se utilizarán datos simulados para facilitar la prueba del sistema podrían usarse datos reales siempre que estos utilizasen el formato especificado o se adaptasen previamente a este. Dicho formato vendrá dado por el esquema de eventos que se registrará en el motor de procesamiento de eventos complejos.

Las situaciones de interés detectadas por el sistema se determinarán a través de la definición de una serie de patrones en el lenguaje EPL de Esper (el motor de procesamiento de eventos complejos usado). Si bien será un conjunto limitado de patrones, se podría añadir cualquier otro patrón que se desee usando el lenguaje EPL previamente mencionado para definir patrones cuyos eventos de entrada cumplan los esquemas previamente definidos.

La base de datos almacenará las situaciones de interés que se consultarán mediante una API REST que estará limitada a ofrecer las operaciones necesarias para obtener los datos a mostrar en la app; aunque puede extenderse con operaciones adicionales en caso de que sea necesario.

Por último, la app mostrará la situación relevante para el usuario final en base a las situaciones de interés detectadas por el motor de procesamiento de eventos complejos.

1.4. Visión general del documento

El resto del documento está estructurado de la siguiente forma:

- **Contexto del proyecto:** recoge todo lo que rodea al proyecto; los usuarios a los que está orientado el sistema, el modelo de ciclo de vida elegido, las tecnologías, lenguajes y herramientas utilizadas para desarrollarlo...
- **Planificación:** explica cómo ha sido el avance del proyecto a través del tiempo, describiendo las distintas iteraciones en las que se ha dividido el proceso de desarrollo.

- **Análisis del sistema:** describe el análisis llevado a cabo al empezar cada una de las iteraciones; los requisitos recogidos, los atributos de calidad que debe tener el producto, la gestión del presupuesto, etc.
- **Diseño:** describe la etapa de diseño llevada a cabo en cada una de las iteraciones del proceso de desarrollo: el diseño de la arquitectura física y lógica, el diseño de la base de datos, y el diseño de la aplicación de Android.
- **Implementación:** desarrolla cómo ha sido implementado cada componente del sistema, tanto de los patrones de eventos, como de la aplicación en Mule ESB explicando la implementación de cada flujo de datos en la misma y del servicio REST que publica, como de la aplicación para Android.
- **Entregables** del proyecto
- **Procesos de soporte y pruebas:** habla de los procesos de soporte que se han llevado a cabo durante el desarrollo del proyecto como la toma de decisiones, la gestión de riesgos, incluyendo también las pruebas realizadas al finalizar cada iteración.
- **Conclusiones y trabajo futuro**
- **Bibliografía**

Por último, el documento incluye tres anexos:

- **Manual de instalación** tanto de la aplicación para dispositivos Android como del resto del sistema, necesario para que la aplicación funcione correctamente.
- **Manual de usuario** de la aplicación.
- **Boceto de la aplicación**

1.5. Estandarización del documento

Este documento ha sido redactado en base al estándar ISO/IEC/IEEE 16326, sobre la gestión de proyectos de ingeniería software y de sistemas. En concreto, se ha intentado adaptar la estructura de este a lo descrito en el apartado 7 del estándar, que trata de los elementos de un plan de gestión de proyecto.

Hay apartados que carecían de sentido dado el contexto del actual proyecto, desarrollado como TFG ya que el estándar está orientado a proyectos de ingeniería software y de sistemas de grandes organizaciones. Apartados como la gestión de las subcontratas, del personal, etc. se han omitido.

Además, se han tomado decisiones como incluir la bibliografía al final en lugar de al principio como indica el estándar, favoreciendo la legibilidad del documento y no distanciándonos más aún de la estructura típica de un TFG.

1.6. Acrónimos

- **CEP** Complex Event Processing
- **EPL** Event Processing Language
- **ESB** Enterprise Service Bus
- **EDA** Event-driven architecture
- **SOA** Service Oriented Architecture
- **API** Application Programming Interface

- **AMQP** Advanced Message Queuing Protocol
- **MQTT** Message Queuing Telemetry Transport
- **STOMP** Simple Text Oriented Protocol
- **HTTP** Hypertext Transfer Protocol
- **REST** Representational State Transfer
- **JSON** JavaScript Object Notation
- **SQL** Structured Query Language
- **SDK** Software Development Kit
- **IDE** Integrated Development Environment
- **IoT** Internet of Things
- **SO** Sistema Operativo
- **APK** Android Application Package
- **URL** Uniform Resource Locator
- **TFG** Trabajo de Fin de Grado
- **UML** Unified Modeling Language

1.7. Definiciones

- **Procesamiento de Eventos Complejos (CEP):** Tecnología emergente que permite procesar, analizar y correlacionar grandes cantidades de eventos, para detectar y responder en tiempo real a situaciones críticas o relevantes del negocio.
- **Evento simple:** Un evento simple es una situación indivisible que ocurre en un instante concreto de tiempo.
- **Evento complejo:** Un evento complejo es una situación con mayor significado semántico que el evento simple, que nos provee de información comprensible y de gran valor, a partir de combinar y analizar otros eventos anteriormente sucedidos [3].
- **Patrón de evento:** Un patrón de evento es la definición de una serie de condiciones que al cumplirse da lugar a un evento complejo, a partir de uno o más eventos simples o de otro evento complejo.
- **Bus de Servicios Empresariales:** Es un elemento de integración de aplicaciones que permite trabajar en entornos heterogéneos, con diferentes tecnologías y protocolos. Combina servicios web, mensajería, transformación, encaminamiento y enriquecimiento de datos, etc.

2. Contexto del proyecto

En este capítulo se habla del contexto del proyecto, necesario de conocer antes de pasar a la planificación del mismo y por supuesto antes de comenzar el desarrollo. Se hará una descripción general del proyecto, se hará una relación de las características de los usuarios del sistema, se justificará el modelo de ciclo de vida elegido para el desarrollo, y se enumerarán, dando una breve explicación de en qué consisten, las tecnologías, los lenguajes y las herramientas seleccionadas para el proyecto.

2.1. Descripción general del proyecto

Se simularán una serie de datos relacionados con el tráfico, aparcamientos y polución de una ciudad mediante un simulador específico para el Internet de las cosas. Los datos simulados se enviarán a un bróker de mensajería RabbitMQ simulando que provienen de diversas fuentes. La aplicación Mule se suscribirá a los datos recibidos en el bróker mediante el protocolo AMQP 0.9.1; en dicha aplicación se programarán una serie de flujos que permitan la gestión y procesamiento de dichos datos para la detección de situaciones de interés a través de un motor CEP integrado en el sistema. Dichas situaciones detectadas se almacenarán en una base de datos para poder luego ser accedidas a través de una API REST. Dicha API será accedida desde la app Android desarrollada para mostrar la información que se considere relevante para los usuarios finales.

2.2. Características del usuario

Los usuarios de la aplicación se pueden dividir en dos grandes grupos:

- **Autoridades competentes:** Lo formarán los organismos públicos que necesiten la información del estado del aparcamiento, el tráfico y la polución del aire en la ciudad porque son los encargados de actuar ante las distintas situaciones perjudiciales para el medioambiente y la ciudadanía.

Se encargarán de disolver los atascos, de aumentar excepcionalmente la bolsa de aparcamientos o de limitar la entrada de coches en cierta zona si ya no quedan aparcamientos, de aplicar ciertos protocolos si se superan los niveles de polución recomendados, etc.

Se incluirá en este grupo al ayuntamiento de la ciudad, a la policía local, y a otras autoridades que tengan competencias en tráfico o medioambiente.

- **Resto de usuarios de la aplicación:** Lo formarán el resto de ciudadanos que decidan descargar la aplicación. Pueden tener muy diversos perfiles y usar la aplicación por distintas razones.

Algunos, con inquietudes ecológicas, la usarán para reducir sus emisiones y estar al corriente de los niveles de contaminación de la ciudad. Otros podrán usarla para su beneficio personal, para reducir el gasto de combustible y evitar perder el tiempo buscando aparcamiento o esperando en un atasco. Y habrá otros que la usen por los ambos motivos.

Lo bueno de la aplicación es que todos los usuarios, sea cual sea su perfil y sean cual sean las razones por las que use, contribuirán a conseguir el objetivo final de la misma, que es reducir los niveles de contaminación de la ciudad y mejorar la calidad de vida de la gente.

2.3. Modelo de ciclo de vida

Elegir el modelo de ciclo de vida es algo fundamental antes de emprender el desarrollo de un proyecto, ya que este definirá todos los procesos de un proyecto, desde el análisis de los requisitos hasta el final de su vida útil.

El modelo clásico en la gestión de proyectos es el modelo en cascada o modelo secuencial. En él primero, hasta que no se acaba una etapa del proceso de desarrollo no se empieza con la siguiente.

Esto tiene sus ventajas, como que la estructura del proyecto está clara, con los pasos que se deben dar de principio a fin bien definidos, es fácil de implementar y entender, y fácil de documentar.

Sin embargo, cuando hablamos de un proyecto de Ingeniería Software el modelo se muestra poco práctico, ya que los productos terminados no existen, deben estar continuamente actualizándose ya que la novedad, y la actualización y mejora continua del producto es lo que marca la diferencia entre un proyecto exitoso y un fracaso. Además, los requisitos de un proyecto de ingeniería software son cambiantes, no se pueden saber con exactitud desde un principio.

Por tanto, necesitamos un modelo fácilmente adaptable, que permita tener un producto entregable rápidamente para obtener retroalimentación y poder trabajar en mejorarlo, y así continuamente.

Por eso, el modelo de vida escogido es el modelo de vida iterativo incremental. Dividiremos el trabajo a realizar en diferentes iteraciones, y en cada iteración repetiremos todas las etapas del proceso de desarrollo: análisis de requisitos, diseño, implementación, pruebas, y documentación.

Al final de cada iteración tendremos un producto entregable, que aunque no es el producto final, aporta un nuevo valor, y puede ser evaluado para sacar nuevos requisitos para la siguiente iteración a partir de posibles mejoras, correcciones de errores, etc.

2.4. Tecnologías

En esta sección se enumerarán las tecnologías usadas durante el desarrollo del proyecto, explicando brevemente qué son, para qué sirven, y para qué han sido utilizadas en este proyecto.

2.4.1. Esper

Esper es un motor de procesamiento de eventos complejos (CEP) de código abierto basado en Java, que permite procesar y analizar grandes cantidades de eventos en tiempo real y reaccionar en base a ellos [4].

Ofrece un lenguaje de definición de patrones de eventos, Esper EPL.

2.4.2. Mule ESB

Mule ESB es un Enterprise Service Bus que permite integrar diferentes tecnologías y protocolos, permitiendo trabajar en entornos heterogéneos y combinar fuentes heterogéneas de información[5].

Integra los enfoques dirigidos por eventos (EDA) y orientado a servicios (SOA), siguiendo la arquitectura SOA 2.0, que permite que los servicios del sistema sean lanzados cuando sucede un evento.

En el proyecto desarrollado, lo utilizamos para recibir datos a través de una cola de trabajo de RabbitMQ, enviarlos en forma de eventos simples al motor CEP de Esper que generará eventos complejos que se enviarán a otra cola, se almacenarán en una base de datos, invocarán a servicios de una API, etc.

2.4.3. RabbitMQ

RabbitMQ[6] es un bróker de mensajería open source, ligero y de fácil despliegue. Aunque el principal protocolo y para el que fue originalmente implementado es AMQP, también soporta otros como MQTT, STOMP, HTTP, etc.

Lo usaremos para recibir los eventos simples simulados en una cola de entrada, desde la que recibirá los eventos nuestra aplicación. Tras procesar esos eventos, nuestra aplicación enviará los eventos complejos generados a una cola de salida también de RabbitMQ.

2.4.4. API REST

Una API REST es una interfaz que dos sistemas de computación utilizan para intercambiar información de manera segura a través de Internet[7].

La usaremos para conectar nuestra aplicación Android con el resto del sistema. Para ello, haremos uso de los métodos GET, POST, PUT, y DELETE del protocolo HTTP, enviando y recibiendo datos en formato JSON.

2.4.5. MySQL

MySQL[8] es un sistema de gestión de bases de datos relacional de código abierto, considerado el más popular del mundo.

Usaremos una base de datos MySQL en el proyecto para almacenar todos los eventos complejos generados en el motor Esper.

2.4.6. Android SDK

El SDK (Software Development Kit) de Android[9] nos provee de un conjunto de herramientas orientadas al desarrollo de aplicaciones en Android: un depurador de código, distintas bibliotecas, un emulador de dispositivos basado en QEMU, una extensa documentación, etc.

Haremos uso de estas características para desarrollar nuestra aplicación Android.

2.5. Lenguajes

En este apartado se hará una relación de los lenguajes usados, tanto lenguajes de programación como lenguajes de consultas y de procesamiento de eventos, explicando para qué son usados cada uno en nuestro sistema.

2.5.1. Java

Java [10] es el lenguaje utilizado en la mayor parte del proyecto, tanto para el servicio REST, como para la implementación de la aplicación de Android, como para desarrollar los componentes encargados de hacer funcionar el motor CEP de Esper (enviar los eventos al motor, desplegar en él esquemas y patrones, etc.).

2.5.2. SQL

SQL (Structured Query Language) es un lenguaje utilizado para administrar y recuperar información de sistemas de gestión de bases de datos relacionales como MySQL.

Ha sido utilizado para realizar las consultas de inserción y recuperación de los eventos complejos almacenados en nuestra base de datos, desde ciertos componentes de Mule ESB y desde el servicio REST.

2.5.3. Esper EPL

Es el lenguaje que ofrece Esper para declarar patrones de eventos, lo que hace es modificar y extender el lenguaje SQL para incluir operadores de patrones, ventanas de datos de eventos y temporales, etc.

Se ha utilizado para diseñar todos los esquemas y patrones de eventos usados en el proyecto.

2.6. Herramientas

En esta sección, se hablará de las herramientas utilizadas para el desarrollo de nuestro sistema, tanto los entornos de desarrollo, como el simulador usado para generar los eventos simples, como herramientas utilizadas para realizar el diseño de la aplicación, o herramientas orientadas a hacer pruebas.

2.6.1. Anypoint Studio

Anypoint Studio[11] es un IDE low-code que usando Mule ESB nos permite mediante “drag and drop” (arrastrar y soltar) desde una paleta de componentes, conectores, transformadores, etc. integrar y transformar datos de distintas fuentes, desarrollar APIs, conectarse a BBDD y muchas más funcionalidades.

Se ha usado la versión 6.6, ya que para la versión 7 no hay servidor gratuito disponible.

Es la herramienta central de nuestro proyecto, donde hemos creado los distintos flujos que se conectan entre sí y se encargan de desplegar los patrones y los esquemas en el motor Esper, recibir los eventos simples y enviarlos al motor, consumir los eventos complejos y almacenarlos en una base de datos, producir el servicio REST, etc.

2.6.2. Android Studio

Android Studio[9] es el IDE oficial para el desarrollo de aplicaciones para Android, basado en IntelliJ IDEA. Soporta Kotlin, Java y C++ como lenguajes de programación y usa Gradle como herramienta de construcción de código.

Este ha sido el entorno del desarrollo donde he desarrollado la app para Android.

2.6.3. Eclipse IDE

Eclipse es otro IDE, que aunque permite el desarrollo en muchos lenguajes de programación, se usa principalmente para el desarrollo de aplicaciones Java [12].

Se ha utilizado para desarrollar la API REST.

2.6.4. nITROGEN

nITROGEN es un sistema de generación de datos compatible con arquitecturas IoT desarrollado dentro del grupo de investigación UCASE de la UCA [13], [14].

Lo he usado para simular todos los eventos simples, que se envían a la cola de entrada del servidor RabbitMQ.

2.6.5. Balsamiq

Balsamiq es una herramienta para el prototipado de interfaces de usuario de baja fidelidad. Permite crear de manera rápida y con una curva de aprendizaje mínima bocetos de interfaces de usuario interactivas mediante “drag and drop” [15].

Ha sido utilizada para realizar el boceto de la aplicación móvil, que podemos ver en el Anexo C.

2.6.6. Postman

Postman [16] es una plataforma para implementar, usar y probar APIs. Simplifica cada etapa del desarrollo de una API REST, permitiendo generar colecciones de peticiones, implementar prueba, monitorizar el estado de una API, etc.

La hemos utilizado para realizar pruebas sobre la API implementada, como hablaremos en el apartado 8.3.3.

3. Planificación

En este capítulo se describirá la planificación seguida durante el desarrollo del proyecto.

Como ya se comentó en el apartado 2.3, para el proyecto se ha utilizado un ciclo de vida iterativo incremental.

Esto hace que el proceso de desarrollo se haya dividido en una serie de iteraciones, y en cada una de las iteraciones hayamos pasado por todas las etapas del desarrollo de un producto software: análisis, diseño, implementación y pruebas. Además, paralelamente a la ejecución de cada iteración, se ha ido documentando todo lo realizado en la presente memoria.

En el apartado 3.1 se explicará la iniciación del proyecto, cómo surgió la idea del proyecto, cómo se dio la primera aproximación a las tecnologías usadas, etc. En el apartado 3.2 se hablará de las distintas iteraciones del proyecto (excluyendo la primera aproximación).

Por último, tendremos un diagrama de Gantt en el apartado 3.3 que muestra una visión general del tiempo utilizado para cada iteración y cada etapa dentro de esa iteración.

3.1. Iniciación del proyecto

La primera aproximación a este proyecto se dio durante la asignatura de Ingeniería de Sistemas de Información. En la asignatura se aprenden la mayoría de las tecnologías utilizadas en este proyecto: el funcionamiento del motor Esper y el lenguaje Esper EPL, y el desarrollo de una aplicación mediante Mule ESB.

Como proyecto de final de la asignatura se desarrolló una aplicación usando Mule ESB, implementando una serie de patrones en Esper EPL, creando un pequeño servicio REST que cuenta los eventos que ocurren y almacenando en una base de datos todos los eventos complejos generados.

Tras reunirme con los tutores del presente TFG, decidimos que una buena opción era usar lo aprendido en la asignatura para hacer un proyecto parecido pero ampliamente extendido y pulido, dotado de mayor complejidad, y complementado con una aplicación para dispositivos Android que permitiera a los usuarios acceder a la información obtenida de los eventos complejos de una forma amigable y comprensible.

3.2. Iteraciones del proceso de desarrollo

Como ya se explicó y se justificó en el apartado 2.3, para el desarrollo del proyecto se decidió usar un modelo de ciclo de vida iterativo incremental, de modo que en cada iteración volviéramos a pasar por todas las fases del proceso de desarrollo, obteniendo al final de cada iteración un producto con un valor mayor que en la anterior.

En esta sección, se enumerarán las distintas iteraciones en las que se ha desarrollado el sistema, explicando qué se ha realizado en cada una, y hablando de cada etapa del proceso de desarrollo llevadas a cabo en cada iteración. Es decir, en cada iteración se hablará de análisis, diseño, implementación y pruebas.

3.2.1. Diseño de patrones

Una vez decidido el tema del proyecto, se procedió a pensar en los patrones a implementar. Tenían que ser patrones que nos ayudaran en la tarea de monitorizar el aparcamiento, el tráfico y la polución en la ciudades, y detectaran situaciones de interés relacionadas con estos temas.

Primero se hizo una relación de las situaciones de interés que se querían detectar, después de esta relación se diseñaron los patrones en pseudocódigo, para pasar finalmente a implementarlos usando el lenguaje Esper EPL, guardando cada patrón en un archivo .epl.

Además de los patrones para detectar los eventos complejos, se crearon por supuesto los esquemas que debían seguir los eventos simples que recibirá el motor de nuestra aplicación.

Se decidió qué parámetros debía incluir cada evento simple, tanto el de tráfico, como el de calidad de aire, como el de aparcamiento, de manera que fuera suficiente para detectar después todas las situaciones de interés requeridas a partir de ellos. Una vez hecho esto, se diseñaron en pseudocódigo y se implementaron también en EsperEPL.

Una vez implementados los patrones y esquemas, se hicieron pruebas en Esper Notebook, como desarrollaremos después en el apartado 8.3.1 de la documentación, para comprobar que eran correctos sintácticamente, y que nos ayudaban a conseguir los requisitos inicialmente identificados.

3.2.2. Implementación de los flujos en Mule

A continuación, se procedió a desarrollar lo que sería la parte central del proyecto, la aplicación desarrollada mediante Mule ESB.

Primero, se pensó en qué flujos debería tener la aplicación y qué funciones debía cumplir cada uno de ellos. Después se diseñó la estructura de todos los flujos, arrastrando desde la paleta de Anypoint Studio todos los componentes necesarios.

Una vez diseñada la estructura de los flujos, se procedió a configurar todos los componentes correctamente: configurando la conexión con la cola de entrada y la cola de salida de RabbitMQ, configurando la creación del servicio REST, configurando el envío de los eventos simples al motor de Esper y el consumo en otro flujo de los eventos complejos generados en él, configurando la conexión con la base de datos, etc.

En esta iteración, la API apenas incluía tres métodos: uno para saber si estaba en funcionamiento el servicio REST, otro para sumar 1 al número de eventos complejos cuando se generara uno y otro que devolviera el número de eventos. Además, en la base de datos se almacenaban todos los eventos, sin filtrar según fuera el tipo de estos. Todo esto se mejoró ampliamente, a la vez que se añadieron nuevos componentes para refinar todo el proceso, en la iteración descrita en el apartado 3.2.4.

Al acabar la implementación se probó todo el sistema: los eventos simples se reciben correctamente, los esquemas y patrones se despliegan correctamente, los eventos complejos se generan cuando deben generarse y se almacenan correctamente.

3.2.3. Desarrollo de la estructura de App

A continuación, se analizaron los requisitos de las funcionalidades que tendría la aplicación de Android. En base a eso, se diseñó cómo sería la navegación por la aplicación, y cómo sería la interfaz, realizando algún boceto, y eligiendo características que tendría el diseño como la paleta de colores que usaría.

Acabado todo esto, se creó un nuevo proyecto en Android Studio y se comenzaron a implementar todas las Activities que tendría la aplicación, el Navigation Drawer que se decidió que fuera un menú lateral y que permitiera navegar entre una pantalla y otra.

Sobre todo, este primer desarrollo estuvo centrado en el diseño de los layouts de las distintas pantallas de la aplicación, el desarrollo de la pantalla de bienvenida y el diseño de la navegación entre una pantalla y otra. Acabada esta iteración, nos quedó una aplicación completamente navegable, en la que se podía apreciar información de la aplicación como el logo, el icono, el objetivo, pero que no tenía ninguna funcionalidad ya que no estaba implementada la conexión con la API REST ni las tareas asíncronas que se encargarían de formatear y presentar los datos recibidos desde la API. Todo esto se mejoraría y ampliaría, completando la funcionalidad de la aplicación, en la iteración descrita en el apartado 3.2.5.

Por supuesto, la aplicación se probó, comprobando que fuera completamente navegable, y que el apartado visual fuera el esperado desde un principio.

3.2.4. Mejora del proyecto Mule y desarrollo de la API REST

En esta iteración se mejoró el proyecto desarrollado con Mule ESB, y se crearon los métodos de la API que nos servirán después para obtener los eventos complejos desde la app de Android.

Lo primero que se hizo fue analizar cómo se podía mejorar lo ya implementado, se decidió añadir un componente que comprobara si los eventos simples que se recibían en el sistema cumpliesen con el formato deseado, almacenar los eventos complejos en tablas distintas de la base de datos, según el tema de estos.

Además, se estudió qué métodos se debían implementar como parte del servicio REST para proporcionar a la aplicación en Android toda la información necesaria.

Hecho esto, se modificó la estructura de los flujos de Mule ESB, añadiendo nuevos componentes, y se procedió a implementar los nuevos métodos del servicio REST.

Al acabar la implementación, se ejecutó el proyecto Mule para ver que todo lo que se había modificado funcionaba correctamente, y mediante Postman se comprobó que los métodos de la API se comportaban como debían.

3.2.5. Mejora de la aplicación Android y conexión con la API

En esta fase del proyecto se mejoró la aplicación de Android, dotándola de funcionalidad completa y estableciendo conexión con la API REST antes implementada. De esta manera, la aplicación accede en tiempo real a los eventos complejos generados en el motor CEP, y los presenta al usuario.

Primero, se actualizará el análisis de los requisitos de la aplicación que se hizo en el apartado 3.2.3, de manera que lo que se pretende mostrar al usuario esté en sintonía con la información que recibiremos desde la API REST.

También actualizamos el diseño de las distintas pantallas para incluir la información que se recibirá desde la API, y se implementarán todas las tareas asíncronas encargadas de recuperar la información realizando una petición al servicio REST y mostrar la información de forma que sea comprensible al usuario.

Finalizada toda la implementación, realizaremos pruebas sobre el sistema completo, para comprobar que la información se muestra correctamente y que la conexión entre todos los componentes del sistema, incluida la aplicación Android, se realiza correctamente.

3.3. Diagrama de Gantt

El diagrama de Gantt (ver Figura 1) muestra toda la progresión a través del tiempo del proceso de desarrollo, mostrando tanto la duración de las distintas iteraciones del proceso, como de las diferentes etapas dentro de cada iteración.

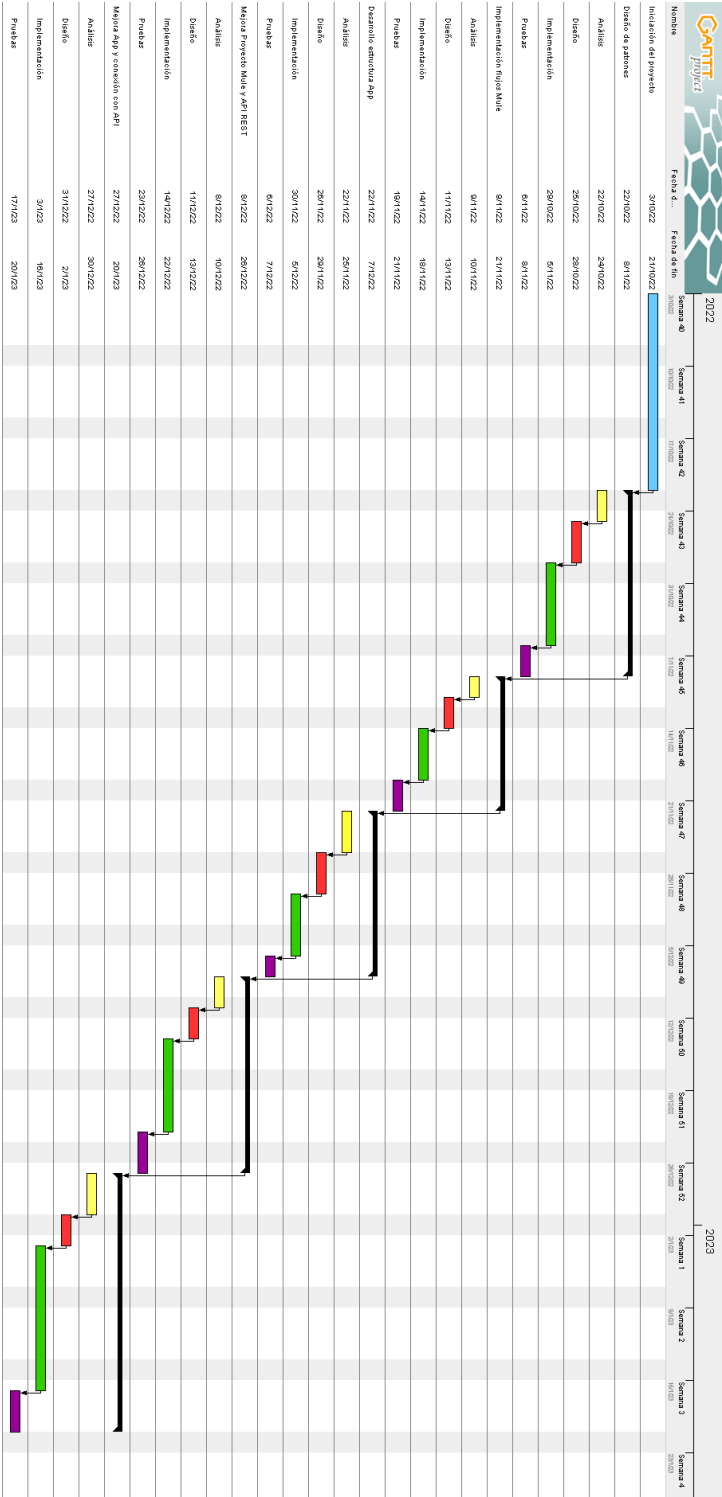


Figura 1. Diagrama de Gantt del proyecto

4. Análisis del sistema

Durante este capítulo, se realizará un análisis del sistema que vamos a desarrollar. Aunque haremos un exhaustivo análisis del sistema al empezar la primera iteración, al usar un ciclo de vida iterativo incremental, al inicio de cada iteración volveremos a analizar el sistema, extrayendo nuevos requisitos y actualizando los que sean necesarios, y actualizando el presupuesto si es necesario.

En el apartado 4.1 tendremos la especificación de requisitos, describiendo los requisitos de información y los funcionales, modelando diagramas de casos de uso y de secuencia, haciendo una descripción de los escenarios de los casos de uso, etc.

Posteriormente, en el apartado 4.2 hablaremos de la garantía de calidad de nuestro producto, es decir, de los requisitos no funcionales que debemos tener en cuenta durante el proceso de desarrollo (seguridad, operabilidad, fiabilidad, mantenibilidad, compatibilidad...)

Por último tendremos en el apartado 4.3 la gestión del presupuesto necesario para desarrollar el proyecto, hablando de las distintas partidas y el coste de los recursos necesarios para llevar a cabo el proyecto.

4.1. Especificación de requisitos

Se realizará a continuación un análisis de los requisitos del sistema, en concreto de:

- Requisitos de información, que describen qué información deberá almacenar el sistema para lograr los objetivos marcados.
- Requisitos funcionales, que recogen las funcionalidades que debe tener el sistema si quiere cumplir sus objetivos.

Además, se describirán algunos casos de uso y se modelarán diagramas de casos de uso y de secuencia.

4.1.1. Requisitos de información

- El sistema debe almacenar en una base de datos los eventos complejos generados.
- Los eventos complejos deben ser almacenados en una tabla distinta según el evento simple con el que estén relacionados; es decir, los eventos complejos relacionados con el aparcamiento en una tabla, los relacionados con el tráfico en otra diferente, e igual con los relacionados con la polución del aire.
- El sistema debe almacenar las coordenadas de cada sensor (realmente no existen sensores ya que simulamos los eventos, pero en un entorno real sí que contaríamos con ellos).

4.1.2. Requisitos funcionales

- El sistema debe ser capaz de detectar eventos simples que recibe desde una cola de mensajes de RabbitMQ.
- El sistema debe comparar los eventos simples con una serie de patrones desplegados en el motor CEP y generar así eventos complejos, con mayor significado semántico que los primeros.

- El sistema debe enviar los eventos complejos a otra cola de mensajes de RabbitMQ y almacenarlos en una base de datos.
- El sistema debe ofrecer una API REST, que servirá para acceder a los eventos complejos generados desde la aplicación de Android.
- El sistema debe ofrecer una aplicación para Android que permita a los usuarios de la aplicación acceder a los datos sobre el tráfico, el aparcamiento y la polución de la ciudad en tiempo real.
- La aplicación de Android debe mostrar los eventos de forma amigable para el usuario, así como información detallada de cada evento.
- La aplicación de Android debe mostrar en un mapa de la ciudad los aparcamientos que se encuentran libres actualmente, permitiendo, al pulsar en el aparcamiento que se encuentra libre, ver más información sobre la plaza de aparcamiento e iniciar una ruta hacia ella mediante Google Maps.
- La aplicación de Android debe también mostrar en un mapa de Cádiz los atascos que existen en la ciudad, también mostrando más información y permitiendo establecer una ruta hacia el atasco mediante la aplicación de Google Maps.

4.1.3. Diagrama de casos de uso

En UML, un diagrama de casos de uso es una forma de representar gráficamente los distintos casos de uso del sistema, así como las relaciones entre ellos y los actores del sistema.

En nuestro caso, realizaremos un diagrama de casos de uso de la aplicación Android, como podemos ver en la Figura 2.

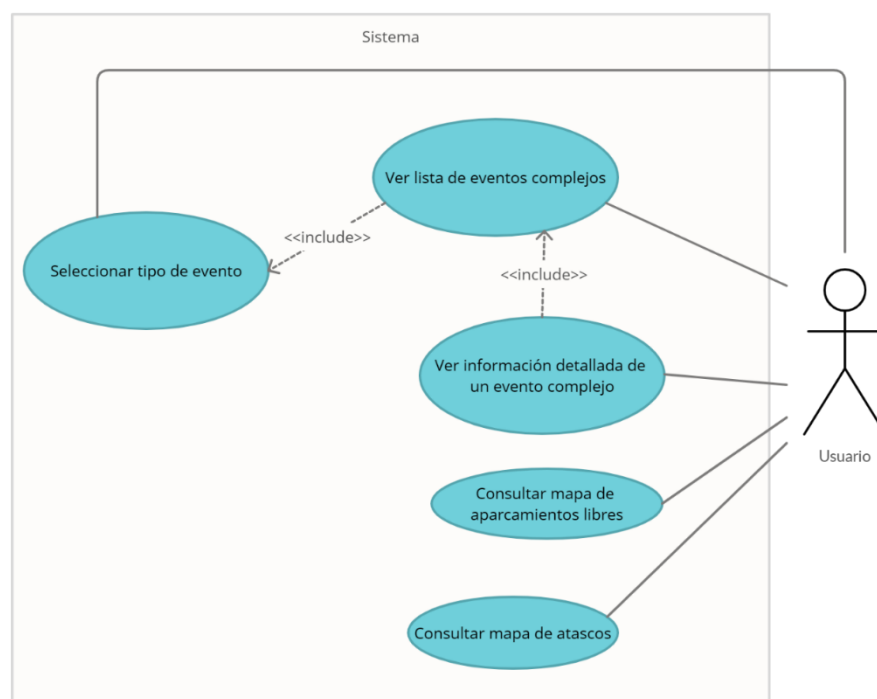


Figura 2. Diagrama de casos de uso App

4.1.4. Descripción de casos de uso

En este apartado describiremos en profundidad los casos de uso que se muestran en el diagrama, indicando los actores, las precondiciones y postcondiciones que deben de cumplirse para el correcto funcionamiento del mismo, y el escenario principal, además de ciertas extensiones de los casos de uso.

SELECCIONAR TIPO DE EVENTO

En la Tabla 1 podemos ver la descripción del caso de uso en el que un usuario elige uno de los tres tipos de eventos existentes.

Nombre	RF-01. Seleccionar tipo de evento.
Descripción	El usuario elige el tipo de evento (aparcamiento, tráfico o polución) del que desea ver la lista de eventos complejos.
Actores	Usuario.
Precondiciones	Ninguna.
Postcondiciones	El sistema se encargará de cargar la lista completa de los eventos que del tipo escogido.
Escenario principal	<ol style="list-style-type: none">1. El usuario inicia la aplicación.2. El usuario abre el menú lateral pulsando en el icono que lo abre.3. El sistema muestra las partes del sistema a las que se puede navegar desde el menú.4. El usuario pulsa en “Eventos”.5. La aplicación le muestra una pantalla con tres opciones: “Aparcamiento”, “Tráfico” y “Polución”.6. El usuario elige el tipo de evento deseado.

Tabla 1. RF-01 - Seleccionar tipo de evento

VER LISTA DE EVENTOS COMPLEJOS

La Tabla 2 muestra la descripción del caso de uso en el que se le muestra al usuario la lista de eventos complejos del tipo que haya elegido.

Nombre	RF-02. Ver lista de eventos complejos.
Descripción	El usuario elige el tipo de evento y la aplicación le muestra la lista de eventos de ese tipo que han sucedido.
Actores	Usuario.

Precondiciones	El servicio REST debe estar en funcionamiento y la aplicación debe poder enviar peticiones al mismo sin problemas.
Postcondiciones	El sistema mostrará al usuario la lista completa de eventos del tipo que eligió, indicando el nombre del evento y la fecha y hora a la que sucedió.
Escenario principal	<ol style="list-style-type: none"> 1. El usuario inicia la aplicación. 2. El usuario abre el menú lateral pulsando en el icono que lo abre. 3. El sistema muestra las partes del sistema a las que se puede navegar desde el menú. 4. El usuario pulsa en “Eventos”. 5. La aplicación le muestra una pantalla con tres opciones: “Aparcamiento”, “Tráfico” y “Polución”. 6. El usuario elige el tipo de evento deseado. 7. El sistema muestra la lista de eventos del tipo elegido por el usuario. De cada evento le muestra el nombre, la fecha y la hora a la que tuvo lugar, y un botón para ver más información del mismo.

Tabla 2. RF-02 - Ver lista de eventos complejos

VER DETALLES DE EVENTO

En la Tabla 3 se describe el caso de uso en el que se le muestra a un usuario los detalles del evento en el que ha pulsado.

Nombre	RF-03. Ver información detallada de un evento complejo.
Descripción	La aplicación muestra al usuario información detallada de un evento.
Actores	Usuario.
Precondiciones	El servicio REST debe estar en funcionamiento y la aplicación debe poder enviar peticiones al mismo sin problemas.
Postcondiciones	Se le muestra al usuario la información detallada sobre un evento en concreto de forma amigable y comprensible.
Escenario principal	<ol style="list-style-type: none"> 1. El usuario inicia la aplicación. 2. El usuario abre el menú lateral pulsando en el icono que lo abre.

	<ol style="list-style-type: none"> 3. El sistema muestra las partes del sistema a las que se puede navegar desde el menú. 4. El usuario pulsa en “Eventos”. 5. La aplicación le muestra una pantalla con tres opciones: “Aparcamiento”, “Tráfico” y “Polución”. 6. El usuario elige el tipo de evento deseado. 7. El sistema muestra la lista de eventos del tipo elegido por el usuario. De cada evento le muestra el nombre, la fecha y la hora a la que tuvo lugar, y un botón para ver más información del mismo. 8. El usuario pulsa en el botón para acceder a los detalles de un evento en concreto. 9. La aplicación muestra toda la información disponible del evento.
--	--

Tabla 3. RF-03 - Ver información detallada de un evento complejo

CONSULTAR MAPA DE APARCAMIENTOS LIBRES

La Tabla 4 describe el caso de uso en el que se le presentan a un usuario en un mapa los aparcamientos que se encuentran libres.

Nombre	RF-04. Consultar mapa de aparcamientos libres.
Descripción	La aplicación muestra un mapa de Google al usuario, donde muestra los aparcamientos que se encuentran libres actualmente.
Actores	Usuario.
Precondiciones	El servicio REST debe estar en funcionamiento y la aplicación debe poder enviar peticiones al mismo sin problemas.
Postcondiciones	Se mostrará al usuario un mapa que muestre todas las plazas de aparcamiento libres en la ciudad.
Escenario principal	<ol style="list-style-type: none"> 1. El usuario inicia la aplicación. 2. El usuario abre el menú lateral pulsando en el icono que lo abre. 3. El sistema muestra las partes del sistema a las que se puede navegar desde el menú. 4. El usuario pulsa en “Aparcamientos”.

	5. El sistema carga el mapa y se lo muestra al usuario, indicando con marcadores los aparcamientos que existen libres en la ciudad.
Extensiones	6. El usuario pulsa sobre un marcador que indica un aparcamiento libre. 6.1 La aplicación muestra un cuadro sobre el marcador con más información sobre la plaza de aparcamiento. 6.2 El usuario pulsa en el icono para ir hasta la plaza de aparcamiento libre. 6.3 Se saldrá de la aplicación y se abrirá Google Maps y se le indicará al usuario la ruta hasta el aparcamiento libre.

Tabla 4. RF-04 - Consultar mapa de aparcamientos libres

CONSULTAR MAPA DE ATASCOS

En la Tabla 5 podemos ver una descripción del caso de uso en el que se le muestran a un usuario sobre un mapa los atascos que existen actualmente en la ciudad.

Nombre	RF-05. Consultar mapa de atascos.
Descripción	La aplicación muestra un mapa de Google al usuario, donde muestra los atascos que existen en tiempo real.
Actores	Usuario.
Precondiciones	El servicio REST debe estar en funcionamiento y la aplicación debe poder enviar peticiones al mismo sin problemas.
Postcondiciones	Se mostrará al usuario un mapa que muestre todos los atascos que existen en la ciudad en tiempo real.
Escenario principal	1. El usuario inicia la aplicación. 2. El usuario abre el menú lateral pulsando en el icono que lo abre. 3. El sistema muestra las partes del sistema a las que se puede navegar desde el menú. 4. El usuario pulsa en “Atascos”. 5. El sistema carga el mapa y se lo muestra al usuario, indicando con marcadores los atascos que existen libres en la ciudad.
Extensiones	6. El usuario pulsa sobre un marcador que indica un aparcamiento libre.

	<p>6.1 La aplicación muestra un cuadro sobre el marcador con más información sobre el atasco.</p> <p>6.2 El usuario pulsa en el icono para ir hasta el atasco (esto tiene sentido por ejemplo para a policía que deba ir a devolver el tráfico a la normalidad o para una ambulancia si la causa del atasco es un accidente).</p> <p>6.3 Se saldrá de la aplicación y se abrirá Google Maps y se le indicará al usuario la ruta hasta la zona del atasco.</p>
--	---

Tabla 5. RF-05 - Consultar mapa de atascos

4.1.5. Diagramas de secuencia

En este apartado recoge distintos diagramas de secuencia que modelan la interacción entre los distintos objetos de un sistema mediante líneas de vidas o mensajes. En concreto, se han modelado tres diagramas de secuencia. El primero (ver Figura 3) recoge la interacción entre los componentes de la aplicación de Mule ESB, el segundo (ver Figura 4) la interacción necesaria para ver los detalles de un evento, y el tercero, que podemos observar en la Figura 5, la interacción necesaria entre los distintos componentes para mostrar el mapa de aparcamientos libres.

APLICACIÓN MULE ESB

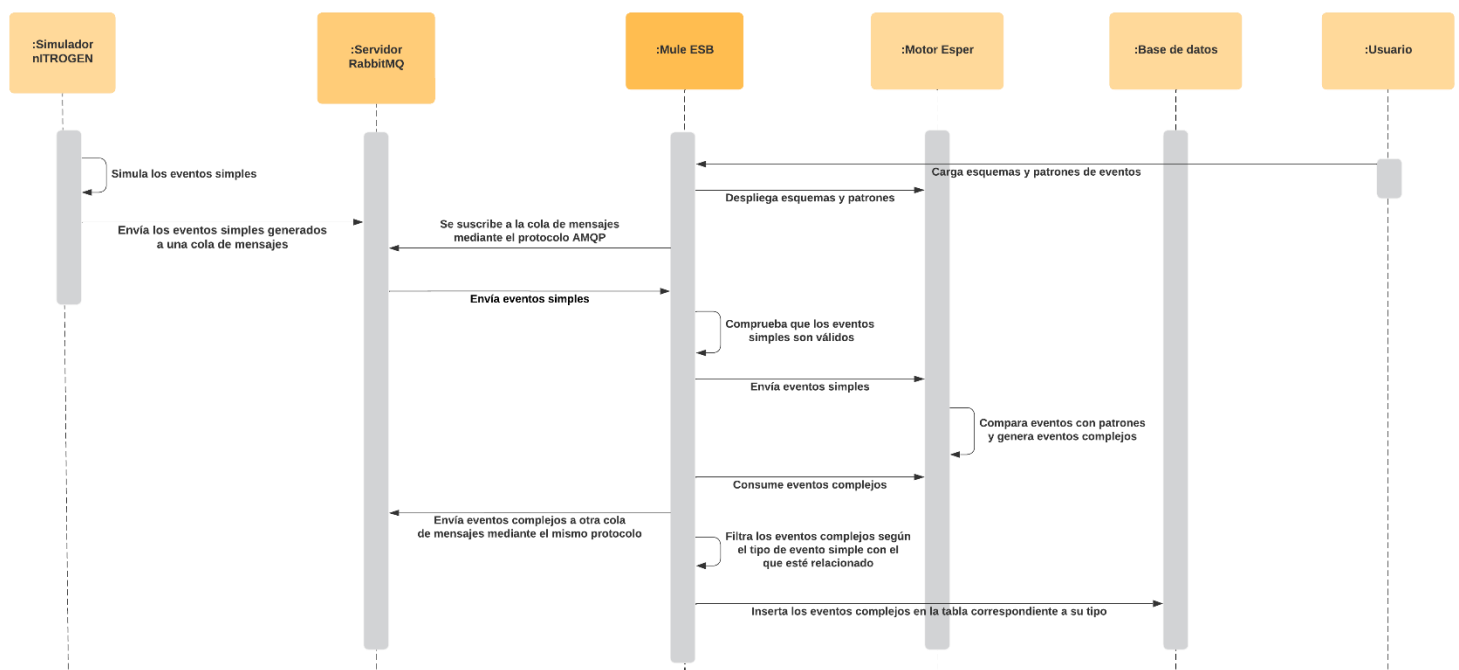


Figura 3. Diagrama de secuencia "Mule ESB"

VER DETALLES DE EVENTO

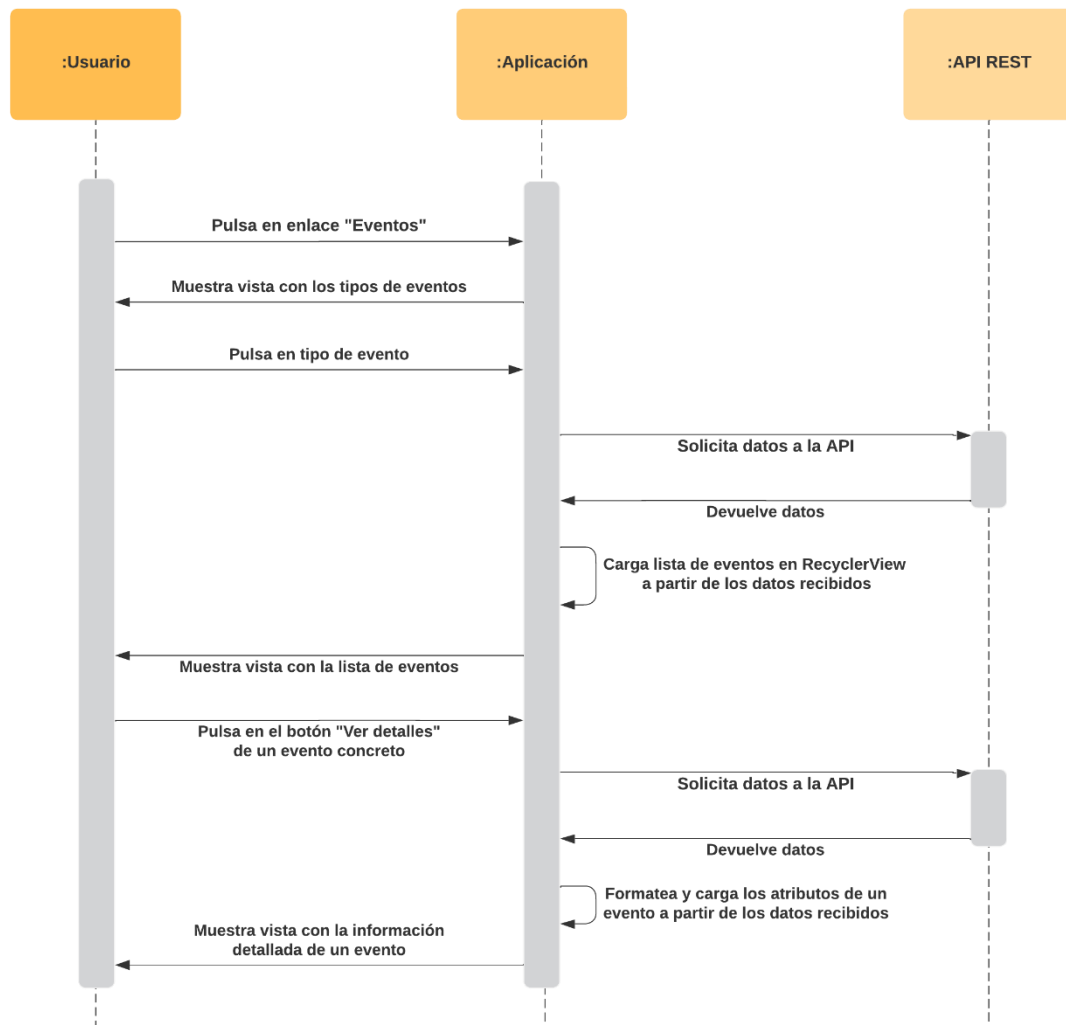


Figura 4. Diagrama de secuencia "Ver detalles de evento"

VER MAPA DE APARCAMIENTOS LIBRES

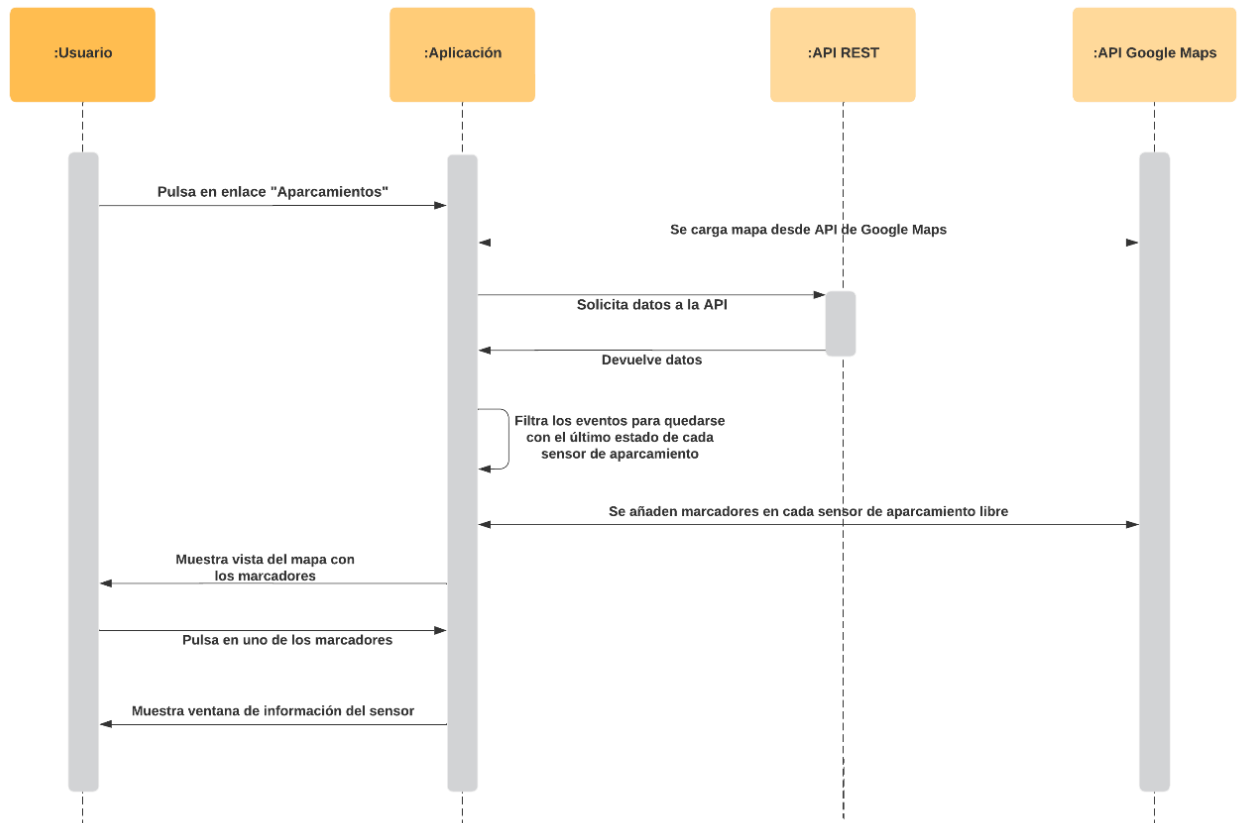


Figura 5. Diagrama de secuencia "Ver mapa de aparcamientos libres"

4.2. Garantía de calidad

En este apartado se recogerán los distintos requisitos no funcionales que garantiza la calidad del sistema. Es decir, el sistema no debe simplemente cumplir las funcionalidades anteriormente especificadas, sino que también debe tener ciertos atributos de calidad.

4.2.1. Seguridad

- El sistema deberá asegurarse de que todos los eventos que recibe siguen el formato esperado y en caso contrario descartarlos y no enviarlos al motor CEP.

4.2.2. Interoperabilidad

- El sistema deberá permitir la interoperabilidad a través de una plataforma intermedia de integración de aplicaciones, en nuestro caso Mule ESB.
- El sistema deberá ser nodo consumidor de una cola de mensajería desde la que recibirá datos, al igual que nodo productor de otra cola a la que los mandará.
- El sistema permitirá conectarse desde un dispositivo Android a través de la aplicación a un servicio REST publicado en nuestra aplicación de Mule.

4.2.3. Operabilidad

- La aplicación Android deberá hablar en términos familiares para el usuario, utilizando metáforas visuales conocidas para los iconos y botones.
- La aplicación deberá seguir un estilo consistente en cuanto a los colores usados y el texto mostrado.
- La aplicación deberá tener un diseño minimalista, evitando incluir información irrelevante en la interfaz de usuario.

4.2.4. Transferibilidad

- El sistema deberá desarrollarse con herramientas portables: usando lenguajes de programación portables como Java, lenguajes de consulta comunes como SQL, etc.
- La aplicación Android deberá adaptar la disposición y el tamaño de los elementos en pantalla a la resolución de pantalla del dispositivo en que se use.

4.2.5. Eficiencia

- La interfaz de usuario debe actualizarse de forma asíncrona, para disminuir los tiempos de carga.
- Los tiempos de carga de la aplicación deben ser razonables para que estos no perjudiquen negativamente a la experiencia de usuario.

4.2.6. Mantenibilidad

- El código de la aplicación debe ser modular y fácilmente extensible, ya que una aplicación Android necesita estar continuamente actualizándose.
- El estilo de programación debe ser consistente, y seguir las convenciones de programación más extendidas.
- El código debe estar debidamente documentado, para que sea comprensible para cualquier programador que no haya participado del proceso de desarrollo.

4.3. Gestión del presupuesto

Este proyecto no es un proyecto real de una organización, sino un proyecto realizado como TFG. Por tanto, no es posible llevar un registro de las horas exactas dedicadas al proyecto, ni de otro tipo de costes.

Sin embargo, en este apartado se pretende hacer una estimación de lo que sería el coste del proyecto realizado en un entorno real de una organización.

Hemos dividido el presupuesto en tres partidas principales:

- Presupuesto del hardware (ver Tabla 6)
- Presupuesto del software
- Presupuesto de mano de obra (ver Tabla 7)

Por último, en la Tabla 8, podemos observar el presupuesto total, sumando las tres partidas mencionadas, que será la cantidad de dinero necesaria para llevar a cabo el proyecto.

PRESUPUESTO HARDWARE

Artículo	Unidades	P.V.P	Precio total
MSI GF75 Thin 9SC Características: <ul style="list-style-type: none">• Intel Core i7-9750H• NVIDIA GeForce GTX 1650• RAM 16 GB• 512 GB SSD• 17.3"	1	1.049€	1.049€
Consumo eléctrico	0,13 kWh	0,127 €/kWh x 330 h	5,45€
Fibra óptica 1Gbps Vodafone	4 meses	44,30€/mes	177.2€
			1.231,65€

Tabla 6. Presupuesto hardware

PRESUPUESTO SOFTWARE

Todo el software utilizado es gratuito, ya sea porque es de código abierto como RabbitMQ o Esper, porque sea gratuito como Android Studio, porque se han usado versiones gratuitas como la Community Edition de Mule ESB. Por tanto el coste total en software sería de 0€.

PRESUPUESTO MANO DE OBRA

Personal	Cantidad	Horas trabajadas	Salario bruto/hora	Precio total
Ingeniero informático junior	1	470 h	11,35€/h	5.334,5€
				3.745,5€

Tabla 7. Presupuesto mano de obra

PRESUPUESTO TOTAL

Partida	Total
Hardware	1.231,65€
Software	0€
Personal	5.334,5€
	6.566,15

Tabla 8. Presupuesto total

5. Diseño

El diseño es la fase siguiente al análisis de requisitos y previa a la implementación. Se parte de los requisitos que se han identificado y se define y estructura el sistema para que haga lo que se espera de él, con el detalle suficiente para que posteriormente quede claro todo lo que se debe implementar.

Con este objetivo, en este capítulo describiremos tanto la arquitectura física (apartado 5.1) como la arquitectura lógica del sistema (apartado 5.2), el esquema de la base de datos (apartado 5.3), y el diseño de la aplicación para dispositivos Android: tanto de la navegación (apartado 5.4.1) y de la interfaz de usuario (apartado 5.4.2).

5.1. Arquitectura física

La arquitectura física o arquitectura hardware describe los componentes físicos que forman el sistema, y las relaciones entre ellos y el exterior. La arquitectura física del sistema a desarrollar no tiene demasiada complejidad. Hemos decidido dividirla en tres módulos, como podemos observar en la Figura 6 , un módulo de entrada, un módulo de procesamiento y uno de salida.

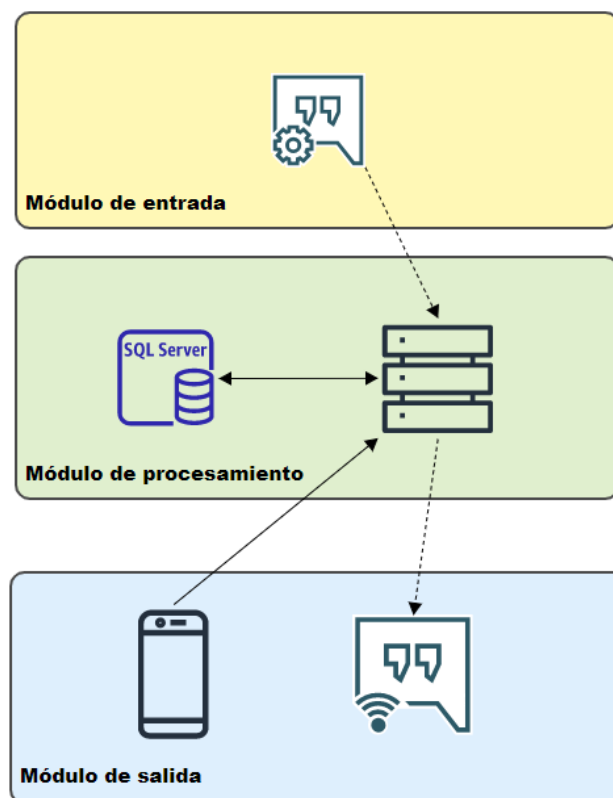


Figura 6. Arquitectura física

5.1.1. Módulo de entrada

El módulo de entrada se dedica a proporcionar los datos de entrada al sistema, en nuestro caso, los eventos simples.

En nuestro sistema, esto se hace mediante dos componentes distintos conectados entre sí:

- **Simulador nITROGEN:** Se encarga de generar los datos que después recibirá la aplicación, generando datos aleatorios dentro de unos rangos definidos por nosotros, y generando mensajes con un formato especificado, que después se envían a la cola de mensajería de RabbitMQ a la que se conectan.

Si la aplicación estuviera desplegada en un entorno real y los datos no fueran simulados, el lugar del simulador en el módulo de entrada de datos lo ocuparían los distintos tipos de sensores desde los que recibiríamos los eventos simples en nuestra cola de RabbitMQ.

- **Servidor de RabbitMQ:** Este servidor, como veremos después, forma parte tanto del módulo de entrada como de salida. En cuanto a la función que realiza en el módulo de entrada, este servidor cuenta con una cola de mensajería en la que el simulador publica como mensajes los eventos simples simulados y a la que después se suscribe la aplicación de Mule ESB para obtener estos eventos.

5.1.2. Módulo de procesamiento

Es la parte central del proyecto, toma los datos de entrada, los procesa, los almacena y los pone a disposición de la aplicación móvil.

En nuestro sistema lo formarían el servidor donde se ejecuta la aplicación de Mule ESB que contiene la mayor parte de la lógica del sistema, y el servidor MySQL donde se encuentra la base de datos que almacena los eventos complejos.

5.1.3. Módulo de salida

El módulo de salida es lo que recibe los datos finales, ya transformados y procesados por el módulo de procesamiento.

En nuestro caso lo formarían dos componentes:

- **Servidor de RabbitMQ:** Como ya hemos dicho antes, este servidor también forma parte del módulo de salida, ya que el módulo de procesamiento envía los datos ya procesados (eventos complejos) a una cola de mensajes de salida que se encuentra en este servidor.
- **Dispositivo Android:** Es, podemos decir, el dispositivo final de nuestro sistema. Se conecta con el módulo de procesamiento para recibir los datos ya procesados, y presentar la información al usuario. Gracias a la capa de abstracción que proporciona el sistema operativo, cualquier dispositivo que tenga una versión de Android mayor a 8.0 podrá usar la aplicación sin ningún inconveniente.

5.2. Arquitectura lógica

La arquitectura lógica o arquitectura software describe los componentes software que conforman un sistema y cómo se interconectan entre ellos para permitir el intercambio de información entre unos y otros.

Como podemos ver en la Figura 7, en la arquitectura lógica podemos observar los mismos módulos que en la arquitectura física, uno de entrada, uno de procesamiento, y otro de salida.

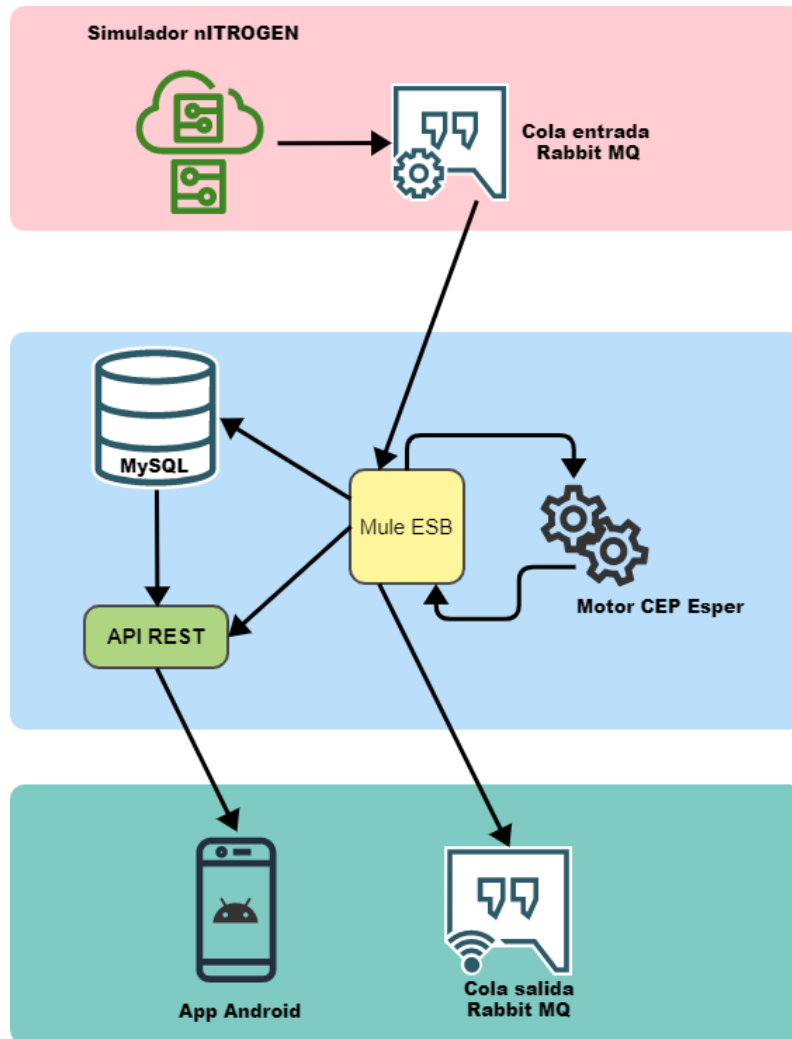


Figura 7. Arquitectura lógica

5.2.1. Módulo de entrada

El módulo de entrada es el que genera los datos que después recibe el módulo de procesamiento, en concreto el elemento central de este, nuestra aplicación Mule.

En nuestro sistema, esto se hace mediante dos componentes distintos conectados entre sí:

- **Simulador nITROGEN:** Como ya comentamos en la arquitectura física, contamos con un simulador, que genera los datos de manera aleatoria, siempre dentro de unos rangos y siguiendo un formato definido por el programador. En el simulador tenemos tres conectores, cada uno simulará un tipo de evento simple y se conectará a la cola de RabbitMQ donde los publicará.

Si contáramos con sensores, en lugar del simulador tendríamos aquí el componente software del sensor que mediante un protocolo de conexión inalámbrica enviaría los datos a la cola de RabbitMQ.

- **Cola de entrada de RabbitMQ:** Esta cola de mensajería recibe los eventos simples en forma de mensajes desde el simulador. Posteriormente, la aplicación Mule se conectará con esta cola para consumir los mensajes publicados en ella.

5.2.2. Módulo de procesamiento

Esta parte es la parte fundamental del proyecto, donde se lleva a cabo el procesamiento de eventos complejos, donde se reciben los eventos simples, se transforman, se comparan con una serie de patrones generándose una serie de eventos complejos que nos proporcionan una mayor información, y se almacenan y se envían al módulo de salida esos eventos generados. Sin este módulo los dos restantes carecerían de sentido.

En nuestro sistema lo forman, la **aplicación desarrollada mediante Mule ESB**, y los dos componentes software integrados en esta: el **servicio REST** que publica y el **motor Esper** que utiliza para el procesamiento de eventos complejos. También formaría parte de este módulo la **base de datos**, en la que se almacenan los eventos para dotar a la aplicación de persistencia, y de la que obtiene datos la API REST.

5.2.3. Módulo de salida

Recibe los datos ya procesados, en nuestro caso los eventos complejos, y se encarga de presentarlos al usuario.

En nuestro caso lo formarían dos componentes:

- **Cola de salida de RabbitMQ:** El módulo de procesamiento envía los datos ya procesados (eventos complejos) a esta cola de mensajería de RabbitMQ, de forma que puedan ser leídos por un usuario o consumidos por algún otro proceso.
- **Aplicación para dispositivos Android:** La aplicación recibe, mediante peticiones al servicio REST, los eventos complejos generados, y presenta la información obtenida de ellos, de manera amigable, comprensible y contextualizada al usuario. Esto hace que la información obtenida finalmente por el usuario tenga un valor mucho más valor que los que aportaban los eventos simples de los que partíamos al principio.

5.3. Esquema de la base de datos

Como ya pudimos constatar en los requisitos de información recogidos en el apartado 4.1.1, gracias a la integración y la transformación de datos realizada en los flujos de Mule ESB, no tenemos la necesidad de almacenar nada aparte de los eventos complejos generados.

Esto hace que el esquema de la base de datos sea extremadamente sencillo, constando simplemente de tres tablas, `aparcamientoevents`, `polucionevents` y `traficoevents`.

Cada una de estas tablas almacena los eventos complejos sucedidos como una cadena de caracteres, además de un id que nos servirá para identificar cada evento y que se genera automáticamente al introducirse un nuevo evento mediante la propiedad `AUTO_INCREMENT`.

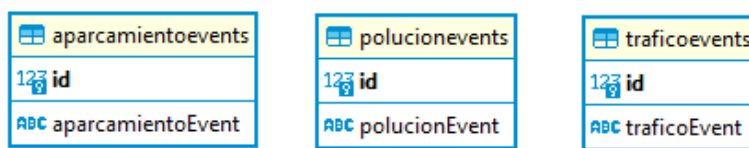


Figura 8. Esquema de la base de datos

5.4. Diseño de la aplicación Android

A la hora de diseñar una aplicación para dispositivos móviles, hay una serie de cosas que hay que tener en cuenta, y que lo diferencian del diseño de aplicaciones web o aplicaciones de escritorio para ordenadores.

Un aspecto fundamental es el uso eficiente de la pantalla, usando una estética minimalista, reduciendo los tiempos de carga y definiendo una jerarquía visual clara. Otra es la usabilidad y la accesibilidad, la aplicación debe comportarse igual independientemente del dispositivo y su resolución de pantalla, permitir navegar entre las distintas vistas de forma sencilla, etc.

El estilo y diseño de las aplicaciones Android ha sido estandarizado por Google, en el estándar conocido como Material Design[17].

Para diseñar la aplicación previamente a la implementación de la misma, hicimos uso de la herramienta Balsamiq, realizando un boceto navegable de lo que sería más tarde nuestra aplicación. El boceto, que incluiremos en el anexo C, recoge tanto la interfaz de usuario como la navegación.

Al ser simplemente un boceto, solamente se muestran algunas pantallas de la aplicación, ya que hay algunas en las que el diseño se repite, y con lo diseñado nos basta para tener una idea completa de lo que después intentamos desarrollar.

El boceto es navegable, pulsando en los iconos, botones y enlaces podemos ir navegando de una pantalla a otra tal y como lo haríamos si estuviéramos usando el dispositivo real. Esta es una de las principales ventajas de la herramienta utilizada.

Después, en los apartados 5.4.1 y 5.4.2, hablaremos de las decisiones tomadas en cuanto al diseño de la navegación de la aplicación y de la interfaz de usuario, basándonos en lo que se puede ver en el boceto.

5.4.1. Diseño de la navegación

Como hemos comentado, uno de los aspectos esenciales de una aplicación para Android es que la navegación dentro de ella sea sencilla y siga los estándares que cumplen la mayoría de las aplicaciones existentes, de manera que se adapte también a los modelos mentales que ya tienen los usuarios de esta. Si la estructura de la navegación no es buena, el usuario puede perderse fácilmente.

En el boceto incluido en el Anexo C, como hemos mencionado anteriormente, se puede observar la navegabilidad de la aplicación, de la misma manera en la que usaríamos la aplicación si estuviéramos usando un dispositivo real.

Hay dos formas principales de organización de la navegación:

- **Navegación plana:** Permite cambiar entre diferentes partes de la aplicación usando los menús, ya sean menús inferiores o superiores, o menús laterales.
- **Navegación jerárquica:** Permite ir navegando de una pantalla a otra, pulsando en un enlace o un botón. Debe permitir en todo momento volver atrás, ya que si no podemos ir a otra pantalla que no sea inferior jerárquicamente a la que nos encontramos.

Como podemos ver en el boceto, nuestra aplicación combina los dos tipos de organización de la navegación: usa navegación plana usando un menú lateral que se abre al pulsar en el icono de menú situado en la esquina superior izquierda y usa navegación jerárquica desde la pantalla de eventos, partiendo desde la pantalla que permite elegir el tipo de eventos y acabando en la información detallada de cada evento.

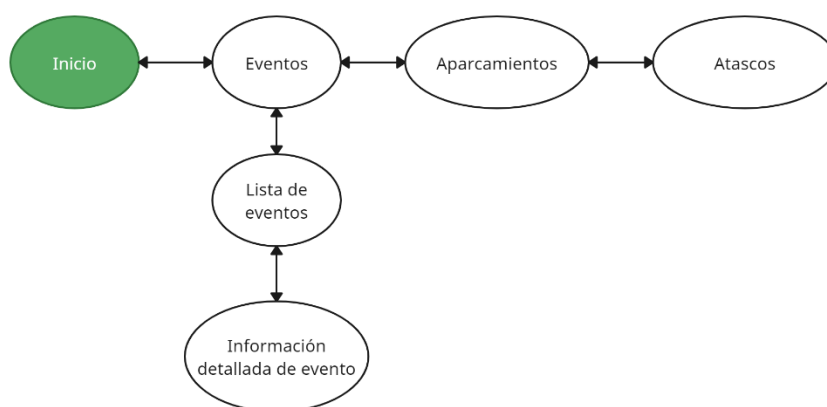


Figura 9. Diagrama de navegación de la App

El diagrama de la Figura 9 recoge la navegación entre las distintas pantallas de la aplicación. Aunque no lo recoge el diagrama desde cualquiera de las vistas inferiores en cuanto a jerarquía se refiere, también se puede navegar a cualquiera de las vistas

principales usando el menú lateral. La pantalla de Inicio es la que se carga al abrir nuestra aplicación.

5.4.2. Diseño de la interfaz de usuario

La interfaz de usuario es algo diferencial en una aplicación Android. Usar un diseño minimalista, elegir bien la paleta de colores, adaptar las vistas para que se vean de la misma manera sea cual sea la resolución de pantalla del dispositivo, aprovechar el poco tamaño de la pantalla, son aspectos a tener en cuenta a la hora del diseño de la app.

En cuanto a la paleta de colores de la aplicación, se han usado principalmente colores de la paleta “Light Green” de las paletas de colores [18] que recomienda el estándar Material Design de Google, ya que el verde es un color que representa el tema de la aplicación, el cuidado del medioambiente. Distintos colores de esta paleta se han utilizado para el icono de la aplicación, el logo, la toolbar, los botones, etc.

Además, se han utilizado otros colores de otras paletas de Material Design como el amarillo, naranja y rojo para mostrar la información de las alertas amarillas, naranjas y rojas por polución.

En cuanto a la adaptación a todas las resoluciones de pantalla, a la hora de indicar el tamaño de los elementos visuales de la aplicación, se decidió que se usarían unidades independientes de la densidad de píxeles, como sp (scalables pixels) para indicar el tamaño del texto o dp (dots per inch) para los demás elementos visuales. Android se encarga en tiempo de ejecución de traducir los valores representados en estas unidades a la cantidad de píxeles reales para cada densidad.

También se ha usado una estética minimalista, aprovechando el poco tamaño de pantalla, por ejemplo decidiendo usar un Navigation Drawer lateral desplegable como menú en lugar de usar un menú inferior que nos restaría espacio en la pantalla, o decidiendo la implementación de una ScrollView a la hora de mostrar la lista de eventos, de manera que solo se muestren unos cuantos eventos pero el usuario pueda ir deslizándose hacia abajo o arriba en la lista para ver todos los existentes.

Todas estas características descritas de la interfaz de usuario son también observables en el boceto de la aplicación incluido en el anexo C, situado al final del presente documento.

6. Implementación

Una vez diseñado el sistema, se procedió a la implementación de todos los componentes del mismo, a lo largo de las distintas iteraciones del proyecto. Los aspectos más importantes de esa implementación serán descritos en este capítulo. El capítulo está organizado de forma que se explican la implementación de cada componente del sistema, de uno en uno.

6.1. Esquemas y patrones de eventos

Los esquemas y patrones implementados en el lenguaje Esper EPL son seguramente la parte más importante del proyecto, y sobre lo que gira todo lo demás.

Los esquemas serán los que definan los tipos de eventos simples, que serán los que se después se emularán y se recibirán en Mule ESB desde una cola del bróker de mensajería RabbitMQ. Para que estos eventos simulados se procesen deberán seguir lo establecido en los esquemas que ahora describiremos y que desplegaremos en el motor Esper.

Estos eventos simples después se compararán con los patrones de eventos que despleguemos en el motor CEP. Si un evento simple sigue uno de los patrones, se generará un evento complejo, que tendrá un mayor significado semántico que el evento simple y nos proporcionará información relevante.

Esper EPL soporta la creación de jerarquías de patrones de eventos. Mediante “insert into” se crea un flujo de eventos complejos que podrán formar parte de la implementación de otros patrones de eventos, haciendo que unos patrones sean dependientes de otros.

6.1.1. Calidad del aire

En este apartado se describirán el esquema del eventos simple de calidad de aire y los patrones de eventos complejos relacionados con la calidad del aire. Para la implementación en Esper EPL se han usado una gran variedad de operadores, funciones, y ventanas de tiempo.

AIR QUALITY EVENT

Como podemos ver en la Figura 10 los eventos simples de calidad del aire tienen 4 parámetros, todos números enteros, el identificador del sensor, el nivel de dióxido de nitrógeno (NO₂), el nivel de dióxido de azufre (SO₂) y el nivel de ozono (O₃). Estos tres gases son de los principales causantes de la contaminación del aire.

```
@public @buseventtype create json schema AirQualityEvent
(idMetre int, nitrogenDioxideValue int, sulfurDioxideValue int, ozoneValue int);
```

Figura 10. Esquema “AirQualityEvent”

NITROGEN DIOXIDE ALERT

Este patrón (ver Figura 11) genera una alerta cuando se recibe un evento de calidad de aire en el que el valor del NO₂ es mayor a 10 µg/m³, valor marcado como meta en las directrices de calidad de aire de OMS.

```
@public INSERT INTO NitrogenDioxideAlert
SELECT idMetre, nitrogenDioxideValue, sulfurDioxideValue, ozoneValue
FROM AirQualityEvent WHERE nitrogenDioxideValue > 10;
```

Figura 11. Patrón "NitrogenDioxideAlert"

SULFUR DIOXIDE ALERT

Este patrón (ver Figura 12) es igual que el anterior pero para el dióxido de azufre. Genera una alerta cuando se recibe un evento de calidad de aire con valor del SO₂ mayor a 40 µg/m³, valor marcado como meta en las directrices de calidad de aire de OMS.

```
@public INSERT INTO SulfurDioxideAlert
SELECT idMetre, nitrogenDioxideValue, sulfurDioxideValue, ozoneValue
FROM AirQualityEvent WHERE sulfurDioxideValue > 40;
```

Figura 12. Patrón "SulfurDioxideAlert"

OZONE ALERT

Al igual que los dos anteriores, el patrón que podemos ver en la Figura 13, genera una alerta cuando se recibe un evento de calidad de aire en el que el nivel, en este caso, del O₃ es mayor a 100 µg/m³, valor marcado como meta en las directrices de calidad de aire de OMS.

```
@public INSERT INTO OzoneAlert
SELECT idMetre, nitrogenDioxideValue, sulfurDioxideValue, ozoneValue
FROM AirQualityEvent WHERE ozoneValue > 100;
```

Figura 13. Patrón "OzoneAlert"

AVERAGE POLLUTANT LEVELS

Este patrón (ver Figura 14) obtiene la media de los niveles de los tres gases contaminantes mencionados durante 30 segundos (con datos reales sería una hora en lugar de 30 segundos).

```
@public INSERT INTO AveragePollutantLevels
SELECT avg(nitrogenDioxideValue) as averageNitrogenDioxideLevel,
avg(sulfurDioxideValue) as averageSulfurDioxideLevel,
avg(ozoneValue) as averageOzoneLevel,
current_timestamp() as timestamp
FROM AirQualityEvent.win:time_batch(30 seconds);
```

Figura 14. "AveragePollutantLevels"

POLLUTION YELLOW ALERT

Este patrón (ver Figura 15) genera una alerta amarilla cuando la media de un solo contaminante es mayor a los niveles establecidos por las directrices de la OMS antes mencionados.

```
@public INSERT INTO PollutionYellowAlert
SELECT averageNitrogenDioxideLevel, averageSulfurDioxideLevel,
averageOzoneLevel, timestamp
FROM AveragePollutantLevels WHERE (averageNitrogenDioxideLevel > 10
AND averageSulfurDioxideLevel <= 40 AND averageOzoneLevel <= 100)
OR (averageNitrogenDioxideLevel <= 10 AND averageSulfurDioxideLevel > 40
AND averageOzoneLevel <= 100) OR (averageNitrogenDioxideLevel <= 10
AND averageSulfurDioxideLevel <= 40 AND averageOzoneLevel > 100);
```

Figura 15. "PollutionYellowAlert"

POLLUTION ORANGE ALERT

Este patrón (ver Figura 16) genera una alerta naranja cuando las medias de dos contaminantes son mayores a los niveles establecidos por las directrices de la OMS antes mencionados.

```
@public INSERT INTO PollutionOrangeAlert
SELECT averageNitrogenDioxideLevel, averageSulfurDioxideLevel,
averageOzoneLevel, timestamp
FROM AveragePollutantLevels WHERE (averageNitrogenDioxideLevel > 10
AND averageSulfurDioxideLevel > 40 AND averageOzoneLevel <= 100)
OR (averageNitrogenDioxideLevel <= 10 AND averageSulfurDioxideLevel > 40
AND averageOzoneLevel > 100) OR (averageNitrogenDioxideLevel > 10
AND averageSulfurDioxideLevel <= 40 AND averageOzoneLevel > 100);
```

Figura 16. Patrón "PollutionOrangeAlert"

POLLUTION RED ALERT

Este patrón (Figura 17) genera una alerta roja cuando las medias de tres contaminantes son mayores a los niveles establecidos por las directrices de la OMS antes mencionados.

```
@public INSERT INTO PollutionRedAlert
SELECT averageNitrogenDioxideLevel, averageSulfurDioxideLevel, averageOzoneLevel, timestamp
FROM AveragePollutantLevels WHERE averageNitrogenDioxideLevel > 10
AND averageSulfurDioxideLevel > 40 AND averageOzoneLevel > 100;
```

Figura 17. Patrón "PollutionRedAlert"

6.1.2. Aparcamiento

En esta sección se describirán el esquema del eventos simple de aparcamiento y los patrones de eventos complejos relacionados con el aparcamiento. Para la implementación de los patrones de eventos complejos, usando Esper EPL, se han utilizado ventanas de tiempo y de datos, y una gran variedad de operadores y funciones.

PARKING EVENT

Como podemos observar en la Figura 18, los eventos simples de aparcamiento solamente tienen dos parámetros, los dos números enteros, el identificador del sensor y el valor, que será 0 si el aparcamiento está libre y 1 si está ocupado.

```
@public @buseventtype create json schema ParkingEvent
(idMetre int, value int);
```

Figura 18. Esquema "ParkingEvent"

FREE PARKING SPOT

Este patrón (Figura 19) hace que se genere un evento complejo si un aparcamiento está libre, es decir, si el valor es igual a 0.

```
@public INSERT INTO FreeParkingSpot
SELECT idMetre FROM ParkingEvent.win:time(1 seconds)
WHERE value = 0;
```

Figura 19. Patrón "FreeParkingSpot"

TOTAL FREE PARKING SPOTS

Este patrón (ver Figura 20) cuenta el número total de aparcamientos libres que hay 1 segundo concreto (con datos reales serían 2 minutos ya que cada 2 minutos los sensores instalados en los aparcamientos emitirán una señal indicando su estado).

```
@public INSERT INTO TotalFreeParkingSpots
SELECT count (*) as totalPlazas
FROM FreeParkingSpot.win:time_batch(1 seconds);
```

Figura 20. Patrón "TotalFreeParkingSpots"

OCCUPIED PARKING SPOT

Este patrón, que podemos ver en la Figura 21, hace lo contrario a FreeParkingSpot, genera un evento complejo si un aparcamiento está ocupado (el valor es igual a 1).

```
@public INSERT INTO OccupiedParkingSpot
SELECT idMetre FROM ParkingEvent.win:time(1 seconds)
WHERE value = 1;
```

Figura 21. Patrón "OccupiedParkingSpot"

TOTAL OCCUPIED PARKING SPOTS

El patrón *TotalOccupiedParkingSpots* (ver Figura 22), como indica su nombre, cuenta los aparcamientos que se encuentran ocupados en 1 segundo (con datos no simulados serían 2 minutos).

```
@public INSERT INTO TotalOccupiedParkingSpots
SELECT count (*) as totalPlazas
FROM OccupiedParkingSpot.win:time_batch(1 seconds);
```

Figura 22. Patrón "TotalOccupiedParkingSpots"

AVERAGE OCCUPATION

Este patrón (ver Figura 23) genera un evento complejo que indica la media de ocupación de la bolsa de aparcamiento con sensores instalados en una hora (30 segundos en la simulación).

```
@public INSERT INTO AverageOccupation
SELECT avg(totalPlazas) as avgOccupation,
current_timestamp().getHourOfDay() as timestamp
FROM TotalOccupiedParkingSpots.win:time_batch(30 seconds);
```

Figura 23. Patrón "AverageOccupation"

MAX OCCUPATION HOUR

Este patrón que podemos observar en la Figura 24 indica la hora del día de máxima ocupación, a partir de los eventos complejos generados que cumplen con el anterior patrón.

```
@public INSERT INTO MaxOccupationHour
SELECT max(avgOccupation) as Occupation,
timestamp as maxOccupationHour
FROM AverageOccupation.win:length_batch(24);
```

Figura 24. Patrón "MaxOccupationHour"

6.1.3. Tráfico

En este apartado se describirá la implementación del esquema de eventos simple de tráfico y los patrones de eventos complejos relacionados. Para la implementación en Esper EPL de los patrones se han utilizado ventanas de tiempo, una gran variedad de operadores y funciones, y "patterns".

TRAFFIC JAM EVENT

Los eventos simples de aparcamiento (Figura 25) tendrán tres parámetros: el identificador del sensor (número entero), la velocidad del coche que circula bajo el sensor (decimal de tipo double), y la matrícula del coche (una cadena de caracteres).

```
@public @buseventtype create json schema TrafficJamEvent
(idMetre int, speed double, carPlate string);
```

Figura 25. Esquema "TrafficJamEvent"

SPEED OVER 15

Este patrón (ver Figura 26) capta cada vehículo que pasa bajo el sensor con velocidad superior a 15 km/h.

```
@public INSERT INTO SpeedOver15
SELECT idMetre, speed, carPlate,
current_timestamp() as timestamp
FROM TrafficJamEvent WHERE speed > 15;
```

Figura 26. Patrón "SpeedOver15"

SPEED BELOW 15

Este patrón (ver Figura 27) hace lo contrario al anterior, capturando cada vehículo que circula con velocidad inferior a 15 km/h.

```
@public INSERT INTO SpeedBelow15
SELECT idMetre, speed, carPlate,
current_timestamp() as timestamp
FROM TrafficJamEvent WHERE speed < 15;
```

Figura 27. Patrón "SpeedBelow15"

POSSIBLE TRAFFIC JAM

Este patrón (Figura 28) detecta si puede existir un atasco, captando todos los vehículos que circulan por debajo de los 15 km/h hasta que uno lo hace a una velocidad anterior.

```
@public INSERT INTO PossibleTrafficJam
SELECT t2.idMetre as idMetre,
t1[0].timestamp as timestampStart,
t2.timestamp as timestampEnd
FROM PATTERN [t1=SpeedBelow15 until t2=SpeedOver15]
GROUP BY t2.idMetre;
```

Figura 28. Patrón "PossibleTrafficJam"

CURRENT TRAFFIC JAM

Este patrón, observable en la Figura 29, determina si existe o no un atasco a partir de los eventos complejos "PossibleTrafficJam". Si desde que circula el primero de los coches a menos de 15 km/h hasta que vuelve la circulación a la normalidad (pasa un coche a más de 15 km/h) han pasado más de 1 segundo (2 minutos con datos no simulados) se establece que existe un atasco.

```
@public INSERT INTO CurrentTrafficJam
SELECT idMetre, timestampStart, timestampEnd, (timestampEnd - timestampStart) as duration
FROM PossibleTrafficJam
WHERE (timestampEnd - timestampStart).getSecondOfMinute() > 1
OR (timestampEnd - timestampStart).getSecondOfMinute() >
current_timestamp.set('hour', 0).set('min', 0).set('sec', 1).set('msec', 0).getSecondOfMinute();
```

Figura 29. Patrón "CurrentTrafficJam"

TOTAL DAY TRAFFIC JAMS

Este patrón (ver Figura 30) cuenta los atascos que se producen en la ciudad en un día (12 minutos en la simulación), a partir de los eventos complejos de tipo “CurrentTrafficJam”.

```
@public INSERT INTO TotalDayTrafficJams
SELECT count(*) as total
FROM CurrentTrafficJam.win:time_batch(12 minutes);
```

Figura 30. Patrón "TotalDayTrafficJams"

AVERAGE SPEED ZONE HOUR

El patrón que podemos ver en la Figura 31 detecta la velocidad media con la que se circula bajo un sensor en una hora (30 segundos en la simulación).

```
@public INSERT INTO AverageSpeedZoneHour
SELECT idMetre, avg(speed) as avgSpeed
FROM TrafficJamEvent.win:time_batch(30 seconds)
GROUP BY idMetre;
```

Figura 31. Patrón "AverageSpeedZoneHour"

AVERAGE SPEED ZONE DAY

Este patrón (ver Figura 32) detecta la velocidad media con la que se circula bajo un sensor en un día (12 minutos en la simulación).

```
@public INSERT INTO AverageSpeedZoneDay
SELECT idMetre, avg(avgSpeed) as avgSpeed
FROM AverageSpeedZoneHour.win:time_batch(12 minutes)
GROUP BY idMetre;
```

Figura 32. Patrón "AverageSpeedZoneDay"

TOTAL VEHICLES ZONE HOUR

Este patrón (ver Figura 33) cuenta el número de vehículos que circulan en una hora (30 segundos en la simulación) bajo un sensor.

```
@public INSERT INTO TotalVehiclesZoneHour
SELECT idMetre, count(*) as numVehicles
FROM TrafficJamEvent.win:time_batch(30 seconds)
GROUP BY idMetre;
```

Figura 33. Patrón "TotalVehiclesZoneHour"

AVERAGE VEHICLES ZONE DAY

Este patrón, cuya implementación se puede ver en la Figura 34, determina la media diaria (12 minutos en la simulación) de vehículos que circulan bajo el sensor por hora.

```
@public INSERT INTO AverageVehiclesZoneDay
SELECT idMetre, avg(numVehicles) as avgVehicles
FROM TotalVehiclesZoneHour.win:time_batch(12 minutes)
GROUP BY idMetre;
```

Figura 34. Patrón "AverageVehiclesZoneDay"

6.2. Flujos de Mule ESB

En esta sección se describirá la implementación de los 4 flujos del proyecto de Mule ESB, tanto los componentes arrastrados desde la paleta de Anypoint Studio, como la configuración necesaria.

FLUJO 1: RECEPCIÓN Y TRATAMIENTO DE EVENTOS SIMPLES

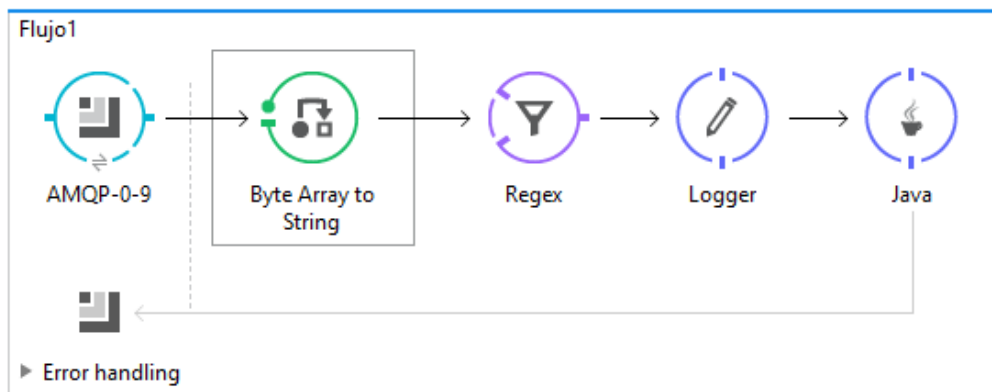


Figura 35. Flujo 1 Mule ESB "Recepción y tratamiento de eventos simples"

El primer flujo será el encargado de recibir los eventos simples, tratarlos, y mandarlos al motor de Esper, como podemos observar en la Figura 35.

Recibiremos los eventos simples como mensajes desde una cola del bróker de mensajería RabbitMQ usando el protocolo AMQP 0.9. Para ellos deberemos configurar el conector de AMQP para que se conecte con nuestro servidor de RabbitMQ, indicando el host, el puerto donde está lanzado el servicio, el nombre de usuario y la contraseña.

Después, por pasos, convertiremos el mensaje recibido como una cadena de bytes a una cadena de caracteres, lo compararemos con una expresión regular para comprobar que tiene el formato deseado, lo imprimimos por consola, y lo enviamos al motor de Esper.

La expresión regular con la que comparamos los mensajes recibidos es: `^{\\"eventName\\":\\"[^\\"]*"}`

De esta manera nos aseguramos de que los mensajes recibidos tengan el mismo formato con el que los simulamos desde nITROGEN, empezando por el nombre del tipo de evento simple. En caso contrario son ignorados.

Si cumplen el formato deseado, se imprimen por pantalla y se envían al motor CEP de Esper mediante el componente Java llamado “SendEventToEsperComponent”.

FLUJO 2: DESPLIEGUE DE PATRONES EN EL MOTOR CEP

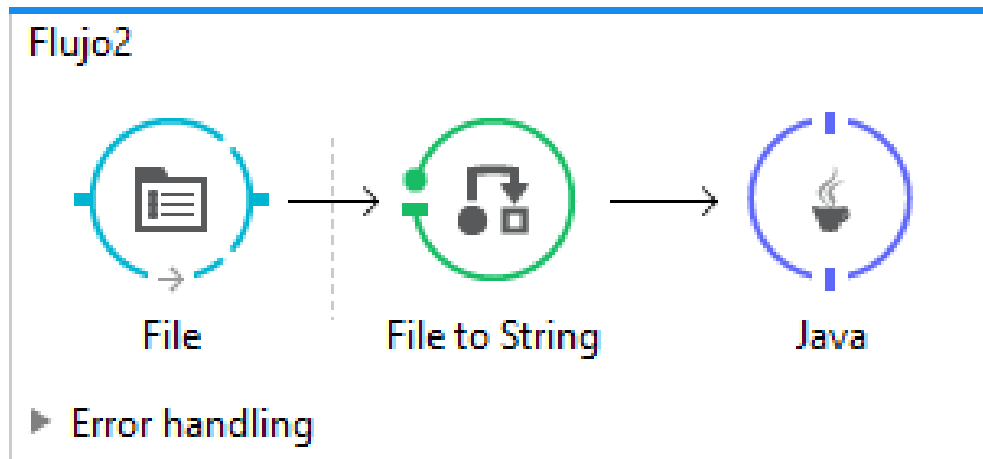


Figura 36. Flujo 2 Mule ESB "Despliegue de patrones en el motor CEP"

El segundo flujo, que podemos observar en la Figura 36 es el encargado de recibir los ficheros que contienen los esquemas y patrones implementados en el lenguaje EsperEPL y desplegarlos en el motor CEP.

Para ello, irá cogiendo cada fichero en formato .epl de la carpeta PatronesSinProcesar, lo moverá a la carpeta PatronesProcesados, convertirá el contenido (el esquema o patrón) a una cadena de texto y lo desplegará en el motor Esper mediante el componente Java llamado “AddEventPatternToEsperComponent”.

FLUJO 3: ALMACENAMIENTO Y ENVÍO DE EVENTOS COMPLEJOS

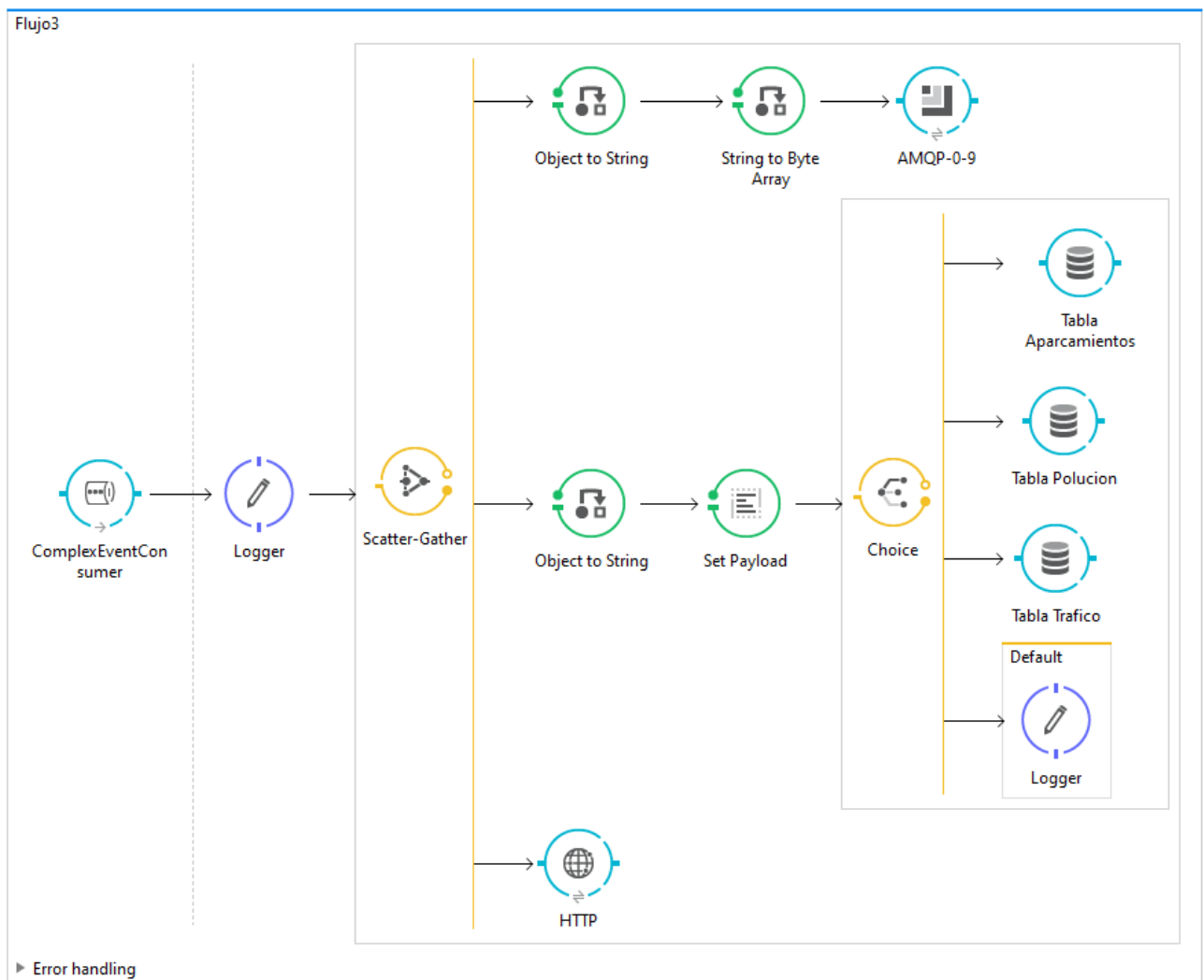


Figura 37. Flujo 3 Mule ESB "Almacenamiento y envío de eventos complejos"

Este flujo que podemos observar en la Figura 37 será el encargado de consumir los eventos complejos que se generen en el motor Esper, los imprime por consola, y realiza tres acciones en paralelo:

- Envía los eventos complejos a otra cola de mensajes de RabbitMQ.
- Almacena los eventos complejos según el tipo de evento (aparcamiento, tráfico o polución) en una tabla distinta de una base de datos MySQL.
- Invoca a un método POST de la API REST que veremos en el siguiente flujo cómo publicamos.

Para consumir los eventos complejos hacemos uso de un módulo VM Connector, que nos permite gestionar eventos asíncronos que suceden dentro de la aplicación.

Antes de esto, debemos configurar dentro de los elementos globales de nuestra aplicación de Anypoint Studio, un VM Endpoint que establecerá la ruta de la cola a la

que el motor de Esper enviará los eventos complejos y desde la que los consumiremos con el VM Connector creado.

Nuestro VM Endpoint tendrá configuración que se puede observar en la Figura 38.

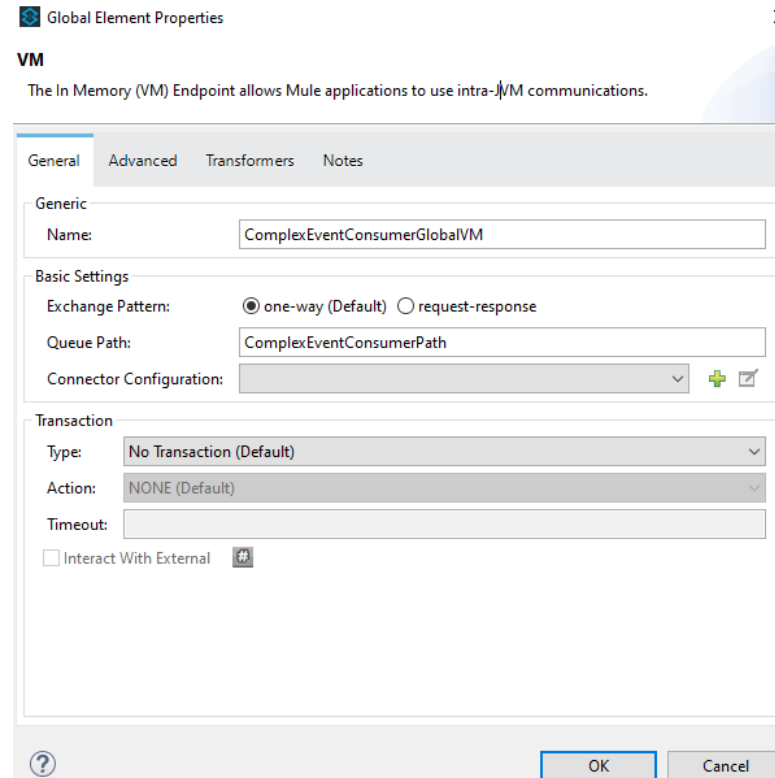


Figura 38. Configuración VM Endpoint

En nuestro VM Connector deberemos introducir la ruta establecida en el endpoint, que será también la ruta que indicaremos en el componente Java “AddEventPatternToEsperComponent” antes mencionado que envíe los eventos complejos generados en el motor CEP.

Una vez consumidos los eventos complejos, los mostramos por consola mediante un Logger, y mediante un componente Scatter-Gather definimos los tres subflujos paralelos anteriormente descritos.

En el primero, transformamos el objeto JSON que contiene el evento complejo en una cadena de caracteres, y esta en una cadena de Bytes, y se enviarán a una cola de mensajes de RabbitMQ distinta a la cola desde la que recibíamos los eventos simples, también mediante el protocolo AMQP 0.9.

En el segundo también transformamos el objeto a una cadena de caracteres, que establecemos como payload del flujo. A continuación, mediante un componente Choice, según sea el nombre del evento complejo generado, se almacenan en una tabla de la base de datos u otra.

Esto último se implementó en la iteración 4 (Mejora del proyecto Mule y desarrollo de la API REST), en la iteración 2 (Implementación de los flujos de Mule) todos los eventos complejos eran insertados en la misma tabla, independientemente del tipo de evento.

Si el evento complejo está relacionado con el aparcamiento, se almacenará en la tabla *aparcamientoevents* mediante la consulta SQL:

```
INSERT INTO aparcamientoevents (aparcamientoEvent) VALUES ([payload]);
```

Si el evento complejo está relacionado con el tráfico, se almacenará en la tabla *traficoevents* mediante la consulta SQL:

```
INSERT INTO traficoevents (traficoEvent) VALUES ([payload]);
```

Por último, si el evento complejo trata de la polución, se almacenará en la tabla *polucionevents* mediante la consulta:

```
INSERT INTO polucionevents (polucionEvent) VALUES ([payload]);
```

En cada componente Database deberemos seleccionar nuestra configuración de MySQL, introduciendo el host, el puerto donde está corriendo el servicio, el nombre de usuario, la contraseña, y el nombre de la base de datos.

En el último de los subflujos, simplemente mediante un componente HTTP invocamos al método POST “AddEvent” a partir de su URL. Este método aumentará en uno la variable numberOfEvents y devolverá un mensaje de éxito.

Para que funcione, en la configuración del componente deberemos también introducir el protocolo (HTTP en nuestro caso), el host, y el puerto donde se está ejecutando el servicio REST.

FLUJO 4: CREACIÓN DE SERVICIO REST

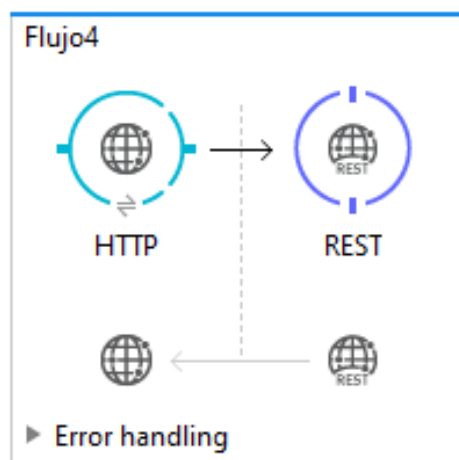
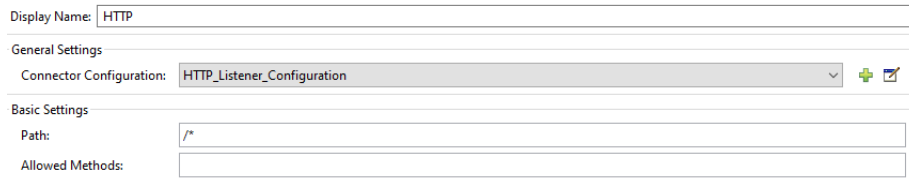


Figura 39. Flujo 4 Mule ESB "Creación de servicio REST"

Este flujo se encarga de publicar una API REST, que después usaremos entre otras cosas para acceder a la información de los eventos complejos desde la aplicación de Android.

Primero tenemos un componente HTTP, que configuraremos para que escuche todas las peticiones HTTP que se hagan al puerto 8081 sea cual sea la ruta. Para ello estableceremos el parámetro Path a “/*”, como se puede ver en la Figura 40.



Display Name: HTTP

General Settings

Connector Configuration: HTTP_Listener_Configuration

Basic Settings

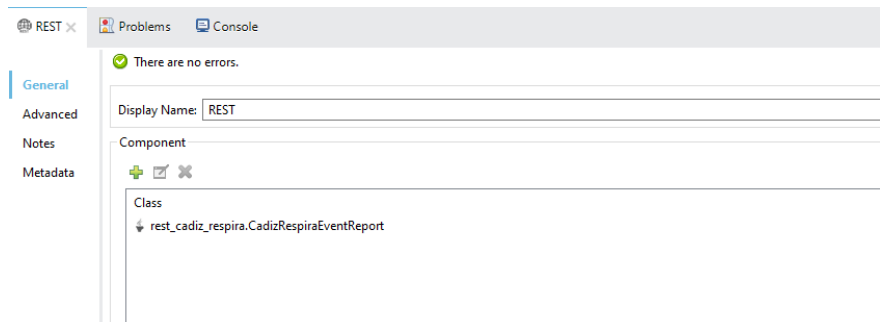
Path: /*

Allowed Methods:

Figura 40. Configuración componente HTTP

Además, en la HTTP_Listener_Configuration del componente indicaremos que lance el servicio en el puerto 8081 en el host por defecto (0.0.0.0).

Después, añadimos un componente REST en el que añadimos nuestra clase Java que contiene la implementación de todos los métodos del servicio REST, tal y como se observa en la Figura 41.



REST

General

Advanced

Notes

Metadata

Display Name: REST

Component

Class

rest_cadiz_respira.CadizRespiraEventReport

Figura 41. Configuración componente REST

Para los métodos de la API REST, se hará uso de JAX-RS, que nos provee de anotaciones como @GET, @POST, @Path, @Produces, @PathParam y @XmlRootElement, que nos harán mucho más sencilla la implementación de todos los métodos.

En la iteración 2 (Implementación de los flujos en Mule), ya se creó el servicio REST. Sin embargo, la funcionalidad de este servicio era muy limitada, y no sería hasta la iteración 4 cuando se ampliaría esta. Al final de la iteración 2, la API REST tenía tres métodos:

El primero de los métodos simplemente era un método GET que mostraba si el servicio REST funcionaba correctamente, como podemos observar en la Figura 42.

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayIsWorking() {
    return "CadizRespiraEventReport is working.";
}
```

Figura 42. Método API "sayIsWorking"

El segundo de los métodos, observable en la Figura 43, mediante un POST, aumentaba en uno una variable declarada estática llamada `numberOfEvents`, de manera que cada vez que se generara un evento complejo nuevo, se aumentara el número de eventos, y se devolviera un mensaje de éxito.

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Path("/AddEventReport")
public String addEvent() {
    numberOfEvents++;
    return "Event added correctly.";
}
```

Figura 43. Método API "addEvent"

El tercero simplemente era un método GET que mostraba el valor de `numberOfEvents`, es decir, el número de eventos complejos sucedidos hasta el momento. Podemos verlo en la Figura 44.

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/NumberOfEvents")
public String getNumberOfEvents() {
    return numberOfEvents + " events have happened.";
}
```

Figura 44. Método API "getNumberOfEvents"

En la iteración 4 en cambio, se crearon una serie de métodos que se dedicaban a recuperar los eventos complejos almacenados en la base de datos y a presentarlos para que sean recuperados después por la aplicación para Android.

En concreto, se crearon 6 métodos GET, aunque se mostrarán en detalle solamente dos de ellos, ya que los demás son prácticamente iguales, solo que para los otros dos tipos de eventos.

```

@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/aparcamientos")
public List<Event> getEventosAparcamiento() {
    ResultSet rs = null;
    List<Event> eventosAparcamiento = new ArrayList<Event>();
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con = DriverManager.getConnection(MYSQL_URL,MYSQL_USER,MYSQL_PASSWORD);
        String query = "select * from aparcamientoevents";
        PreparedStatement st = con.prepareStatement(query);
        rs = st.executeQuery();

        while(rs.next()) {
            Event eventoAparcamiento = new Event();
            eventoAparcamiento.setId(rs.getInt("id"));
            eventoAparcamiento.setEvent(rs.getString("aparcamientoEvent"));
            eventosAparcamiento.add(eventoAparcamiento);
        }
    }
    catch(Exception e){System.out.println("Error: " + e); }

    return(eventosAparcamiento);
}

```

Figura 45. Método API "getEventosAparcamiento"

Este método que vemos en la Figura 45 establece una conexión con la base de datos utilizando el conector de jdbc y ejecuta una consulta que obtiene todos los eventos de la tabla de aparcamientos. A partir de estos eventos, por cada evento recibido como respuesta de la consulta crea un objeto de la clase Event, que podemos observar en la Figura 46 , le asigna los atributos del evento, y los añade a una lista de objetos Event. El método produce un JSON de la lista de objetos Event que se crean.

La clase Event tiene la etiqueta @XmlElement, que permite hacer automáticas de objetos de la clase a tipo XML o tipo JSON como es en nuestro caso.

```

@XmlRootElement
public class Event {
    private int id;
    private String event;

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getEvent() { return event; }
    public void setEvent(String event) { this.event = event; }
}

```

Figura 46. Clase de retorno de API "Event"

Para establecer la conexión con la base de datos, se usan tres parámetros, la URL para conectarse a la base de datos y el usuario y contraseña necesarios para acceder a ella. En nuestro caso, están declarados como se ve en la Figura 47.

```
private final static String MYSQL_URL =  
"jdbc:mysql://localhost/CadizRespiraEventReport?autoReconnect=true&useSSL=false";  
private final static String MYSQL_USER = "root";  
private final static String MYSQL_PASSWORD = "isi";
```

Figura 47. Parámetros conexión MySQL

El método explicado, que devuelve un JSON con la lista de eventos de aparcamiento existentes en la base de datos, está replicado para cada tipo de evento, ya que cada uno se encuentra en una tabla distinta de la base de datos. Esto será igual para el método que ahora se pasará a comentar y que podemos ver en la Figura 48, que devuelve un JSON con un evento concreto de aparcamiento, pasando como parámetro el identificador del evento.

Este método hará uso de la etiqueta `@PathParam`, que permite usar en la función parámetros que se han enviado como parte del Path.

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
@Path("/aparcamientos/{idEvento}")  
public Event getEventoAparcamiento(@PathParam("idEvento") int id) {  
    ResultSet rs = null;  
    Event eventoAparcamiento = new Event();  
    try {  
        Class.forName("com.mysql.cj.jdbc.Driver");  
        Connection con = DriverManager.getConnection(MYSQL_URL,MYSQL_USER,MYSQL_PASSWORD);  
        String query = "select * from aparcamientoevents where id = " + id;  
        PreparedStatement st = con.prepareStatement(query);  
        rs = st.executeQuery();  
  
        while(rs.next()) {  
            int idEvento = (rs.getInt("id"));  
            eventoAparcamiento.setId(idEvento);  
            String event = (rs.getString("aparcamientoEvent"));  
            eventoAparcamiento.setEvent(event);  
        }  
    }  
    catch(Exception e){System.out.println("Error: " + e); }  
    return(eventoAparcamiento);  
}
```

Figura 48. Método API "getEventoAparcamiento"

6.3. Aplicación de Android

La aplicación móvil, aunque no es el elemento central del sistema sino que es un complemento a este para hacerlo más accesible y comprensible, seguramente sea el más extenso en cuanto a código y lógica interna se refiere.

Como ya explicamos con anterioridad, fue desarrollado durante dos iteraciones:

- La iteración recogida en el apartado 3.2.3, donde se crearon todos los *layout* (plantillas que se encargan de la apariencia visual de la vista y que normalmente se rellenarán desde la lógica de la aplicación) y el *NavigationDrawer*, menú principal de la aplicación que nos permite navegar de una pantalla otra.
- La última iteración, recogida en el apartado 3.2.5, donde se implementó la funcionalidad de la app, realizando peticiones a la API y tratando los datos extraídos de los eventos complejos y haciendo que se muestren de una forma sencilla al usuario, ya sea mediante una lista de eventos, mediante una relación detallada de cada atributo de un evento, o mostrando los eventos de aparcamientos libres y atascos actuales sobre un mapa.

6.3.1. Visualización de eventos

A la hora de mostrar la lista de eventos de un tipo, se recibe como parámetro el tipo de evento que se le pasa como extra desde la actividad *VerEventos*, que será un tipo u otro según el botón en el que se pulse. En la Figura 49 podemos ver como se extrae el parámetro dentro del método *onCreate* (donde se inicializa la vista).

```
Bundle extras = getIntent().getExtras();
if (extras != null) {
    tipoEvento = extras.getString("tipoEvento");
}
```

Figura 49. Extracción de parámetro en el método *onCreate*

Después, se le asigna a la *RecyclerView* que nos servirá para mostrar la lista de eventos, un *adapter*, que hará que el *layout* (o plantilla) se rellene según los datos de cada evento complejo recibido, como vemos en la Figura 50.

```
//Referenciamos al RecyclerView
mRecyclerView = (RecyclerView) findViewById(R.id.my_recycler_view);

//Mejoramos rendimiento con esta configuración
mRecyclerView.setHasFixedSize(true);

//Creamos un LinearLayoutManager para gestionar el item.xml creado antes
mLayoutManager = new LinearLayoutManager(this);
//Lo asociamos al RecyclerView
mRecyclerView.setLayoutManager(mLayoutManager);

mRecyclerView.setAdapter(new EventoAdapter(new ArrayList<>()));
```

Figura 50. Inicialización y configuración de *RecyclerView*

Por último, según el tipo de evento (aparcamiento, tráfico o polución), llamamos a una tarea asíncrona u otra, que se encargarán de hacer una petición a la API para obtener la lista de eventos correspondiente y añadirlos a una lista que se le pasará al adapter para que la muestre en pantalla.

La petición a la API se hará como se observa en la Figura 51, que muestra como obtenemos los eventos de tráfico, desde el método *doInBackground* de la tarea asíncrona, que será el método principal de esta.

```
@Override
protected JSONArray doInBackground(Void... voids) {
    JSONArray result = null;
    HttpURLConnection urlConnection = null;
    try {
        URL urlToRequest = new URL("http://" + IP + ":8081/CadizRespiraEventReport/trafico");
        //Realizamos la petición GET a nuestra API
        urlConnection = (HttpURLConnection) urlToRequest.openConnection();
        urlConnection.setRequestMethod("GET");
        urlConnection.connect();

        //Procesar los resultados obtenidos
        InputStream in = urlConnection.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        StringBuilder builder = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            builder.append(line);
        }
        result = new JSONArray(builder.toString());
    } catch (IOException | JSONException e) {
        e.printStackTrace();
    } finally {
        if (urlConnection != null)
            urlConnection.disconnect();
        return result;
    }
}
```

Figura 51. Petición a la API REST de los eventos de tráfico

Después, como se muestra en la Figura 52, en el método *onPostExecute*, que como su propio nombre indica se ejecuta al finaliza la tarea principal llevada a cabo en el método *doInBackground*, si se han recibido eventos, se extraen y se tratan los datos del evento que deseamos mostrar en la lista, y se añade un nuevo objeto de la clase evento con esos datos a un *ArrayList*. Este *ArrayList* será después pasado como parámetro a un *EventoAdapter* que será el encargado de mostrar los eventos de la lista en la pantalla.

```

@Override
protected void onPostExecute(JSONArray data) {
    //Si existen eventos
    if (data != null) {
        //Creamos un ArrayList de Eventos
        ArrayList<Evento> eventos = new ArrayList<>();
        //Para cada posición del JSONArray
        for (int i = 0; i < data.length();i++) {
            try {
                //Extraemos el objeto JSON almacenado en la posición i del JSONArray
                JSONObject oneObject = data.getJSONObject(i);
                //Tomamos el valor asociado a la clave "id"
                String id = oneObject.getString("id");
                //Tomamos el valor asociado a la clave "event"
                String event = oneObject.getString("event");
                //Tomamos nombre de evento
                String nombreEvento = event.substring(0, event.indexOf("=")).substring(1);
                //Tomamos timestamp
                long fechaHora = 0;
                Pattern pattern = Pattern.compile("timestamp=(\\d+)");
                Matcher matcher = pattern.matcher(event);
                if (matcher.find()) {
                    fechaHora = Long.parseLong(matcher.group(1));
                }

                //Añadimos la reserva a nuestro ArrayList
                eventos.add(new Evento(id,"Trafico",event,nombreEvento,fechaHora));
            }
            catch (JSONException e) {
                e.printStackTrace();
            }
            finally {
                //Creamos un adapter pasándole todos nuestros eventos
                mAdapter = new EventoAdapter(eventos);
                //Asociamos el adaptador al RecyclerView
                mRecyclerView.setAdapter(mAdapter);
            }
        }
    }
}

```

Figura 52. Obtención y tratado de los atributos de los eventos de tráfico

Para mostrar información detallada de un evento, se repite más o menos el mismo proceso con algunas diferencias:

- En lugar de recibirse como parámetro el tipo de evento, se recibe el identificador del evento en el que se ha pulsado desde la lista de eventos, que es del que se mostrarán los detalles.

- No solo obtendremos el id del evento, el nombre y la fecha y hora, sino que extraeremos la totalidad de atributos de cada evento complejo ya que se mostrarán todos ellos.
- Como solo se mostrará un evento, no usamos una *RecyclerView*, sino que dinámicamente creamos un *LinearLayout* por cada atributo a mostrar y lo añadimos a la vista.

6.3.2. Mapa de aparcamientos libres

A la hora de mostrar el mapa que señalará los aparcamientos libres, primero centraremos el mapa en Cádiz, moviendo la cámara del mismo mediante un objeto *CameraUpdate* al que se le pasan las coordenadas de la ciudad.

Después, se le asigna al mapa un *InfoWindowAdapter*, que inflará una plantilla creada para las ventanas de información que se mostrarán al pulsar en uno de los marcadores de los aparcamientos libres a partir de los datos establecidos como etiqueta del marcador.

Finalmente, se ejecuta la tarea asíncrona que se encargará de establecer una conexión a la API para obtener qué aparcamientos se encuentran libres y cuáles no.

En la Figura 53 podemos ver cómo hace todo lo comentado.


```

@Override
public void onMapReady(GoogleMap googleMap) {

    mMap = googleMap;

    LatLng cadiz = new LatLng(36.527062, -6.288596);
    CameraUpdate cameraUpdate = CameraUpdateFactory.newLatLngZoom(cadiz, 12);
    mMap.moveCamera(cameraUpdate);

    mMap.setInfoWindowAdapter(new GoogleMap.InfoWindowAdapter() {
        @Override
        public View getInfoWindow(Marker marker) {
            View v = getLayoutInflater().inflate(R.layout.aparcamiento_custom_info_window, null);
            TextView idSensorTv = v.findViewById(R.id.idSensor);
            TextView fechaHoraTv = v.findViewById(R.id.timestamp);

            InfoAparcamientoLibre sensorInfo = (InfoAparcamientoLibre) marker.getTag();
            idSensorTv.setText(sensorInfo.getIdSensor());
            fechaHoraTv.setText(sensorInfo.getFechaHora());

            return v;
        }

        @Override
        public View getInfoContents(Marker marker) {
            return null;
        }
    });

    new getAparcamientosLibres(mMap).execute();
}

```

Figura 53. Preparación del mapa de aparcamientos libres

La tarea asíncrona, recuperará los eventos de aparcamiento tal y como se hacía en ListaEventos, sin embargo, en el método PostExecute, si el evento recibido desde el servicio REST es un evento complejo del tipo “FreeParkingSpot” o “OccupiedParkingSpot”, se almacena en la posición idSensor-1 el par estado, timestamp, siendo estado un booleano, true si la plaza de aparcamiento está libre, y false si está ocupada.

De esta manera, al recibir todos los eventos, tendremos al final en la lista estadoSensor, el estado de cada uno de los sensores de aparcamiento, y la fecha y hora a la que se comprobó por última vez ese estado (timestamp del evento complejo).

A continuación, en la Figura 54, podemos ver la implementación de todo lo mencionado:

```
if (typeOfEvent.equals("FreeParkingSpot")||typeOfEvent.equals("OccupiedParkingSpot")) {  
    Pattern pattern = Pattern.compile("timestamp=(\\d+), idMetre=(\\d+)");  
    Matcher matcher = pattern.matcher(valoresEvento);  
    if (matcher.find()) {  
        // Añade el marcador al mapa utilizando idMetre para acceder a las coordenadas correspondientes  
        idSensor = Integer.parseInt(matcher.group(2));  
        if(typeOfEvent.equals("FreeParkingSpot")){  
            pair = new AbstractMap.SimpleEntry<>(true,matcher.group(1));  
            estadoSensor.set(idSensor-1,pair);  
        }  
        else {  
            pair = new AbstractMap.SimpleEntry<>(false,matcher.group(1));  
            estadoSensor.set(idSensor-1,pair);  
        }  
    }  
}
```

Figura 54. Obtención del estado de cada plaza de aparcamiento

Tras hacer esto para cada evento, el método recorre cada posición de la lista estadoSensor y si el estado del sensor es true, obtiene el par de coordenadas asociado al sensor, y añade un marcador en esa posición del mapa, de color verde. Además, como etiqueta, le asigna un objeto de la clase *InfoAparcamientoLibre*, con el id del sensor y la fecha y hora a la que se generó el evento complejo.

Este objeto es desde el que anteriormente vimos que se recuperaba la información para mostrar la ventana de información al pulsar en el marcador.

En la Figura 55 podemos ver toda la implementación.

```

for (int x = 0; x < 10; x++) {
    if (estadoSensor.get(x).getKey()) {
        LatLng coordinates = Coordinates.getCoordinates(x);
        if (coordinates != null) {
            long timestamp = Long.parseLong(estadoSensor.get(x).getValue());
            Date date = new Date(timestamp);
            SimpleDateFormat sm = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
            String fechaYHora = sm.format(date);
            Marker marker = mMap.addMarker(new MarkerOptions()
                .position(coordinates)
                .icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_GREEN)));
            marker.setTag(new InfoAparcamientoLibre(String.valueOf(x+1), fechaYHora));
        }
    }
}

```

Figura 55. Inserción en el mapa de marcadores para cada aparcamiento libre

7. Entrega del producto

Al usar un modelo de ciclo de vida iterativo incremental, al final de cada entrega teníamos ya varios productos entregables. Sin embargo, solo hablaremos aquí de los productos entregables que tenemos después de la última iteración, ya que la mayoría son una actualización de los productos entregables de las anteriores iteraciones, con mejoras y correcciones.

Serán varios los productos que se entregarán del sistema:

- Archivo comprimido (.zip) que contiene el proyecto desplegable en Anypoint Studio.
- Configuración en formato .xml necesaria para generar los eventos simples simulados desde nITROGEN.
- Archivo comprimido que contendrá todos los ficheros en formato .epl que implementan los patrones y esquemas en el lenguaje Esper EPL.
- Apk de la aplicación para dispositivos Android.
- El presente documento, que sirve como memoria de todo el trabajo realizado, documentando todo el proceso de desarrollo del proyecto.

8. Procesos de soporte y pruebas

En este capítulo se hablará de los procesos de soporte que se han llevado a cabo de forma paralela al desarrollo, como la gestión de decisiones y la gestión de riesgos, y de las pruebas, funcionales y no funcionales, llevadas a cabo al finalizar cada iteración.

8.1. Gestión y toma de decisiones

Tener un sistema de toma de decisiones o DSS (Decision Support System) definido es algo fundamental en una empresa. En un proceso de desarrollo software, muchas decisiones son tomadas, ya que para cada problema dado habrá muchas alternativas a la hora de implementar algo.

Establecer una manera consistente en la que tomar las decisiones, documentar el proceso llevado a cabo para tomar una decisión, las alternativas descartadas, las razones que nos llevaron a decidirnos por la opción elegida, etc., nos permitirán aprender de los errores, poder volver atrás y elegir un camino distinto, y prevenir posibles conflictos.

Como este proyecto se ha realizado de forma individual, la mayoría de decisiones se han tomado también individualmente. Sin embargo, las decisiones estratégicas han sido consultada con los tutores del presente TFG, al igual que a veces han servido de ayuda para tomar decisiones en las que no sabíamos elegir entre las distintas alternativas, o proponiendo ellos otras alternativas diferentes.

8.2. Gestión de riesgos

En este capítulo se habla de la gestión de riesgos que se ha realizado durante el desarrollo del proyecto, incluyendo el análisis de los riesgos que se prevén que pueden suceder y un plan de contingencia, estableciendo las soluciones que se deberían adoptar en caso de que sucedieran.

8.2.1. Análisis de riesgos

En la Tabla 9 podemos ver una enumeración de los riesgos analizados, así como la probabilidad de que sucedan y el impacto que tendrían en el desarrollo del proyecto.

Riesgo	Probabilidad	Impacto
R1. Planificación temporal demasiado exigente. No se llega a los objetivos temporales marcados.	Alta	Alto
R2. Enfermedad del personal. El proyecto se retrasa porque el personal enferma.	Alta	Medio
R3. No consecución de algún requisito funcional.	Media	Muy alto
R4. Herramientas de desarrollo inadecuadas. Las herramientas elegidas inicialmente para desarrollar el proyecto resultan un inconveniente a la hora de lograr los objetivos.	Media	Alto
R5. Fallos en los recursos hardware utilizados para el desarrollo del proyecto.	Alta	Alto

Tabla 9. Riesgos analizados

8.2.2. Plan de contingencia

En la Tabla 10 se pueden observar las soluciones que se deberían de adoptar en caso de que alguno de los riesgos anteriormente mencionados sucediera.

Riesgo	Solución a adoptar
R1	Modificar la planificación inicial, con las posibles consecuencias que eso conlleve. No debería suponer un problema ya que dejamos algo de margen, suponiendo que podía pasar estas cosas.
R2	Al igual que anteriormente, habría que modificar la planificación prevista. Si la condición no se alarga más de lo normal tampoco habría problema.
R3	Asumir las consecuencias y las pérdidas, modificar la planificación y el presupuesto.
R4	Explorar alternativas a las herramientas elegidas, modificar planificación si supone un retraso el uso de otra herramienta y el aprendizaje del uso de la misma.
R5	Reparación o sustitución del equipo hardware, modificando el presupuesto para añadir los costes que suponga.

Tabla 10. Plan de contingencia

8.3. Verificación y validación del software

Al llevar a cabo un proceso de desarrollo que sigue un ciclo de vida iterativo incremental, se han realizado pruebas en cada iteración, ejecutando en cada una distintos tipos de pruebas y usando distintas herramientas para ello, según los componentes del sistema que se hayan implementado en la iteración.

Ya que las pruebas realizadas para los distintos componentes difieren mucho una de otras, al igual que las herramientas utilizadas para las pruebas, se dividirán según el componente a probar.

8.3.1. Patrones de eventos

Para probar los patrones de eventos implementados en Esper EPL se ha hecho uso de Esper Notebook [19]. Esta herramienta nos permite crear escenarios simulados en los que simulamos los eventos que suceden y el tiempo que pasa entre uno y otro. Esto nos permite controlar perfectamente qué eventos complejos deben suceder y cuándo, pudiendo probar el funcionamiento de los patrones.

Por poner un ejemplo, en la Figura 56 podemos ver cómo creamos un escenario para probar eventos de aparcamiento.

```
%esperscenario

ParkingEvent = {idMetre = 1, value = 1}
ParkingEvent = {idMetre = 2, value = 1}
ParkingEvent = {idMetre = 3, value = 0}
ParkingEvent = {idMetre = 4, value = 1}
ParkingEvent = {idMetre = 5, value = 0}

t=t.plus(1 second)

ParkingEvent = {idMetre = 1, value = 1}
ParkingEvent = {idMetre = 2, value = 0}
ParkingEvent = {idMetre = 3, value = 0}
ParkingEvent = {idMetre = 4, value = 1}
ParkingEvent = {idMetre = 5, value = 1}

t=t.plus(30 seconds)
```

Figura 56. Escenario creado en Esper Notebook

Creado el escenario, y por supuesto definidos antes los esquemas y patrones de eventos en Esper EPL, ejecutamos todo y obtenemos los eventos complejos que se generan. Podemos ver un ejemplo de los eventos generados en la Figura 57.

Time	Statement	Stream	Output Event
1970-01-01 00:00:00.000	OccupiedParkingSpot	Insert	OccupiedParkingSpot={timestamp=18000000, idMetre=1}
1970-01-01 00:00:00.000	OccupiedParkingSpot	Insert	OccupiedParkingSpot={timestamp=18000000, idMetre=2}
1970-01-01 00:00:00.000	FreeParkingSpot	Insert	FreeParkingSpot={timestamp=18000000, idMetre=3}
1970-01-01 00:00:00.000	OccupiedParkingSpot	Insert	OccupiedParkingSpot={timestamp=18000000, idMetre=4}
1970-01-01 00:00:00.000	FreeParkingSpot	Insert	FreeParkingSpot={timestamp=18000000, idMetre=5}
1970-01-01 00:00:01.000	TotalFreeParkingSpots	Insert	TotalFreeParkingSpots={timestamp=18001000, totalPlazas=2}
	TotalOccupiedParkingSpots	Insert	TotalOccupiedParkingSpots={timestamp=18001000, totalPlazas=3}
1970-01-01 00:00:01.000	OccupiedParkingSpot	Insert	OccupiedParkingSpot={timestamp=18001000, idMetre=1}
1970-01-01 00:00:01.000	FreeParkingSpot	Insert	FreeParkingSpot={timestamp=18001000, idMetre=2}
1970-01-01 00:00:01.000	FreeParkingSpot	Insert	FreeParkingSpot={timestamp=18001000, idMetre=3}
1970-01-01 00:00:01.000	OccupiedParkingSpot	Insert	OccupiedParkingSpot={timestamp=18001000, idMetre=4}
1970-01-01 00:00:01.000	OccupiedParkingSpot	Insert	OccupiedParkingSpot={timestamp=18001000, idMetre=5}

Figura 57. Eventos complejos generados en Esper Notebook

De esta manera y cambiando los escenarios, cambiando el tipo de evento simple generado en ellos así como el tiempo entre un suceso y otro, se comprobó durante la primera iteración el correcto funcionamiento de todos los patrones.

8.3.2. Mule ESB

Sobre la aplicación de Mule ESB también hemos ido realizado pruebas unitarias de cada componente que añadíamos a los distintos flujos, tanto en la segunda iteración descrita en la sección 3.2.2 en la que se implementaron los cuatro flujos que tiene la aplicación, como en la cuarta iteración descrita en sección 3.2.4 donde se realizaron ciertas mejoras en algunos flujos.

Las distintas pruebas que se han realizado son:

- **Recepción de eventos simples**

Mediante el componente Logger, podemos ver por pantalla que se reciben los eventos simples correctamente desde la conexión con RabbitMQ usando el protocolo AMQP. Se puede ver un ejemplo de esto en la Figura 58.

```
.LoggerMessageProcessor: Mensajes{"eventName": "ParkingEvent", "idMetre": 5, "value": 1}
```

Figura 58. Logger de recepción de eventos simples

- **Despliegue de esquemas y patrones en el motor Esper**

En el mismo componente de Esper desarrollado en Java está definido que si un patrón se despliega correctamente, se imprima por consola un mensaje descriptivo, que se puede observar en la Figura 59 y en la Figura 60.

```
*** Adding new EPL @public @buseventtype create json schema TrafficJamEvent (idMetre int, speed double, carPlate string);
```

Figura 59. Logger de despliegue de esquema sobre motor CEP

```
*** Adding new EPL @public INSERT INTO FreeParkingSpot SELECT current_timestamp() as timestamp, idMetre FROM ParkingEvent.win:time(1 seconds) WHERE value = 0;
```

Figura 60. Logger de despliegue de patrón sobre motor CEP

- **Generación de eventos complejos**

Al igual que antes, en el componente de Esper desarrollado en Java, está implementado que si se genera un evento complejo se imprima por consola un mensaje descriptivo, que se puede observar en la Figura 61.

```
** Complex event 'OccupiedParkingSpot' detected: {timestamp=1674636842549, idMetre=5}
```

Figura 61. Logger de generación de evento complejo en motor CEP

- **Consumo, envío y almacenamiento de eventos complejos**

Como hacíamos para probar la recepción de eventos simples, aquí mediante un componente Logger mostramos por consola un mensaje descriptivo si consumimos un evento complejo, como podemos ver en la Figura 62.

```
LoggerMessageProcessor: SALIDA CONSUMIDOR EVENTOS COMPLEJOS ***** {SpeedOver15={carPlate=NVu73S0, idMetre=15, speed=78.0, timestamp=1674636842658}}
```

Figura 62. Logger de consumo de evento complejo por flujo de Mule ESB

Para comprobar después que se han enviado correctamente a la cola de mensajería de RabbitMQ solo deberemos obtener los mensajes recibidos en la cola, como se ve en la Figura 63.

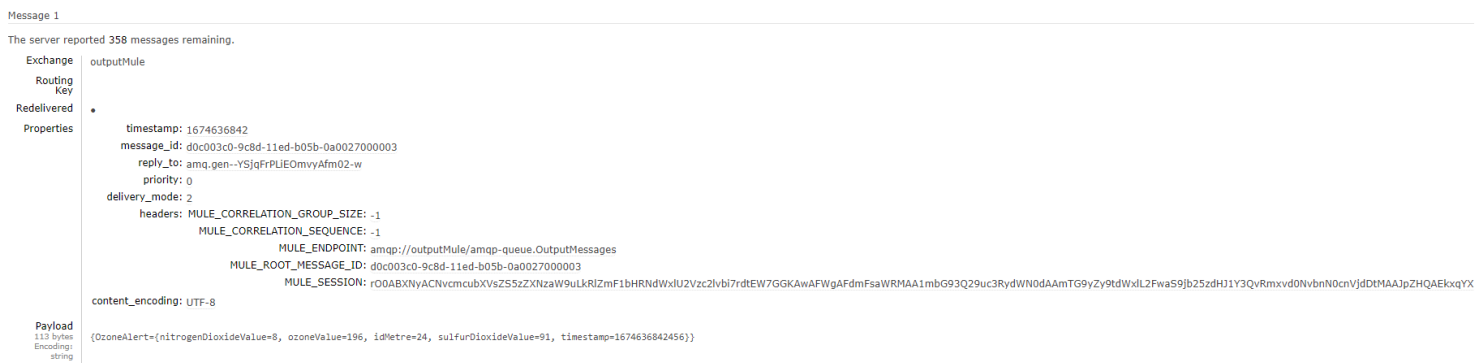


Figura 63. Mensaje recibido en cola de RabbitMQ

Por último, para probar que han sido almacenados en la base de datos, solo deberemos hacer la consulta *SQL SELECT * FROM <tabla de la base de datos>;*

En la Figura 64 podemos comprobar cómo han sido almacenados correctamente los eventos complejos:

```
MariaDB [cadizrespiraeventreport]> select * from polucionevents;
```

id	polucionEvent
1	{SulfurDioxideAlert={nitrogenDioxideValue=7, ozoneValue=110, idMetre=23, sulfurDioxideValue=63, timestamp=1674636842751}}
2	{OzoneAlert={nitrogenDioxideValue=5, ozoneValue=192, idMetre=21, sulfurDioxideValue=62, timestamp=1674636842549}}
3	{OzoneAlert={nitrogenDioxideValue=0, ozoneValue=108, idMetre=24, sulfurDioxideValue=32, timestamp=1674636842456}}
4	{SulfurDioxideAlert={nitrogenDioxideValue=4, ozoneValue=178, idMetre=22, sulfurDioxideValue=79, timestamp=1674636842845}}
5	{OzoneAlert={nitrogenDioxideValue=6, ozoneValue=130, idMetre=21, sulfurDioxideValue=94, timestamp=1674636842658}}
6	{SulfurDioxideAlert={nitrogenDioxideValue=6, ozoneValue=130, idMetre=21, sulfurDioxideValue=94, timestamp=1674636842658}}
7	{SulfurDioxideAlert={nitrogenDioxideValue=5, ozoneValue=192, idMetre=21, sulfurDioxideValue=62, timestamp=1674636842549}}
8	{OzoneAlert={nitrogenDioxideValue=4, ozoneValue=178, idMetre=22, sulfurDioxideValue=79, timestamp=1674636842845}}
9	{OzoneAlert={nitrogenDioxideValue=7, ozoneValue=110, idMetre=23, sulfurDioxideValue=63, timestamp=1674636842751}}
10	{OzoneAlert={nitrogenDioxideValue=8, ozoneValue=196, idMetre=24, sulfurDioxideValue=91, timestamp=1674636842456}}
11	{SulfurDioxideAlert={nitrogenDioxideValue=8, ozoneValue=196, idMetre=24, sulfurDioxideValue=91, timestamp=1674636842456}}
12	{SulfurDioxideAlert={nitrogenDioxideValue=8, ozoneValue=182, idMetre=24, sulfurDioxideValue=66, timestamp=1674636843048}}
13	{OzoneAlert={nitrogenDioxideValue=8, ozoneValue=182, idMetre=24, sulfurDioxideValue=66, timestamp=1674636843048}}
14	{SulfurDioxideAlert={nitrogenDioxideValue=4, ozoneValue=172, idMetre=22, sulfurDioxideValue=53, timestamp=1674636843159}}
15	{OzoneAlert={nitrogenDioxideValue=4, ozoneValue=172, idMetre=22, sulfurDioxideValue=53, timestamp=1674636843159}}
16	{SulfurDioxideAlert={nitrogenDioxideValue=0, ozoneValue=72, idMetre=24, sulfurDioxideValue=45, timestamp=1674636843253}}
17	{SulfurDioxideAlert={nitrogenDioxideValue=7, ozoneValue=54, idMetre=21, sulfurDioxideValue=74, timestamp=1674636843347}}
18	{SulfurDioxideAlert={nitrogenDioxideValue=4, ozoneValue=56, idMetre=25, sulfurDioxideValue=69, timestamp=1674636843456}}
19	{OzoneAlert={nitrogenDioxideValue=8, ozoneValue=180, idMetre=21, sulfurDioxideValue=25, timestamp=1674636843550}}
20	{SulfurDioxideAlert={nitrogenDioxideValue=4, ozoneValue=146, idMetre=21, sulfurDioxideValue=55, timestamp=1674636843657}}
21	{OzoneAlert={nitrogenDioxideValue=4, ozoneValue=146, idMetre=21, sulfurDioxideValue=55, timestamp=1674636843657}}
22	{SulfurDioxideAlert={nitrogenDioxideValue=4, ozoneValue=84, idMetre=24, sulfurDioxideValue=71, timestamp=1674636843750}}

Figura 64. Eventos complejos almacenados en Base de Datos

8.3.3. API REST

Los métodos de la API REST, tanto los implementados en la segunda iteración (descrita en el punto 3.2.2) como los implementados en la cuarta (apartado 3.2.4), han sido probados con Postman , que permite, entre otras cosas, implementar pruebas para APIs.

Por ejemplo, en los métodos implementados en la segunda iteración como el que devolvía el número de eventos producidos, se ha comprobado que la petición es válida (código de estado 200) y que la respuesta sigue el formato correcto, además de que el número de eventos es mayor a 0. Todo esto se puede observar en la Figura 65, donde se ve el código de la prueba y el resultado obtenido.



Figura 65. Prueba de método `getNumberOfEvents` en Postman

Para los métodos implementados en la cuarta iteración, que se encargan de devolver los eventos complejos también se implementaron pruebas en Postman.

Por ejemplo, para comprobar el método que devuelve todos los eventos de aparcamiento almacenados en la base de datos, se prueba que la respuesta a la petición a la API es válida (código 200) y que el JSON que se recibe como respuesta tiene el formato esperado.

En la Figura 66 se puede ver el código de implementación de la prueba, así como abajo los resultados de la misma tras enviar la petición a la API.

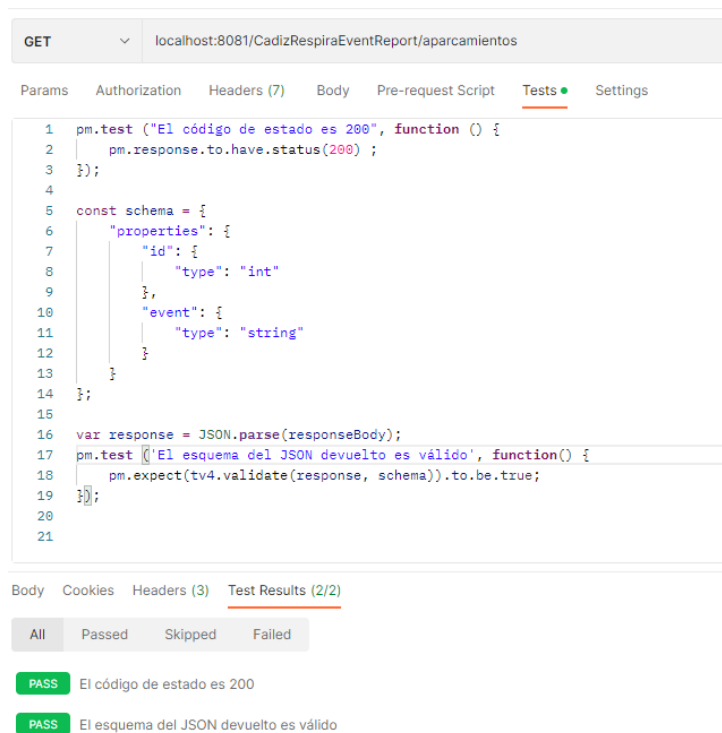


Figura 66. Prueba de método `getAparcamientos` en Postman

8.3.4. Aplicación móvil

Durante el desarrollo de la aplicación, en concreto en la tercera y en la quinta, descritas en el apartado 3.2.3 y 3.2.5, respectivamente, se realizaron pruebas unitarias por cada módulo añadido a la aplicación.

Primero, en la iteración descrita en el apartado 3.2.3, donde se implementaron las distintas pantallas de la aplicación, pero centrándonos sólo en el apartado visual y en la navegación entre ellas, se probó que todos los textos e imágenes se mostraban correctamente, que desde cualquier pantalla se podía navegar a otra a través del menú lateral. También se probó que se podía acceder desde la pantalla de eventos a la lista de eventos de cada tipo (sin mostrar todavía la lista eventos reales) pulsando en cualquiera de los tres botones, y de cada elemento de la lista a la vista de información detallada del evento.

Después, en la última iteración, descrita en el apartado 3.2.5, se probaron cada una de las funcionalidades añadidas:

- Se muestran correctamente las listas de eventos de cada tipo (aparcamiento, tráfico y polución). La lista no se corta sino que se puede “scrollear” hacia abajo para verla completa.
- Se muestra de manera correcta y amigable la información detallada de cada evento complejo generado.
- El mapa de aparcamientos libres se carga correctamente, centrándose en la ciudad de Cádiz y mostrando marcadores verdes únicamente en las plazas de aparcamiento que se encuentran libres. Al pulsar en un marcador, se abre una

ventana de información que indica el identificador del sensor y la hora a la que se generó el evento señalando que no estaba ocupado. También aparecen las opciones para ver la ubicación y generar una ruta hacia la misma, que nos lleva a la aplicación de Google Maps.

- El mapa de atascos se carga también correctamente, mostrando marcadores en las zonas donde se encuentran los atascos que existen actualmente en la ciudad, y con las mismas funcionalidades que la vista anterior.

8.3.5. Pruebas de integración

Una vez se ha probado cada componente por separado, se probó el comportamiento de los componentes al combinarse entre ellos. En nuestro sistema, hay pruebas de integración que carecen de mucho sentido, ya que a lo que precisamente se dedica un ESB como el de Mule, es a integrar datos de distintas fuentes y distintos componentes. Por tanto, los módulos probados en las pruebas unitarias tenían como función precisamente integrar distintos componentes, por lo que la integración de estos ya ha sido probada, como por ejemplo:

- La aplicación Mule se conecta correctamente con una cola de mensajería de RabbitMQ para recibir los eventos simples.
- La aplicación Mule se conecta correctamente con otra cola de mensajería de RabbitMQ para enviar los eventos complejos generados.
- La aplicación Mule se conecta correctamente con una base de datos desplegada en un servidor MySQL local para ejecutar consultas de inserción de los eventos complejos generados.

Sin embargo, sí que ha habido que realizar pruebas de integración de otros componentes que no aparecen en nuestros flujos de Mule ESB. Las pruebas realizadas han verificado lo siguiente:

- La aplicación instalada en un dispositivo Android se conecta sin problema al servicio REST desplegado en el servidor del ordenador donde se ejecuta la aplicación de Mule, permitiendo realizar peticiones a la API. Para que se dé la conexión, el dispositivo móvil y el ordenador deben encontrarse en la misma red.
- El simulador nITROGEN se conecta correctamente con la cola de mensajería de RabbitMQ para enviarle los eventos simples simulados.

8.4. Verificación de la calidad

En esta sección, se explicará cómo se han comprobado los requisitos no funcionales que se recogieron en el apartado 4.2, que hablaba de la garantía de calidad.

Las verificaciones llevadas a cabo son las siguientes:

- Se han simulado desde nITROGEN eventos que no cumplían con el formato esperado y al ejecutar nuestra aplicación Mule ESB, se comprueba que son directamente descartados al no cumplir con la expresión regular del componente *Regex*.

- Los requisitos de interoperabilidad han sido comprobados al realizar las pruebas de integración en las que vimos que los distintos componentes se combinan de forma exitosa.
- En cuanto a la operabilidad, se han realizado observaciones de campos a algunos usuarios de distinto perfil y todos han podido navegar por la aplicación sin problema, ya que la interfaz se les hacía familiar.
- En cuanto a la transferibilidad, mediante el *Device Emulator* de Android Studio, que permite ejecutar la aplicación en diferentes dispositivos Android emulados, se ha podido comprobar que la interfaz se adapta distintas resoluciones de pantalla.
- Se han probado los tiempos de carga mediante la herramienta Firebase Performance Monitoring [20], un servicio que ayuda a obtener información del rendimiento de aplicaciones en tiempo real.
- Se ha llevado a cabo un análisis estático del código mediante exploración, comprobando que efectivamente estaba debidamente documentado, y era fácilmente extensible.

9. Conclusiones y trabajo futuro

En este capítulo haremos una valoración final de cómo ha ido el desarrollo del proyecto y lo que nos ha supuesto personalmente, después haremos un análisis del grado de cumplimiento de los objetivos marcados inicialmente, para acabar hablando de las posibles mejoras del mismo que se podrían realizar en un futuro.

9.1. Valoración del proyecto

La idea de este proyecto nació con la intención de crear un sistema que ayudara a monitorizar los niveles de contaminación de una ciudad, focalizado en Cádiz y abarcando sobre todo el principal causante de que se superen en la ciudad a menudo los niveles recomendados de contaminación, la movilidad.

La gran presencia de vehículos en una ciudad con una superficie pequeña hace que diariamente se produzcan atascos, que hacen que se emitan más emisiones que cuando hay una circulación normal. Además, todo esto es alimentado también por la falta de aparcamiento, que hace que muchos conductores se pasen un gran tiempo en circulación simplemente buscando aparcamientos, lo que provoca también que se contamine más de lo necesario.

Una herramienta que ayudara a monitorizar estas situaciones y así poder actuar rápidamente para reducir su impacto: disolviendo atascos, detectando niveles de polución de aire inusuales o ayudando a los conductores en la tarea de encontrar aparcamiento, mejoraría la calidad de vida de la gente a la vez que aportaría en la más que necesaria tarea de poner freno al cambio climático.

Además, el hecho de usar tecnologías emergentes y de enorme potencial como el procesamiento de eventos complejos, aporta un gran valor a esta herramienta y haría a la ciudad seguir la tendencia de las Smart Cities, implantando un modelo que podría servir después de referencia para otras ciudades.

Personalmente, este proyecto me ha aportado bastante a nivel académico, ya que, aunque se han usado herramientas y lenguajes ya conocidos como Java, Android Studio o SQL, también se ha profundizado en tecnologías como el Procesamiento de Eventos Complejos y los Buses de Servicios Empresariales que hasta hace apenas unos meses eran completamente desconocidos.

Como conclusión, este proyecto ha sido muy útil, no sólo por el aprendizaje de nuevas tecnologías y en tener una pequeña aproximación a lo que es el proceso íntegro de desarrollo de un proyecto de ingeniería, sino para ser conscientes de todo lo que pueden aportar las tecnologías y los sistemas de la información a la sociedad, y más en concreto al desarrollo de las Smart Cities y la lucha contra el cambio climático.

9.2. Cumplimiento de los objetivos propuestos

Al comienzo del proyecto, se establecieron unos objetivos concretos que debían cumplirse para dar el proyecto como acabado. Ahora, valoraremos el grado de cumplimiento de cada uno de esos objetivos, para comprobar que el proyecto ha sido ejecutado con éxito:

- **Implementar una arquitectura software que permita tratar una serie de datos relacionados con el tráfico, el aparcamiento y la contaminación atmosférica, leyendo los datos de una cola de mensajería y procesándolos en tiempo real con un motor CEP** → Este era el objetivo principal del sistema, que podemos decir que se ha cumplido satisfactoriamente. Haciendo uso de Mule ESB, recibimos una serie de eventos desde una cola de RabbitMQ, procesamos esos eventos en el motor CEP de Esper integrado en nuestro sistema, generándose eventos complejos que nos proporcionan información más valiosa.
- **Dotar al sistema de persistencia, almacenando los eventos complejos, para poder acceder en cualquier momento al registro de las situaciones detectadas** → Este objetivo también se ha logrado. Desde nuestro sistema desarrollado en Mule ESB, consumimos los eventos complejos generados en el motor CEP y por un lado almacenamos en una base datos, y por otro lado los enviamos a una cola de mensajes para que puedan ser consumidos por un sistema externo.
- **Desarrollo de una aplicación Android que permita visualizar situaciones de interés detectadas de una manera amigable para el usuario** → También ha sido logrado exitosamente este objetivo. Nuestra aplicación para dispositivos Android muestra la información extraída de los eventos complejos de manera amigable y comprensible.
- **Localizar geográficamente en un mapa de la ciudad las plazas de aparcamiento libres y los atascos existentes a través de la aplicación Android** → Este objetivo también ha sido cumplido. Haciendo uso de la API de Google Maps, mostramos los aparcamientos libres y los atascos detectados en nuestro sistema sobre el mapa de la ciudad.

9.3. Trabajo futuro

Desde un principio, se podía intuir claramente que las posibilidades que podía llegar a ofrecer un sistema como el desarrollado eran infinitas, también fue por esta precisa razón por la que se acotaron claramente los objetivos, para no intentar abarcar demasiado dada la limitación de tiempo y recursos disponibles.

- Instalación de sensores en la ciudad de manera que se sustituyan los datos simulados que actualmente recibe nuestro sistema por datos reales. Habría que instalar sensores de los tres tipos:
 - Sensores en todas las plazas de la bolsa de aparcamiento de la ciudad que detecten si hay un vehículo estacionado en ella o no.
 - Sensores similares a los radares de la DGT instalados sobre los semáforos de la ciudad, sobre todo en los puntos donde sea más habitual que se produzca un atasco.
 - Sensores repartidos por distintos puntos de la ciudad que midan los niveles de polución del aire.
- Desarrollo de aplicación para dispositivos con iOS como sistema operativo, ya que hay una gran cantidad de usuarios para los que nuestro sistema no será accesible al estar solo disponible para Android.

- Desarrollo de una aplicación web que presente también la información de los eventos a usuarios que quieran acceder desde un PC o desde el navegador web de sus dispositivos móviles sin tener que descargarse la aplicación.
- Implementación de un registro de usuarios en la aplicación, que permita que diferenciar el uso de la aplicación por parte de autoridades que por el resto de aplicación. Las primeras podrían acceder así a información de carácter confidencial que estaría oculta para el resto de usuarios.

Bibliografía

- [1] P. S. M., «Cádiz, irrespirable: mantiene dos de los diez focos de más contaminación de Andalucía», *lavozdelsur.es*, 28 de junio de 2022. Accedido: 18 de enero de 2023. [En línea]. Disponible en:
https://www.lavozdelsur.es/actualidad/ecologia/cadiz-irrespirable-mantiene-dos-diez-focos-mas-contaminacion-andalucia_279018_102.html
- [2] Ecologistas en Acción, «La calidad del aire en el Estado español durante 2021», jun. 2022. Accedido: 18 de enero de 2023. [En línea]. Disponible en:
<https://www.ecologistasenaccion.org/wp-content/uploads/2022/06/informe-calidad-aire-2021.pdf>
- [3] J. Boubeta-Puig, J. Rosa-Bilbao, y J. Mendling, «CEPchain: A graphical model-driven solution for integrating complex event processing and blockchain», *Expert Systems with Applications*, vol. 184, p. 115578, dic. 2021, doi: 10.1016/j.eswa.2021.115578.
- [4] «Esper - EsperTech». <https://www.espertech.com/esper/> (accedido 25 de enero de 2023).
- [5] MuleSoft, ¿Qué es Mule ESB? <https://www.mulesoft.com/es/resources/esb/what-mule-esb> (accedido 18 de enero de 2023).
- [6] RabbitMQ, *Messaging that just works — RabbitMQ*. <https://www.rabbitmq.com/> (accedido 18 de enero de 2023).
- [7] Amazon Web Services, ¿Qué es API RESTful? | Guía sobre API RESTful para principiantes / AWS. <https://aws.amazon.com/es/what-is/restful-api/> (accedido 18 de enero de 2023).
- [8] MySQL, *MySQL*. <https://www.mysql.com/> (accedido 18 de enero de 2023).
- [9] Android Developers, *Download Android Studio & App Tools*. <https://developer.android.com/studio> (accedido 18 de enero de 2023).
- [10] «Download the Latest Java LTS Free». <https://www.oracle.com/es/java/technologies/downloads/> (accedido 25 de enero de 2023).
- [11] MuleSoft, *Anypoint Studio | Entorno de desarrollo integrado (IDE)*. <https://www.mulesoft.com/es/platform/studio> (accedido 18 de enero de 2023).
- [12] «Eclipse Downloads | The Eclipse Foundation». <https://www.eclipse.org/downloads/> (accedido 25 de enero de 2023).
- [13] <https://ucase.uca.es/nITROGEN/> (Accedido 25 de enero de 2023).
- [14] A. Garcia-de-Prado, G. Ortiz, J. Hernández, y E. Moguel, «Generación de Datos Sintéticos para Arquitecturas de Procesamiento de Datos del Internet de las Cosas».
- [15] «Balsamiq. Rapid, Effective and Fun Wireframing Software | Balsamiq». <https://balsamiq.com/> (accedido 25 de enero de 2023).
- [16] «Postman API Platform | Sign Up for Free», *Postman*. <https://www.postman.com> (accedido 24 de enero de 2023).
- [17] «Desarrolladores de Android», *Android Developers*. <https://developer.android.com/design?hl=es-419> (accedido 21 de enero de 2023).
- [18] «Color - Material Design», *Desarrollador Android*, 6 de marzo de 2015. <https://desarrollador-android.com/material-design/disenio-material-design/estilo/color/> (accedido 25 de enero de 2023).
- [19] «EsperTech - Leader in Complex Event Processing». <https://esperonline.net/> (accedido 25 de enero de 2023).

- [20] «Firebase Performance Monitoring», *Firebase*.
<https://firebase.google.com/docs/perf-mon?hl=es-419> (accedido 27 de enero de 2023).

A. Manual de instalación

i. Aplicación móvil

El único requisito para instalar la aplicación móvil es tener un dispositivo con una versión de Android igual o superior a Android 8.0 (nivel de API ≥ 26).

Podremos transferir nuestra aplicación al dispositivo móvil directamente desde el proyecto de Android Studio, sin tener que subir la aplicación a ningún repositorio, de dos formas:

- Mediante conexión USB, para lo que deberemos tener activada la depuración USB.
- Mediante conexión inalámbrica, deberemos tener activada la depuración inalámbrica.

Para ambos métodos deberemos tener activadas las opciones de desarrollador y tener activados los permisos para instalar aplicaciones con origen desconocido.

Según el modelo del dispositivo y la versión de Android que use, estas dos opciones se activan de una forma u otra, ya que cambian las opciones de los ajustes.

Se muestra a continuación un ejemplo de cómo activar la depuración USB y la depuración inalámbrica, pero puede variar como se ha comentado, se aconseja buscar antes cómo se activan estas opciones,

1. Entra en “Ajustes” > “Acerca del dispositivo” o “Acerca del teléfono” > “Info software
2. Presiona 7 veces de forma consecutiva la opción “Número de compilación”
3. Regresa a “Ajustes”
4. Presiona “Opciones de desarrollador”
 - a. Presiona “Depuración de USB”
 - b. Presiona “Depuración inalámbrica”

Si la opción elegida es la depuración inalámbrica, tienes dos opciones, usar un código QR o usar un código de vinculación.

Entonces, en Android Studio pulsamos en Run > Select Device... > Pair Devices Using Wi-Fi y elegiremos también la opción. En Android Studio nos aparecerá un código QR, que escanaremos desde el dispositivo, o un código de vinculación, que introduciremos en nuestro dispositivo.

Una vez hecho esto, se construirá la aplicación y se instalará nuestro dispositivo. Si no tenemos activada la opción de instalar aplicaciones con orígenes desconocido, nos solicitará los permisos, aunque normalmente podremos activarlos en el momento.

Si no, podremos activar la opción antes de la instalación de esta forma (también puede variar según el modelo y la versión del SO):

1. Entra en “Ajustes” > “Protección de la privacidad”
2. Selecciona “Permisos especiales”
3. Elige la opción “Instalar aplicaciones desconocidas”

4. Elige la aplicación desde la que se instala la aplicación (explorador de archivos en nuestro caso) y concédele permiso.

ii. RabbitMQ

Para el uso de RabbitMQ, es necesario tener instalado Erlang, para ello, descargaremos la última versión de 64 bits para Windows disponible en el árbol de versiones de la página oficial: https://erlang.org/download/otp_versions_tree.html

Para el uso de RabbitMQ, es necesario tener instalado Erlang, para ello, descargaremos la última versión de 64 bits para Windows disponible en el árbol de versiones de la página oficial: https://erlang.org/download/otp_versions_tree.html

Hecho esto, actualizamos las variables de entorno del sistema, creando una nueva variable llamada ERLANG_HOME y situada en C:\Program Files\Erlang OTP.

Después, añadimos al PATH %ERLANG_HOME%\bin.

Instalada la versión de Erlang, y habiendo permitido las comunicaciones de Erlang en el cortafuegos cuando se solicite, ya podemos pasar a descargar la última versión del instalador de RabbitMQ, rabbitmq-server-{version}.exe desde <https://www.rabbitmq.com/install-windows.html>

Una vez instalado, también tendremos que editar las variables de entorno del sistema, creando la variable RABBITMQ_SERVER en C:\Program Files\RabbitMQServer\rabbitmq_server-{versión instalada}

Hecho esto, añadimos al PATH %RABBITMQ_SERVER%\sbin.

Con esto ya estará instalado el servidor de RabbitMQ y podemos comprobar el estado del servicio con rabbitmqctl status. Para parar el servicio podremos usar rabbitmqctl stop y para iniciarlo, rabbitmq-server start.

Para acceder a la consola gráfica, accedemos a <http://localhost:15672/> y usamos el usuario y contraseña por defecto “guest”.

iii. nITROGEN

Para la simulación de los datos deberemos descargar el archivo comprimido de nITROGEN entrando en <https://ucase.uca.es/nITROGEN/> y pulsando en “Download now”.

Una vez descargado, descomprimos el archivo descargado y ejecutamos el archivo “nITROGEN_GUI.exe”.

iv. Base de datos

Para almacenar los eventos complejos en la base de datos, deberemos descargar e instalar una base de datos SQL como MariaDB, descargable desde <https://mariadb.org/>

Una vez descargado en instalador, lo abrimos, hacemos click en “Siguiente”, aceptamos los términos de licencia, volvemos a hacer click dos veces en “Siguiente”, introducimos una contraseña para el root (la que se usa en la base de datos del proyecto es isi), marcamos utilizar UTF y procedemos a instalarla.

Una vez hecho esto, aceptamos las peticiones del cortafuegos, y pasamos a la creación de la base de datos y las tablas.

Para ello, abrimos la cmd, vamos a la carpeta donde instalamos la base de datos, escribimos `mysql -u root -p`, introducimos la contraseña establecida anteriormente y ejecutamos las siguientes sentencias:

```
CREATE DATABASE IF NOT EXISTS CadizRespiraEventReport;
USE CadizRespiraEventReport;
CREATE TABLE IF NOT EXISTS AparcamientoEvents (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, aparcamientoEvent VARCHAR (4096));
CREATE TABLE IF NOT EXISTS TraficoEvents (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, traficoEvent VARCHAR (4096));
CREATE TABLE IF NOT EXISTS PolucionEvents (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, polucionEvent VARCHAR (4096));
```

Hecho esto tendremos creadas las tres tablas que usaremos en nuestro sistema, podemos comprobarlo mediante la sentencia:

```
show tables;
```

v. Anypoint Studio y proyecto Mule

Para usar Anypoint Studio es necesario instalar JDK8, por ejemplo desde Adoptium <https://adoptium.net/es/temurin/releases/?version=8>

Una vez instalado el Java Development Kit 8, descargaremos la versión 6.6 de Anypoint Studio (para la versión 7 no hay servidor gratuito) entrando en <https://www.mulesoft.com/lp/dl/anypoint-mule-studio>, seleccionando “Previous versions”, y registrándonos.

Después de la descarga, lo descomprimos, abrimos el archivo de configuración AnypointStudio.ini y añadimos, después de la sentencia del launcher y antes de los vmargs, la siguiente sentencia, con la ruta donde hemos instalado JDK8:

```
-vm
C:/[Ruta de instalación de Java]/bin/javaw.exe
```

Hecho esto, abrimos Anypoint Studio y elegimos una ruta para el workspace. Cuando el cortafuegos solicite permitir el acceso a Anypoint Studio se lo concederemos.

Después, haremos clic en Help → Install new Software. En la ventana que se abrirá abrimos el desplegable “Work with” y seleccionamos Mule Runtimes for Anypoint Studio. Desplegamos entonces las opciones de Anypoint Studio Community Runtime (servidor gratuito) y seleccionamos Mule ESB Server Runtime 3.9.0 CE.

Se procederá a la instalación del servidor de Mule ESB y una vez finalizada, Anypoint Studio nos pedirá que reiniciemos la aplicación.

Una vez reiniciada, podemos proceder a importar el proyecto pulsando en File → Import → Anypoint Studio → Anypoint Studio generated Deployable Archive (.zip) y seleccionando nuestro proyecto.

Hecho esto e importado el proyecto, haciendo click sobre el proyecto y pulsando en Run As → Mule Application, podemos ejecutarlo sin problemas.

B. Manual de usuario

Una vez tengamos la aplicación instalada y todos los componentes necesarios ejecutándose, entraremos en la aplicación haciendo click en su icono.

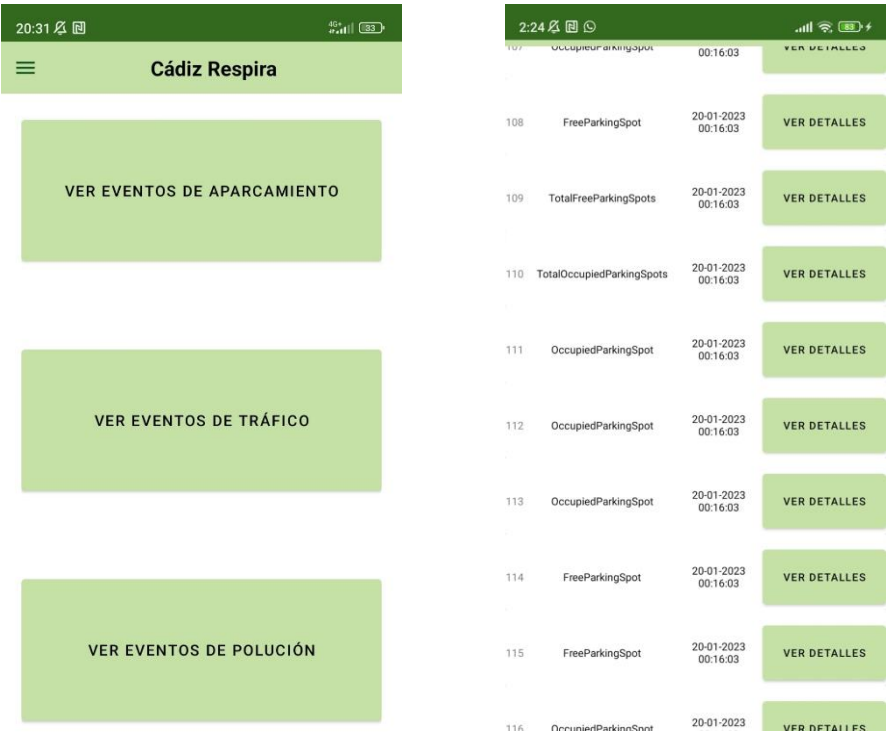
Al acceder a la aplicación, nos encontraremos con la pantalla de bienvenida, que nos muestra información de la app: logo, nombre, objetivos, autor...



Al pulsar sobre el icono de las tres barras (metáfora visual que significa menú), se abrirá el menú principal de la aplicación y podremos navegar a la pantalla que queramos pulsando en su nombre o su icono.



Al acceder a la sección de Eventos, se mostrarán los tipos de eventos, y al pulsar en uno de los botones, se mostrará la lista de eventos del tipo elegido.



Al pulsar en el botón que indica “Ver detalles”, se mostrará información detallada del evento de la siguiente manera:

2:25   	
 Cádiz Respira	
ID Evento	106
Tipo Evento	FreeParkingSpot
Fecha y hora	20-01-2023 00:16:03
ID Sensor	4

2:39   	
 Cádiz Respira	
ID Evento	81
Tipo Evento	SulfurDioxideAlert
Fecha y hora	01-01-1970 01:00:00
ID Sensor	25
Nivel de NO2	108
Nivel de SO2	51
Nivel de O3	108

Si accedemos a la sección “Aparcamientos”, se mostrará un mapa de Google con los aparcamientos que hay libres en la ciudad, de la siguiente manera:



Al pulsar en el marcador de una de las plazas de aparcamiento, se mostrará información del evento, y se mostrarán dos opciones en la esquina inferior derecha, para iniciar una ruta hacia la localización de la plaza de aparcamiento y para abrir la localización en Google Maps, como se ve en la siguiente imagen:



C. Boceto de la aplicación

Como ya explicamos en el apartado 5.4, el boceto realizado con Balsamiq que se podrá observar a partir de la siguiente página es navegable. Esto permite que aparte de ver la interfaz de usuario de la aplicación, podamos comprobar la navegabilidad de la misma.

Sin embargo, como esto nos sirve como prototipo de la aplicación, para hacernos una idea de lo que después queremos implementar, no han sido prototipadas todas las pantallas, ya que muchas son prácticamente iguales (cambiando solo el tipo de evento).

Además, no todos los botones son funcionales. Por tanto, para navegar por el boceto debemos tener en cuenta las siguientes consideraciones:

- Desde cualquier pantalla se puede abrir el menú lateral pulsando en el símbolo.
- Todos los enlaces del menú lateral son navegables.
- En la pantalla de eventos, solo es navegable el botón de aparcamientos.
- En la lista de eventos de aparcamientos, a la que se llega tras pulsar el botón que acabamos de mencionar, solo podremos acceder a los detalles del primero.
- En el mapa de aparcamientos, si pulsamos en el marcador de más arriba, se abrirá una ventana de información. Esto no está implementado en el mapa de atascos ya que es igual.
- El menú lateral no se puede cerrar como tal, se debe navegar a alguna vista a través de los enlaces.



Cádiz Respira

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis lobortis dui vel quam pharetra, non rutrum sapien consequat. Maecenas sed lacinia leo. Integer turpis metus, rhoncus et odio at, aliquam posuere sem.

Nam sed **convallis lectus**, vitae sodales massa. Donec varius erat leo, a scelerisque sapien posuere blandit. Curabitur eget mollis velit. Sed eget erat vel ligula consequat ornare lobortis nec risus. Pellentesque habitant morbi tristique senectus et netus et malesuada.



Manuel Cano Crespo



12:25



ira

Cádiz Respira



Inicio



Eventos



Aparcamientos



Atascos

pharetra,
s sed
et odio

massa.
ien
lit. Sed
ortis nec
ue





Cádiz Respira

Aparcamiento





Tráfico

Polución





Cádiz Respira

-  Inicio
-  Eventos
-  Aparcamientos
-  Atascos









Cádiz Respira

Id	Evento	Fecha y hora	Ver detalles
1	FreeParkingSpot	19/01/2023 19:12:53	Ver detalles
2	FreeParkingSpot	19/01/2023 19:12:53	Ver detalles
3	OccupiedParkingSpot	19/01/2023 19:12:53	Ver detalles
.....			





Cádiz Respira

-  Inicio
-  Eventos
-  Aparcamientos
-  Atascos

detalles

[detalles](#)

detalles

detalles

...





12:25







Cádiz Respira

ID Evento	1
Tipo Evento	FreeParkingSpot
Fecha y hora	19/01/2023 19:12:53
Id Sensor	17





Cádiz Respira

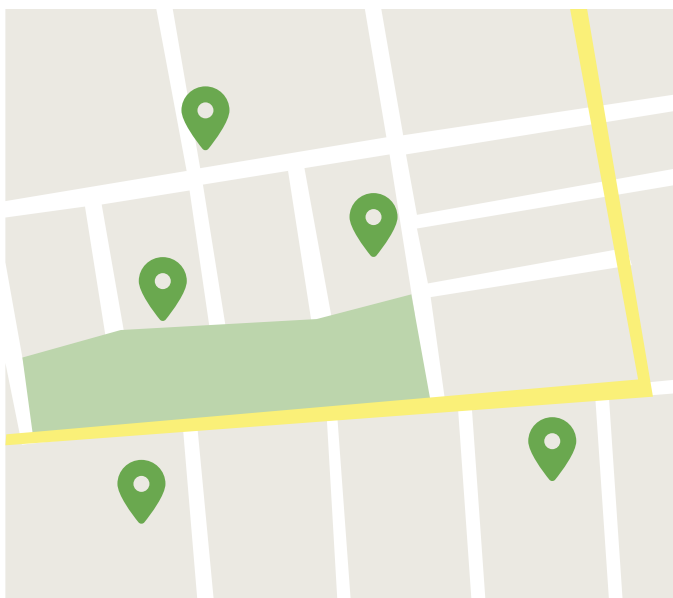
-  Inicio
-  Eventos
-  Aparcamientos
-  Atascos





Aparcamientos libres

Los usuarios de la aplicación podrán encontrar los aparcamientos libres en la zona de estudio. La información se actualiza en tiempo real, por lo que los usuarios podrán encontrar los aparcamientos libres en la zona de estudio.





Cádiz Respira

res

res

res

res

res

res



Inicio



Eventos



Aparcamientos



Atascos



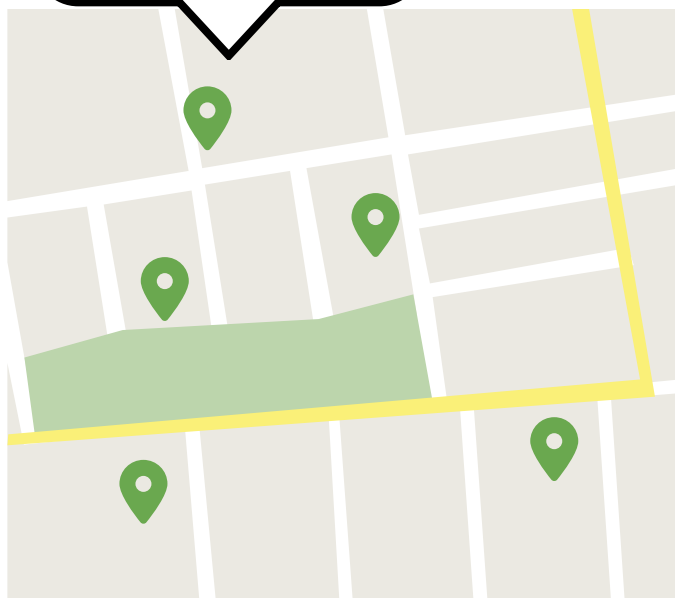
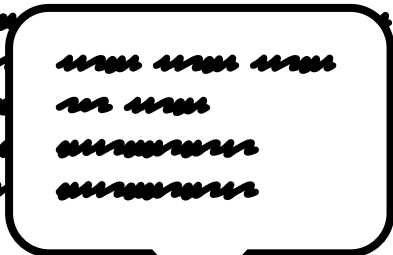


12:26



Cádiz Respira

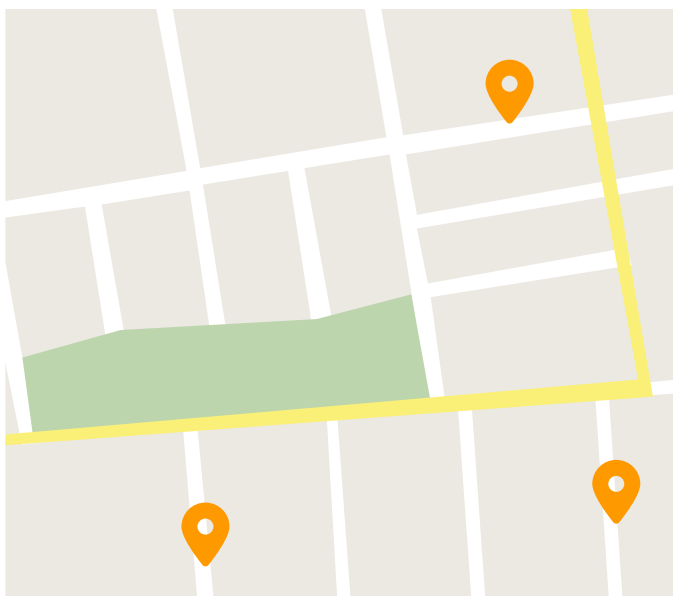
Aparcamientos libres





Atascos

Atascos en la zona de Cádiz Respira. Se muestran los puntos de congestión en la zona de estudio.









12:25



Cádiz Respira

-  Inicio
-  Eventos
-  Aparcamientos
-  Atascos

