

#### **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"><li>• Intel Core i7-9750H</li><li>• NVIDIA GeForce GTX 1650</li><li>• RAM 16 GB</li><li>• 512 GB SSD</li><li>• 17.3"</li></ul>	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.

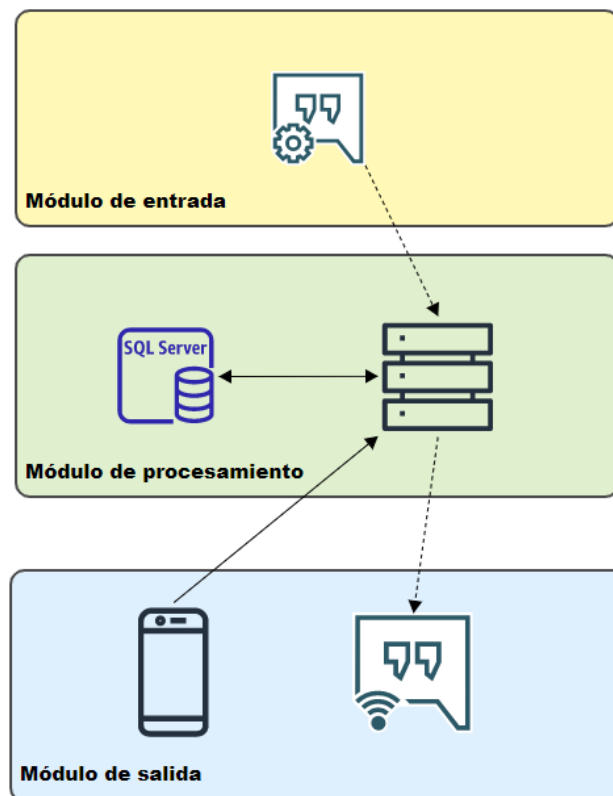


Figura 6. Arquitectura física

#### 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.

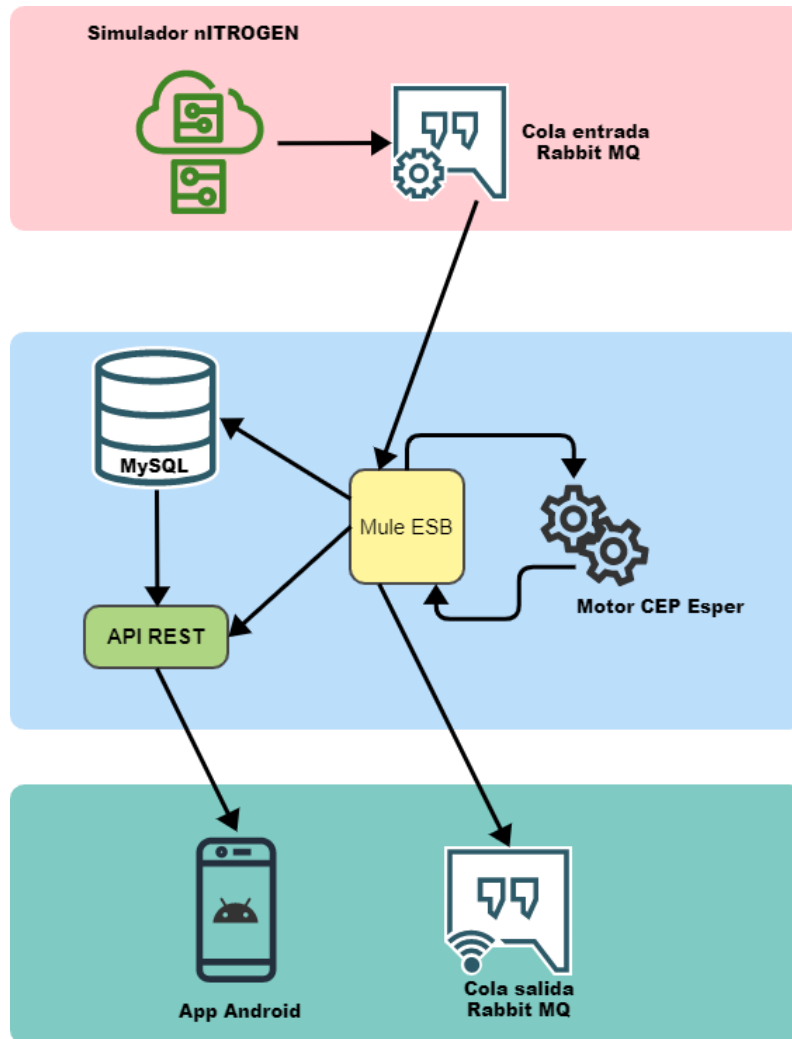


Figura 7. Arquitectura lógica

### 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:

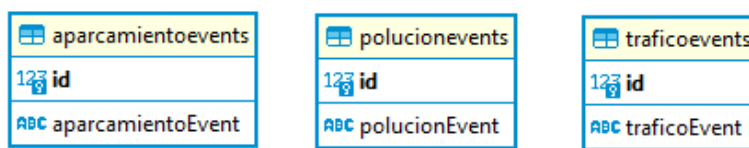
- **Cola de salida de RabbitMQ:** El módulo de procesamiento envía los datos ya procesados (eventos complejos) a esta cola de mensajería de RabbitMQ, de forma que puedan ser leídos por un usuario o consumidos por algún otro proceso.
- **Aplicación para dispositivos Android:** La aplicación recibe, mediante 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.

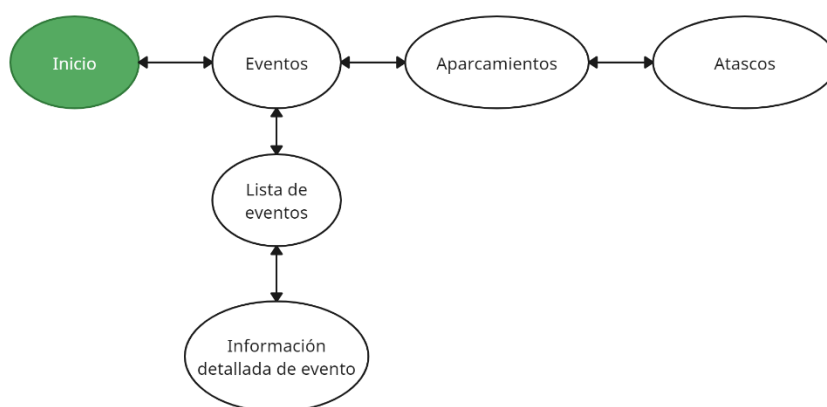


Figura 9. Diagrama de navegación de la App

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 todos 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 (NO<sub>2</sub>), el nivel de dióxido de azufre (SO<sub>2</sub>) y el nivel de ozono (O<sub>3</sub>). 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 NO<sub>2</sub> es mayor a 10 µg/m<sup>3</sup>, 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 SO<sub>2</sub> mayor a 40 µg/m<sup>3</sup>, 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 O<sub>3</sub> es mayor a 100 µg/m<sup>3</sup>, 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

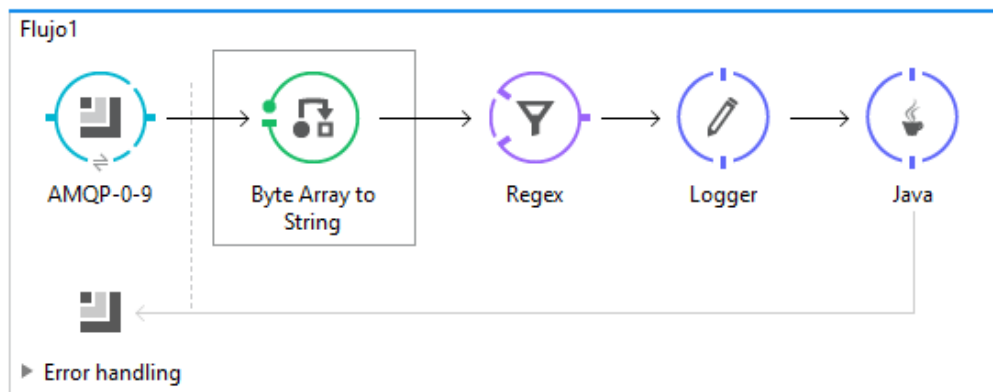


Figura 35. Flujo 1 Mule ESB "Recepción y tratamiento de eventos simples"

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

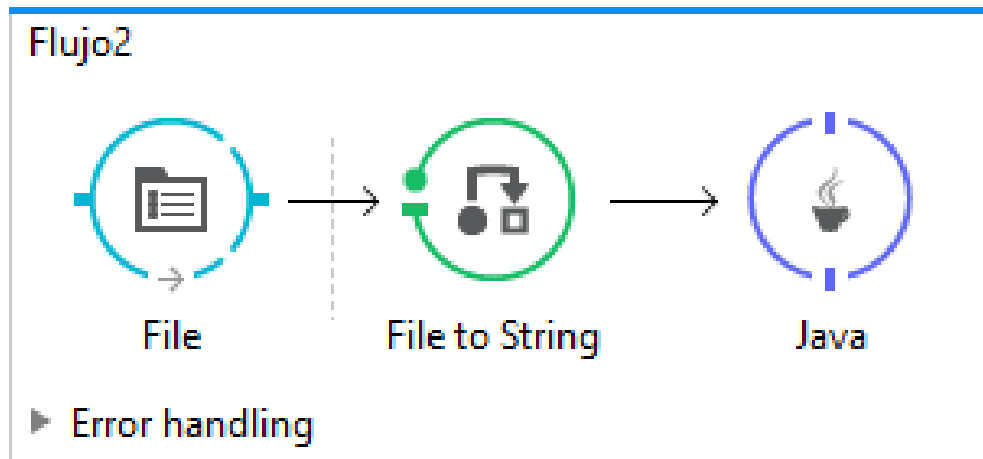


Figura 36. Flujo 2 Mule ESB "Despliegue de patrones en el motor CEP"

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

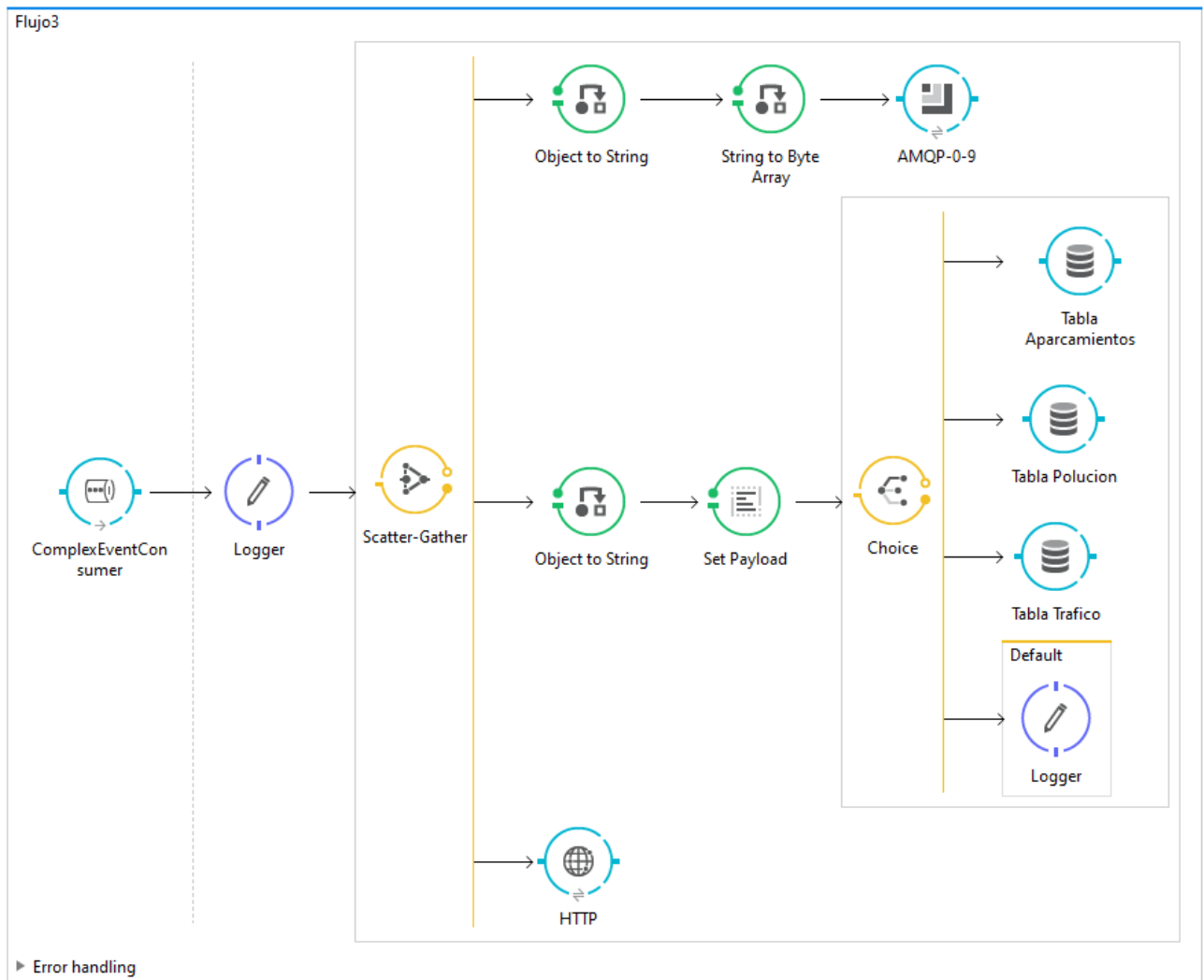


Figura 37. Flujo 3 Mule ESB "Almacenamiento y envío de eventos complejos"

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.

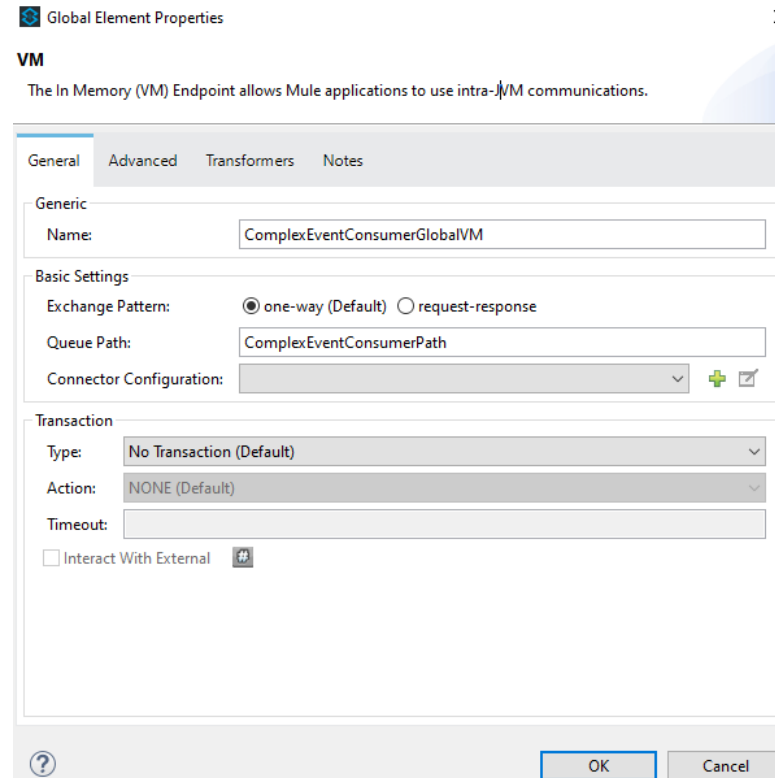


Figura 38. Configuración VM Endpoint

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

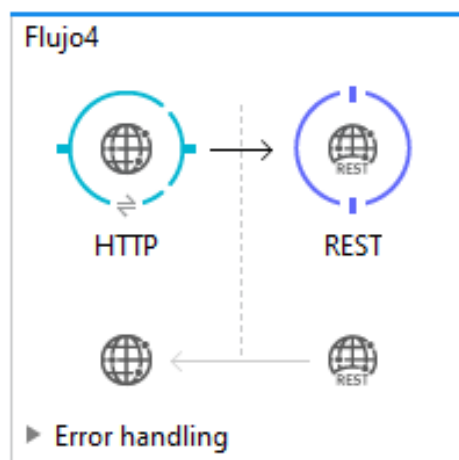
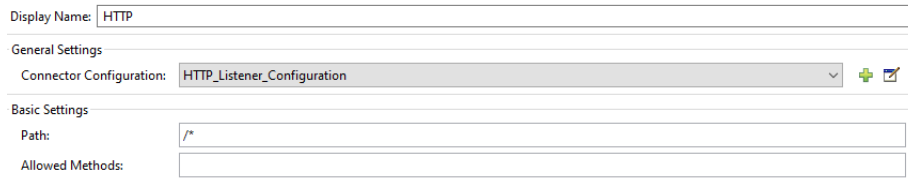


Figura 39. Flujo 4 Mule ESB "Creación de servicio REST"

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

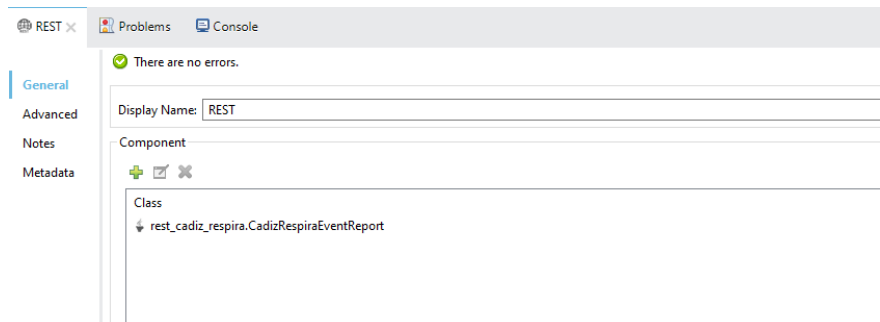
Path: /\*

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

There are no errors.

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,"Tráfico",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
<b>R1.</b> Planificación temporal demasiado exigente. No se llega a los objetivos temporales marcados.	Alta	Alto
<b>R2.</b> Enfermedad del personal. El proyecto se retrasa porque el personal enferma.	Alta	Medio
<b>R3.</b> No consecución de algún requisito funcional.	Media	Muy alto
<b>R4.</b> Herramientas de desarrollo inadecuadas. Las herramientas elegidas inicialmente para desarrollar el proyecto resultan un inconveniente a la hora de lograr los objetivos.	Media	Alto
<b>R5.</b> 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.