Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Procesador de flujos REST en el entorno sanitario

Autor: Pablo Gallegos Jiménez
Tutor: Isabel Román Martínez

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Procesador de flujos REST en el entorno sanitario

Autor:
Pablo Gallegos Jiménez

Tutores:
Isabel Román Martínez
Profesora colaboradora

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Trabajo Fin de Máster: Procesador de flujos REST en el entorno sanitario

Autor: Pablo Gallegos Jiménez

Tutor: Isabel Román Martínez

El tribunal nombrado para juzgar el Trabajo arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El secretario del Tribunal

Resumen

Este proyecto consiste en la personalización de un servicio de procesado de flujo como intermediario de peticiones realizadas a un servidor de información sanitaria. Para ello se deberán procesar y modificar las diferentes peticiones y en función de este procesado, generar alertas mediante emails o mensajes a una cola.

Para su desarrollo se utilizará el Stream Processor de WSO2, que es una de las herramientas de procesado de flujo más usadas actualmente. Para facilitar la comprensión del procesado final y para ayudar a comprender todas las funcionalidades que el Stream Processor tiene, se incluirán tres escenarios de pruebas, de menor complejidad que el escenario utilizado con el servidor sanitario.

Como servidor, se va a hacer uso de un servidor público que cumple el estándar fhir, el cual es el último estándar desarrollado y promovido por la organización internacional HL7. Y como cola de mensajes se hará uso de una cola MQTT.

Abstract

This project consists of the personalization of a flow processing service as an intermediary of requests made to a health information server. To do this, the different requests must be processed and modified and, depending on this processing, generate alerts by email or messages to a queue.

This will be done using the WSO2 Stream Processor, which is currently one of the most commonly used flow processing tools. To facilitate the understanding of the final processing and to help understand all the functionalities that the Stream Processor has, three test scenarios will be included, of less complexity than the scenario used with the health server.

A public server that meet the FHIR standard will be used as a server, which is the latest standard developed and promoted by the international organization HL7. And an MQTT queue will be used as a message queue.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

CRUD	Create, Read, Update y Delete
FHIR	Fast HealthCare Interoperability Resources
HL7	Health Level Seven
HTTP	HyperText Transfer Protocol
IDE	Entorno de Desarrollo Integrado
IoT	Internet of Things
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
REST	REpresentational State Transfer
SOA	Arquitectura Orientada a Servicios
SP	Stream Processor
SQL	Structured Query Language
URI	Identificador de recursos uniforme
URL	Localizador de recursos uniforme
WSO2 SP	WSO2 Stream Processor
WWW	World Wide Web
XML	eXtensible Markup Language

1 INTRODUCCIÓN

Cada vez generamos una mayor cantidad de datos, según IBM cada 24 horas se generan 2´5 millones de Terabytes en la red. Gracias a la tecnología Big Data podemos tratar este gran volumen de datos y obtener información que nos permita tomar mejores decisiones. Es en el sector de la salud donde este proyecto va a incidir, haciendo uso de estándares y servidores específicos para ello, con la intención de obtener información de los datos que nos proporcionan.

1.1 Objetivos y motivación

Los tiempos cambian y cada vez generamos una mayor cantidad de datos, desde una búsqueda en la red hasta una solicitud de una cita médica, casi cualquier paso que damos hoy en día genera datos. Gracias a la tecnología de análisis de Big Data podemos tratar este gran volumen de datos y obtener información que ni siquiera sabíamos que teníamos.

Uno de los sectores donde más hincapié se hace mundialmente en la importancia que podría llegar a tener esta información es el ámbito de salud. Sin necesidad de datos médicos, únicamente con las búsquedas en internet realizadas por las personas geolocalizadas en una zona se puede detectar la aparición de una gripe [1].

Si esto es posible con información de carácter general, es fácil imaginar lo que se puede conseguir si los datos que se procesan provienen de fuentes específicas y la ponemos a disposición de personal sanitario cualificado para ello.

Proyectos como BigMedilytics [2] del Horizonte 2020 [3] a nivel europeo o Savana [4] en España confirman que la implantación del Big Data en el entono sanitario es un hecho. Estos proyectos basados en la generación de algoritmos para mejorar el diagnóstico de enfermedades y la digitalización de las historias clínicas -entre otras aplicaciones- abren nuevas líneas de trabajo, pero también se enfrentan a nuevos retos.

España es uno de los países con mayor inversión en los sistemas informáticos del ámbito sanitario, no obstante, no se ha trabajado mucho en la explotación y transformación de la información y los datos existentes en esos sistemas informáticos [5].

Estos datos podrían ser tratados con el fin de obtener información a través de diversas aplicaciones, y podrían aplicarse sus resultados en campos muy variados, desde el apoyo al autocuidado de las personas hasta la optimización de la inversión económica en recursos sanitarios.

Este proyecto nace con la intención de ser capaz de procesar las peticiones que se realizan a un servidor sanitario y obtener información de ellas. Además, busca servir de punto de partida para que otras personas puedan ahondar aún más en lo que las herramientas de procesado de datos pueden ofrecernos.

Por ello, los objetivos de este trabajo son los siguientes:

- Poder utilizar un servicio de procesamiento de flujo como intermediario de peticiones realizadas a un servidor sanitario.
- Utilización de sistemas de alertas en el servicio de procesamiento de flujo que notifiquen mediante email.
- Integración del servicio de procesamiento de flujo con una cola de mensajes.

1.2 Estructura del documento

El documento posee una estructura sencilla:

- Este primer apartado que introduce el proyecto y presenta la estructura del documento junto a un pequeño glosario de términos.
- En segundo lugar, se exponen los diferentes softwares y tecnologías que se han usado en este proyecto con una breve explicación de estos, así como su estado actual.
- Posteriormente, se presentan diferentes escenarios de prueba explicando en cada escenario los nuevos elementos introducidos y las diferentes formas en las que estos elementos pueden introducirse. También se presenta el escenario final, incluyendo las diversas pruebas realizadas para demostrar el funcionamiento.
- Por último, un apartado de conclusiones donde se incluyen unas posibles líneas de continuación de cara a mejorar lo realizado en este trabajo.
- Como anexos se añadirán el código desarrollado y un breve manual de instalación.

1.3 Glosario

En este apartado se explica el significado de algunos términos, en el contexto del servicio de procesado de flujo, que se usarán en el resto del trabajo:

Stream Processor [6]: Es el servicio de procesado de flujo con el que se trabajará en el proyecto. Para conocer más sobre él, se recomienda la lectura del punto 2.2.1 en el que se habla de él en profundidad.

Evento: Mínima unidad de información con la que se trabaja. Está formado por uno o varios pares nombre:valor. Un ejemplo de evento seria nombre:Pablo,edad:25.

Stream o flujo: Conjunto de eventos que comparten definición. Por ejemplo, un flujo llamado persona que contenga un atributo de tipo string (cadena de caracteres) llamado nombre y un atributo de tipo int (entero) llamado edad.

Source o Fuente: Interfaz que permite recibir información desde fuera del Stream Processor. Debe ser asignada a un flujo para poder procesar esta información. Por ejemplo, un mensaje de una cola MQTT, una petición REST, ... etc.

Sink o Sumidero: Interfaz que permite enviar información fuera del Stream Processor. Debe ser asignada a un flujo para que se reciba la información a publicar. Por ejemplo, un mensaje de una cola MQTT, una petición REST, ... etc.

Query o consulta: Es la forma de realizar el procesado de la información de un flujo. Con una estructura similar a una sentencia SQL, permite aplicar a los diferentes eventos del flujo que se indiquen filtros, unir varios eventos de distintos flujos, realizar operaciones predefinidas e incluso ejecutar un script. Son ejecutadas al recibirse un evento de un flujo sobre el que se realice la consulta.

Shiddi [7]: Núcleo de WSO2 que realiza el procesado de los flujos de las diferentes consultas. Para conocer más sobre él, se recomienda la lectura del punto 2.2.2 en el que se habla de él en profundidad.

Ventanas: El tamaño de una ventana en el Stream Processor determina los eventos que se tienen en cuenta para las diferentes operaciones, en otras palabras, los eventos que se guardan en memoria para las diferentes operaciones. Por ejemplo, si una fuente recibe tres eventos, por el mismo flujo, con valores ‘5’, ‘10’ y ‘15’ se realizará la suma siendo la ventana de tamaño 3, el resultado sería de 30. Si posteriormente se recibe un flujo con valor 40, el resultado sería de 65.

Se adjunta una imagen donde se muestran las relaciones de los diferentes términos explicados anteriormente:

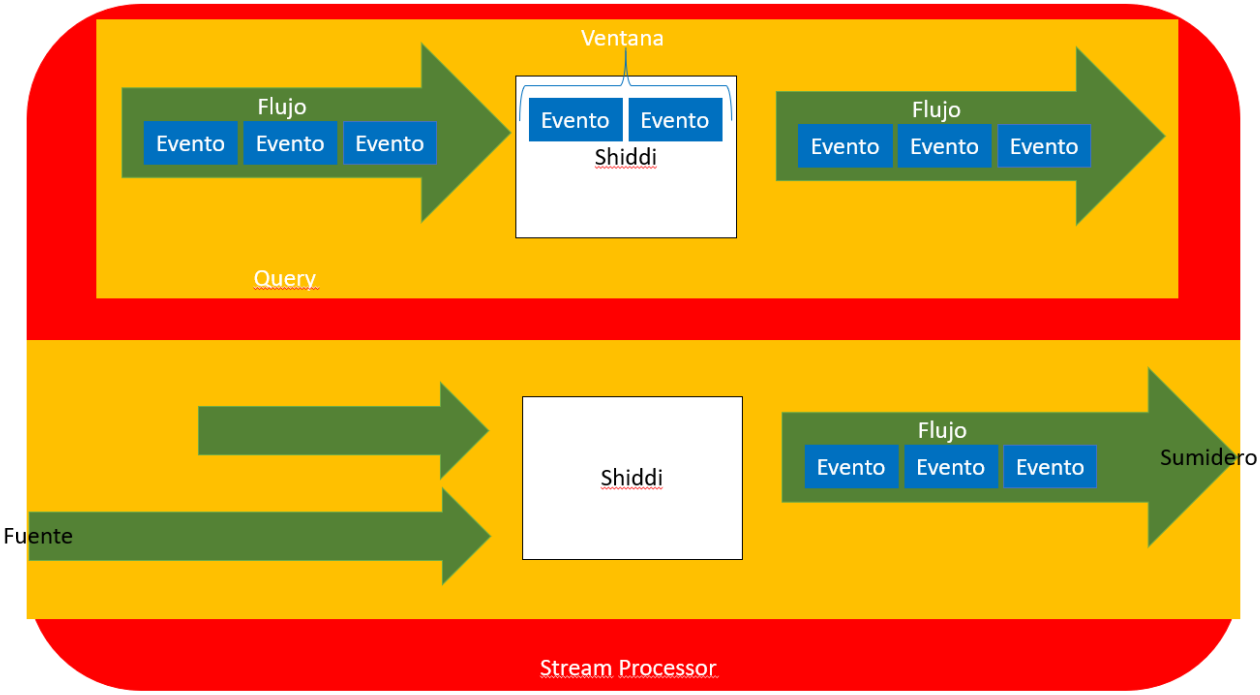


Ilustración 1: Diagrama de conceptos fundamentales del Stream Processor

2 ESTADO DE LA TÉCNICA

En el desarrollo de este proyecto se ha trabajado con diferentes tecnologías y herramientas que actualmente están en auge y cuyo papel en los próximos años puede ser muy importante. En este capítulo se exponen estas herramientas y se da una visión general de su estado actual.

2.1 Big Data

Según la definición de Gartner de 2011 [8], que aún se considera de referencia [9], cuando se habla de Big data hablamos de un conjunto de datos que se presentan con una variedad alta, volúmenes crecientes y velocidad superior. Es lo que se conoce como las 3 'V's. Con el paso del tiempo se han ido incluyendo algunas características más a esta definición. Hoy en día para nombrar ciertos datos como Big Data debemos añadir, al menos, la veracidad y el valor de los datos [10].

El objetivo de las diferentes tecnologías que trabajan con Big Data es analizar estos datos para obtener información. Existen multitud de ejemplos de la utilidad que el procesado de una gran cantidad de datos puede producir; desde determinar fallos hasta descubrir en qué zona y mediante qué plataforma realizar una campaña de publicidad.

Dentro del análisis Big Data existen dos paradigmas muy diferenciados en función a la forma de recibir los datos:

- Batch processing o análisis por lotes: Trabaja con volúmenes muy altos de datos, pero con un ratio bajo de llegada de nuevos bloques de datos. Para realizar este tipo de análisis se suelen utilizar algoritmos tipo Map Reduce y softwares como Apache Hadoop o Apache Sparks.
- Stream processing o análisis de flujo: Trabaja en tiempo real con volúmenes pequeños, pero con la llegada constante de nueva información. Gracias a la llegada de la tecnología IoT está actualmente en auge. Algunos softwares famosos que realizan este tipo de procesado son Apache Storm, Apache Samza o WSO2 Stream Processor. Este es el paradigma que vamos a tratar en este trabajo con ayuda del Stream Processor.

Últimamente están surgiendo escenarios intermedios que se conocen como Microbatch, aunque estos no están tan optimizados actualmente y la metodología para procesar los datos recibidos en este formato está avanzando poco a poco.

2.2 WSO2

WSO2 [11] es una compañía que desarrolla aplicaciones de software abierto (Free source) enfocadas en proveer una arquitectura orientada a servicios (SOA). En la última década es una de las empresas más importantes del sector y su software es usado por multitud de empresas multinacionales como eBay o Motorola.

En los dos últimos años WSO2 ha realizado diversos cambios de cara a simplificar y facilitar la implantación de sus diferentes herramientas pasando de más de 10 productos a 4 productos que engloban gran parte de las funciones que realizaban anteriormente el resto y han eliminado, al menos temporalmente, alguna funcionalidad que no cumplía con los estándares de calidad que si presentan en el resto de las funciones. Actualmente los productos con los que trabaja son:

WSO2 ENTERPRISE INTEGRATOR[1]	WSO2 API MANAGEMENT
Proporciona una integración rápida e interactiva de cualquier tipo de aplicación, dato o sistema. Maneja más de 6 trillones de transacciones por año. Componentes: Data Integration Workflows ESB Integration designer Message bróker	Diseño, creación, reutilización, análisis, manejo e implementación de API Procesa 200k APIs conectando a 20k organizaciones Componentes Analytics Publisher Designer/studio Storefront/marketplace Key Manager Traffic Manager Gateway/Microgateway
WSO2 IDENTITY SERVER	**WSO2 STREAM PROCESSOR**
Identificación, autorización y autorizaciones federadas para asegurar la integración Ayuda a manejar más de 50M de identidades globalmente.	Análisis e integración de datos en tiempo real. Maneja más de 100k eventos por segundo. Componentes Siddhi Business rules manager Stream Processor runtime Dashboard portal Development environment

Tabla 1: Componentes WSO2

Además de esto, también se encargan de realizar diferentes cursos o Webinars donde enseñan a montar escenarios complejos. Por ejemplo, en la propia web se puede consultar un curso para realizar el montaje de un escenario sanitario completo con los cuatro componentes anteriormente comentados:

[1] El uso de estos colores es propio de la web de WSO2, siempre que la página tenga uno de estos colores estará hablando de estos componentes o de una sus partes.

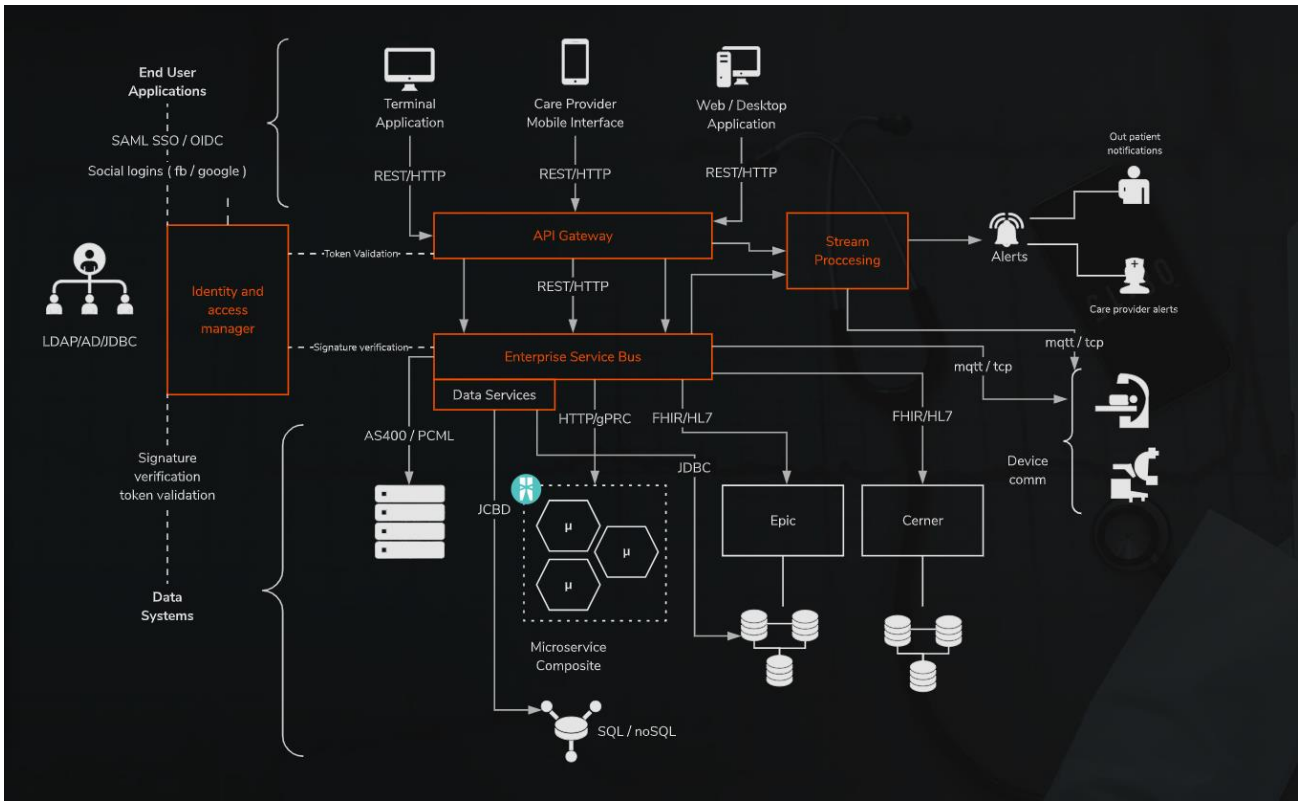


Ilustración 2: Ejemplo escenario sanitario

2.2.1 WSO2 Stream Processor

Como se ha indicado en la introducción, el componente de WSO2 en el que se centra este trabajo es el WSO2 Stream Processor [6].

WSO2 Stream Processor es una plataforma de procesamiento de flujos. Se caracteriza por ser ligera y comprender consultas con formato SQL capaces de capturar, analizar, procesar y actuar sobre eventos en tiempo real. Esto facilita la integración de datos en tiempo real y su análisis.

WSO2 considera que hoy en día lo más importante para una empresa es reaccionar rápidamente a los cambios externos (tendencias de mercado, comportamiento de clientes, patrones ambientales) por lo tanto es fundamental que el procesado de estos datos se produzca rápidamente y se puedan tomar decisiones en tiempo real por ejemplo generando diferentes tipos de alertas. Esta era una conclusión a la que muchas empresas habían llegado y sin embargo no conseguían que las aplicaciones de procesamiento de flujo, escritos en lenguajes de programación de propósito general, dieran los resultados esperados. Esto se debía principalmente a dos problemas:

1. Cambios continuos en las empresas provocaban demasiados cambios en la lógica de procesado con sus revisiones de código y su retraso correspondiente.
2. La mayoría de los sistemas necesitaban al menos 5 o 6 nodos de procesamiento, lo cual podía implicar un proceso complejo y difícil de mantener.

WSO2 buscó la forma de simplificar esto, para ello creó un sistema mediante el cual los desarrolladores solo escribieran scripts sencillos y los comerciales podrían realizar sus ajustes, es decir, para pequeños cambios como por ejemplo modificar el valor al que se producirá una alerta no es necesario que intervengan los desarrolladores, optimizando así la velocidad de cambio. Para ello creó el Stream Processor que recopila eventos con múltiples formatos y mediante sentencias con formato SQL, detecta patrones, puede hacer predicciones, genera avisos y permite visualizarlos en tiempo real. Además, consiguió reducir la complejidad al necesitar solo de dos nodos para ejecutarse alcanzando alta disponibilidad.

WSO2 Stream Processor simplifica las aplicaciones de procesamiento de stream en tres puntos:

1. El desarrollo es más rápido. Gracias principalmente a tener herramientas especializadas, al uso de Siddhi [7] como lenguaje de consultas SQL y al complejo motor de procesamiento de eventos que se encuentra en el núcleo del Stream Processor (SP) . También proporciona:
    - Facilidad de uso
    - Un potente conjunto de herramientas, como filtrado o ventanas.
    - Más de 50 extensiones para casos de uso concretos
    - Soporte para el consumo de eventos de diferentes fuentes entre las que destacan HTTP, e-mail, MQTT, TCP, JMS y Kafka.
    - Compatible con XML, JSON, texto plano, binario y key-value.
    - Integración con sistemas de almacenamiento habituales como RDBMS, Apache Solr, MongoDB, Apache Hbase y Hazelcast.
2. Más fácil de implementar. Al reducir el número de nodos mínimos se simplifica el proceso de despliegue. Sin embargo, no afecta al rendimiento ya que puede manejar fácilmente más de 100k eventos por segundo. En caso de que se requiera un procesado mayor a 100k eventos por segundo, se recomienda usar Apache Kafka (una cola de mensajes distribuida) como intermediario entre las diferentes fuentes y el Stream Processor, ya que su conexión es sencilla y permite reducir la frecuencia de eventos recibidos.
3. Más fácil de modificar. Uno de los mayores problemas que tenían los procesadores anteriores es el lento proceso de cambio ya que los usuarios debían contactar con los desarrolladores para cualquier cambio que se quisiera realizar. WSO2 presenta una interfaz gráfica desde la cual, sin conocimientos de programación, se pueden realizar modificaciones simples como por ejemplo modificar los rangos de valores a los cuales se genera una alerta. Además de esto, presenta una vista completa de lo analizado a través de un panel personalizable que permite componer las vistas tal y como el usuario desee. Está equipado con un generador de widgets que permite construir gráficos personalizados.

2.2.1.1 Scripts WSO2 Stream Processor

El Stream Processor tiene diferentes scripts que permiten arrancar diferentes modos de funcionamiento:

- Editor: Este script se usará en los apartados 3.1 y 3.2. Es el IDE del Stream Processor, al cual se accede desde una interfaz web y que permite el desarrollo del código, la ejecución de estos y la simulación de eventos. No obstante, no permite la ejecución de varios archivos a la vez ni trabajar en modo clúster con otros equipos, por lo que el procesado queda limitado a la capacidad del equipo en el que se ejecuta.
- Worker: Este script se usará en los apartados 3.3 y 3.4. Es el modo de funcionamiento principal del Stream Processor, permite desplegar a la vez diferentes archivos, cada uno realizando sus propios análisis de manera independiente y con ayuda del manager pueden trabajar en modo clúster.
- Manager: Este script permite la coordinación de diferentes Stream Processor arrancados en modo worker para que trabajen como un clúster, sumando así su capacidad de procesamiento.
- Dashboard: Este script suele estar presente en todos los componentes de WSO2. Permite desde una interfaz web contabilizar y mostrar gráficas de las peticiones recibidas en una determinada ip. Permite la creación de dashboards personalizados con diferentes widgets, que muestren el tipo de gráfica y con la información que se desee mostrar. Uno de los dashboard que por defecto están instalados nos muestra las descargas y los ingresos de WSO2 por cada país.

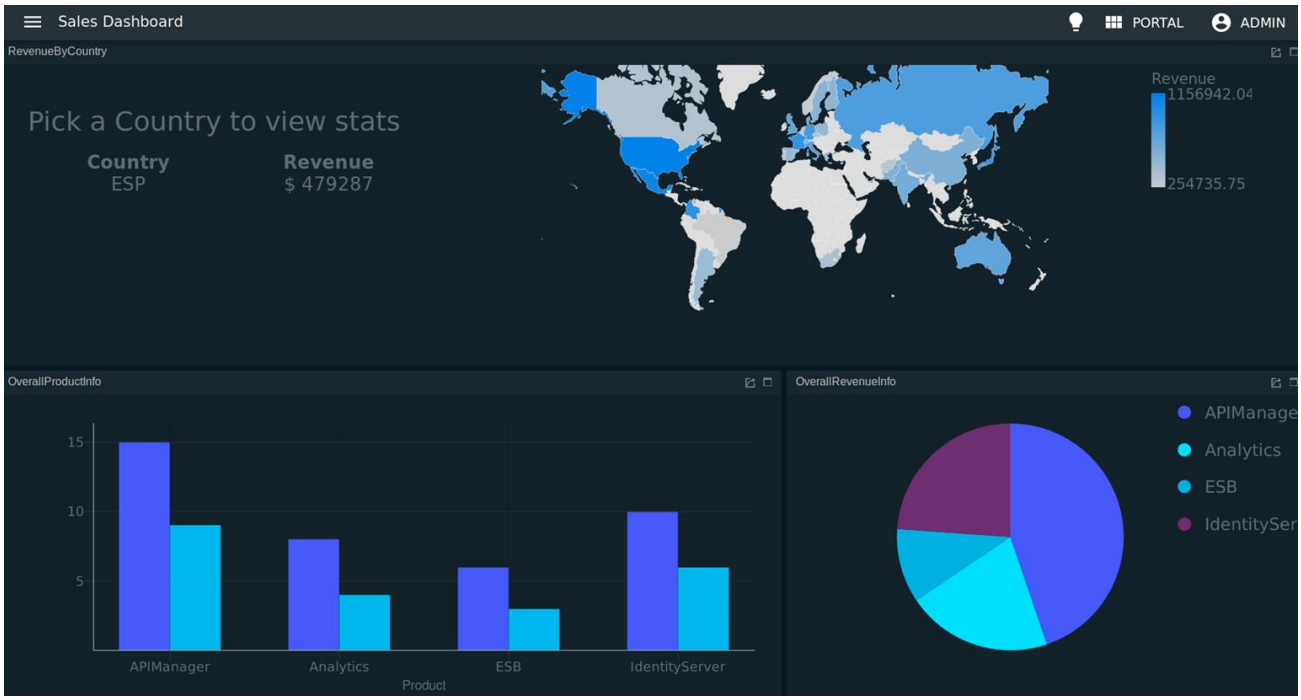


Ilustración 3: Dashboard

2.2.2 Shiddi

Siddhi [7] es un motor de procesamiento y transmisión de eventos complejos que entiende las consultas con formato SQL. Fue desarrollado por WSO2 y presentado como software libre por lo que se puede acceder a él gratuitamente. Puede capturar eventos generados por diversas fuentes, procesarlos, realizar conclusiones complejas y publicar resultados en varios formatos en tiempo real. Forma parte del núcleo del Stream Processor por lo que el lenguaje que se utiliza en el Stream Processor es el que entiende Shiddi.

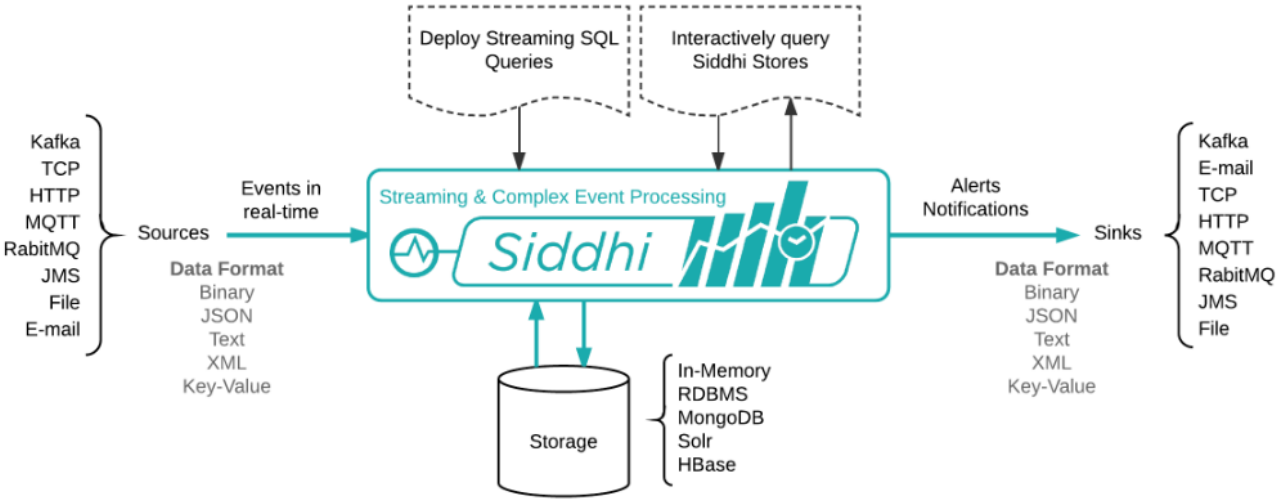


Ilustración 4: Siddhi

Siddhi está preparado para soportar los siguientes casos de uso:

Integración de flujo	Análisis de flujo	Alertas y Notificaciones	Decisiones adaptativas
Recibe eventos de varias fuentes (Kafka, JMS, HTTP, CDC, etc.).	**Puede trabajar usando ventanas de tiempo, ventanas de número de eventos y sesiones.**	**Genera alertas basadas en umbrales.**	**Reglas estáticas de procesamiento.**
Trabaja con múltiples formatos de eventos (JSON, XML, texto plano, etc.)	**Tiene una precisión que permite ir de segundos a años.**	**Correlaciona datos y detecta la falta de eventos y eventos erróneos.**	**Reglas adaptativas de procesamiento.**
Realiza limpieza y preprocesado de datos.	**Detecta tendencias (subidas, bajadas, cambios, ...)**	**Detecta eventos temporales.**	**Utiliza llamada a procedimiento remoto síncrona.**
Puede unir múltiples flujos.	**Realiza predicción en tiempo real con modelos machine learning (PMML, Tensorflow)**	**Detecta la no ocurrencia de eventos.**	**Procesado de consultas a tablas, ventanas y uniones de flujos.**
Puede enviar flujos a base de datos (RDBMS, Cassandra, Hbase, Redis, etc) y a otros servicios.	**Puede utilizar modelos de machine learning online.**	**Publica información a diferentes sumideros (Email, Sistemas de mensajería, servicios y base de datos).**	**Toma de decisión estática y basada en modelos de machine learning local.**

Tabla 2: Casos de Uso Siddhi

Siddhi sufre actualizaciones frecuentemente. Durante la realización de este proyecto ha recibido actualizaciones y algunas han facilitado el desarrollo de éste. La frecuencia de estas actualizaciones se debe a que aún están a mitad de camino del Roadmap que tienen planeado:

- [x] Soporta Kafka
- [x] Soporta NATS
- [x] Distribución de Siddhi en producción.
- [x] Añadir herramientas a Siddhi (Editor)
- [x] Siddhi Kubernetes CRD
- [x] Persistencia incremental periódica del estado
- [ ] Soporta Prometheus para recolección de métricas
- [ ] Soporta despliegues de alta disponibilidad con NATS vía Kubernetes CRD
- [ ] Soporta despliegue distribuido con NATS vía Kubernetes CRD

2.3 REST

REST [12] (o transferencia de estado representacional) es un estilo de arquitectura software para sistemas hipermedias distribuidos, como la WWW. Fue diseñado por Roy Fielding, uno de los principales autores del protocolo HTTP, en el año 2000. Aunque REST no es un estándar, existen multitud de estándares en los que debemos apoyarnos si queremos realizar un diseño REST [13].

Roy Fielding afirmó que la web ha podido escalar eficazmente gracias a una serie de diseños fundamentales que incorpora en su arquitectura:

- Protocolo cliente/servidor sin estado: Cada mensaje contiene toda la información para comprender la petición.
- Conjunto de operaciones bien definidas: las más importantes son POST, GET, PUT y DELETE. Similar a operaciones CRUD en base de datos.
- Sintaxis universal para identificar un recurso: Cada recurso es direccionable únicamente a través de su URL.
- Uso de hipermedios: Tanto para información como para transacciones de estado, haciendo posible navegar entre recursos siguiendo enlaces.

No obstante, en la actualidad se usa el término REST en el sentido más amplio para describir cualquier interfaz que utilice HTTP para obtener datos o indicar la ejecución de operaciones, en cualquier formato sin el uso de otros protocolos como SOAP.

2.4 FHIR

FHIR [14] [15] (Fast Healthcare Interoperability Resources) es el último estándar desarrollado y promovido por la organización internacional HL7, responsable de algunos de los protocolos de comunicación que se utilizan hoy en día en el ámbito sanitario.

FHIR está diseñado para la web, sigue el paradigma REST y utiliza XML y JSON para la representación de los recursos manejados. FHIR personaliza los recursos REST con conceptos del mundo sanitario: paciente, médico, enfermedad, observación, etc., y le proporciona una serie de características extra:

- Un pequeño conjunto de propiedades principales que la mayoría de los sistemas actuales soporta.
- Un mecanismo de extensión que permite añadir nuevas propiedades.
- Un componente que permite una visión legible de los datos almacenados en el recurso

El estándar FHIR puede suponer un gran cambio en el futuro del mundo sanitario, ya que está diseñado desde el punto de vista de las necesidades de las implantaciones y con la idea de que sea tan sencillo como sea posible. Es fácil de aprender, desarrollar e implementar y, lo que es más importante, puede convivir e integrarse correctamente con sistemas anteriores.

2.5 MQTT

MQTT [16] [17] [18] es un protocolo de mensajería basado en la arquitectura publicador/subscriptor que desarrollaron Andy Stanford-Clark y Arlen Nipper en 1999. A pesar de haber sido usado desde entonces ha sido con la llegada de las aplicaciones móviles y aún más con la llegada de la tecnología IoT que su uso ha aumentado exponencialmente.

El funcionamiento de MQTT es bastante sencillo:

- Un topic es la etiqueta que se usa para clasificar los mensajes.
- Los subscriptores pueden enviar un mensaje de suscripción a un bróker indicando el topic (y la QoS) al que quieren subscribirse. También podrían darse de baja si hubieran perdido interés en este topic.
- Los publicadores generan mensajes y lo envían al bróker indicándole un topic determinado.
- El bróker reenvía todos los mensajes recibidos a los subscriptores cuyo topic coincida con el mensaje recibido.

2.5.1 Eclipse Mosquitto

Eclipse Mosquitto [19] es un bróker open source que implementa diferentes versiones de MQTT. Es bastante ligero por lo que es adecuado para todos los dispositivos. Proporciona una biblioteca de C para implementar clientes MQTT, y los clientes de línea de comandos mosquitto_pub y mosquitto_sub.

Este es el bróker que se ha usado en este proyecto, principalmente por su facilidad de despliegue y su simplicidad. Para consultar el proceso de instalación se recomienda la lectura del Anexo C.1 Mosquitto.

3 TRABAJO REALIZADO

En este apartado se explicará el funcionamiento del Stream Processor, como hacer código Shiddi haciendo uso de un escenario de ejemplo y por último, se explicará el escenario real con las diferentes pruebas realizadas.

3.1 Puesta en marcha del Stream Processor-Editor

Una vez descargado e instalado, cuya explicación con detalle se encuentra en el Anexo C.2 Stream Processor, únicamente debemos ejecutarlo desde un terminal. En este caso ejecutaremos el script del editor:

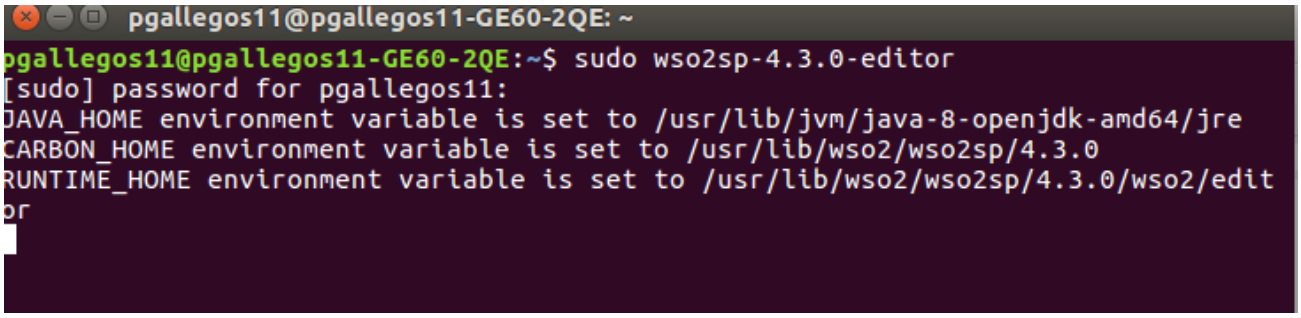


Ilustración 5: Ejecución Stream Processor Editor

En el propio terminal, tras ejecutar el comando anterior, se obtendrá la URL mediante la cual se podrá acceder al editor del Stream Processor

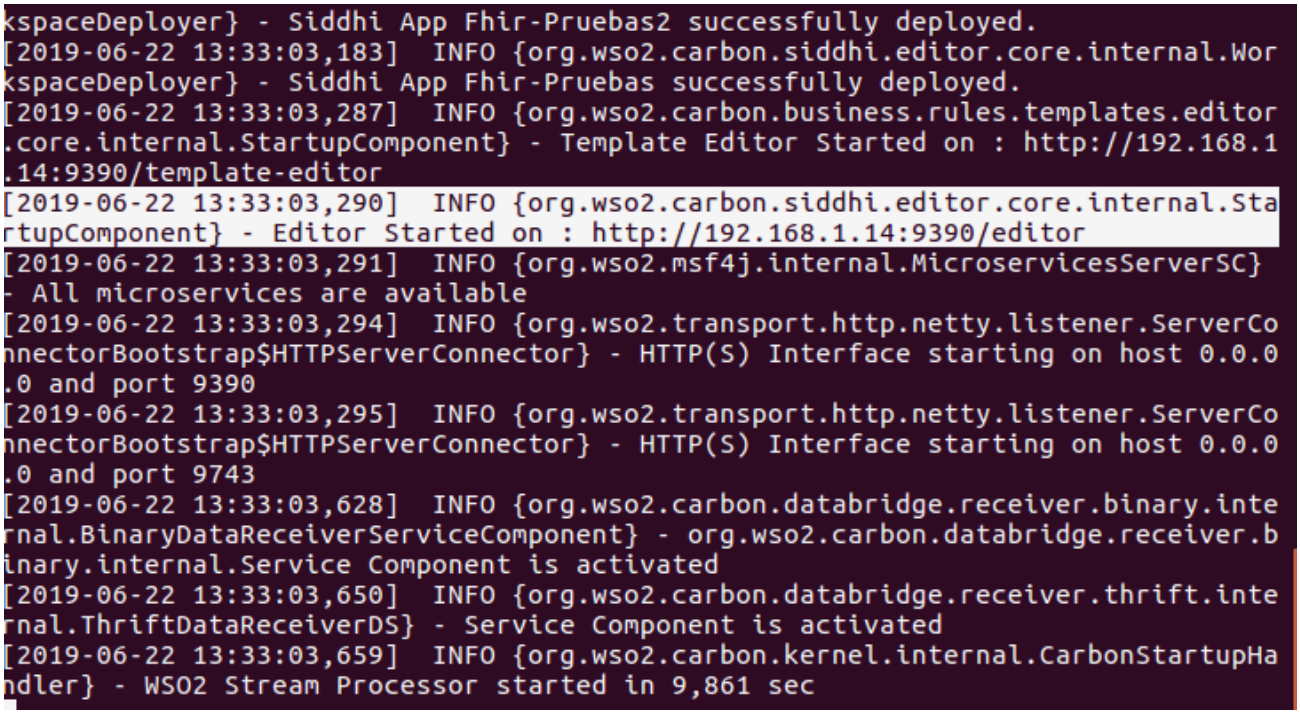


Ilustración 6: URL Editor

En la página inicial nos da la opción de probar algunas muestras de ejemplo, de abrir un archivo y de comenzar a escribir en un archivo nuevo:

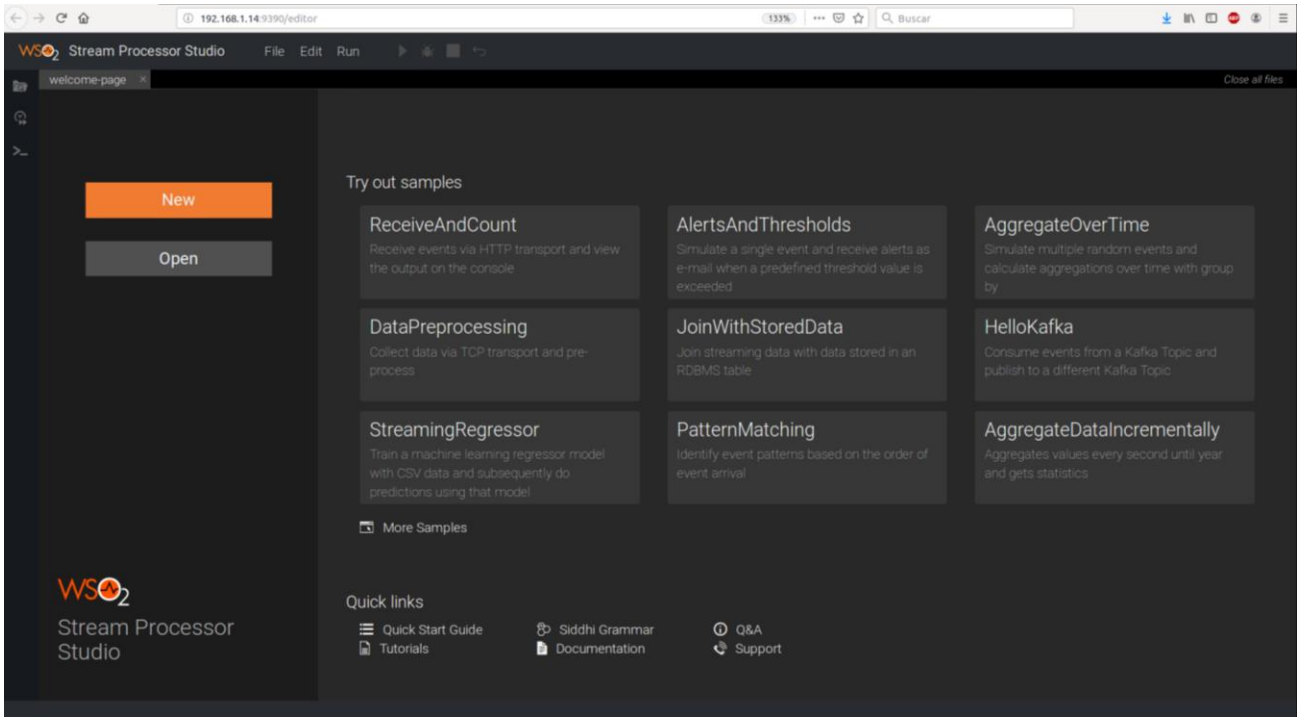


Ilustración 7: Página inicial del editor

Desde la edición de cualquier archivo, podemos abrir tanto el terminal como un simulador de eventos del cual se mostrará su funcionamiento más adelante:

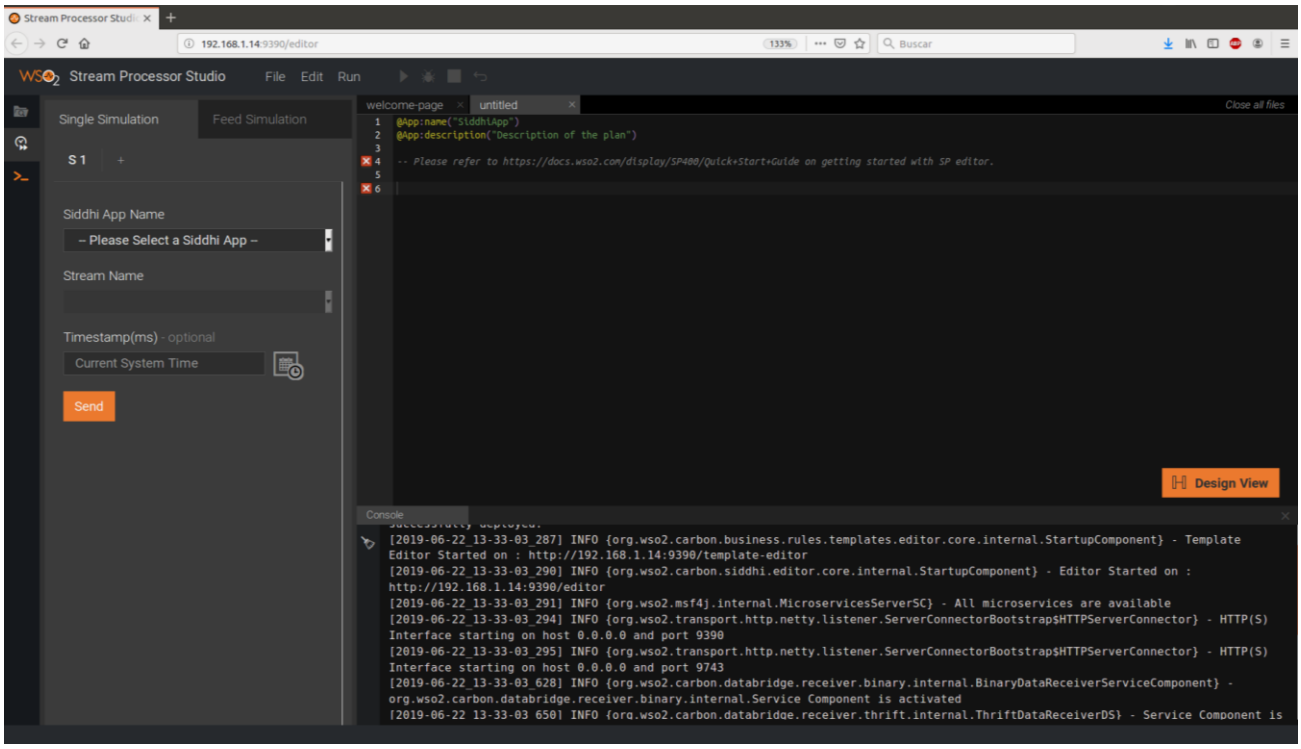


Ilustración 8: Vista de archivo nuevo del editor del Stream Processor

3.2 Manual práctico

El objetivo de este apartado es, mediante el uso de escenarios cada vez más complejos, ir explicando las diferentes funcionalidades que tiene el Stream Processor. Se necesitarían demasiados ejemplos para explicar todas y cada una de las funcionalidades, por lo que se explicarán únicamente aquellas que requieran un tratamiento diferente. Por ejemplo, a pesar de que proporciona diferentes funciones matemáticas, no se explicarán todas ya que su funcionamiento es similar, únicamente se explicarán algunas de ellas. En caso de querer conocer el resto, se recomienda consultar la página oficial de Shiddi [20] o la página de WSO2 de tutorial básico [21].

Para evitar esperas innecesarias y con el fin de solo producir eventos que aporten valor, todas las ventanas están escaladas. Por ejemplo, en el primer escenario se realiza una media del consumo en una tienda tomando como referencia únicamente los últimos tres eventos de compra producidos. En un escenario real, si la tienda quisiera obtener información real, este número sería mucho más grande o simplemente se realizaría mensualmente.

3.2.1 Primer escenario-MediaCompra: stream, sink, filter, windows y función avg

El funcionamiento de este primer escenario es muy sencillo, el Stream Processor recibe eventos que incluyen una cadena de nombre *comprador* y un double de nombre *consumo*. Si el *consumo* es mayor o igual a 16 y menor de 100 imprime por el terminal la media de los tres últimos eventos que cumplen esa condición, en caso de haberse producido menos de 3 eventos realizaría la media de todos los recibidos.

```
1  @App:name('MediaCompra')
2  @App:description('Escenario que calcula la media de los 3 ultimos pedidos')
3
4  -- Flujo basico con dos atributos
5  define stream compra (comprador string,consumo double);
6
7  /*
8  * Flujo con un sumidero o sink asignado de tipo log por lo que se mostrara por el terminal
9  * Todos los mensajes que se muestren comenzaran por el prefijo: Ejemplo media consumo:
10 */
11 @sink(type = 'log', prefix = 'Ejemplo media consumo')
12 define stream mediaConsumo (media double);
13
14 /*
15 * Realiza la media de lo que han consumido las ultimas 3 compras
16 * siempre y cuando el importe sea mayor o igual a 16 y menos de 100
17 */
18 @info(name= 'consultaMedia')
19 from compra[(consumo >= 16 and consumo < 100)]#window.length(3)
20 select avg(consumo) as media
21 insert into mediaConsumo;
```

Ilustración 9: Código Stream Processor del primer escenario- MediaCompra

Como se puede observar, todo código comienza con el **@App:name**, que debe coincidir con el nombre del archivo, y el **@App:description** con una breve descripción de este.

Para montar el primer escenario de pruebas se han añadido dos **flujos (stream).**

El flujo "compra" con dos atributos *comprador* y *consumo* y el flujo "mediaConsumo" con un atributo *media*.

Al flujo "mediaConsumo" se le ha añadido un **sumidero (sink)** de tipo **log** lo que indica que proporcionará

una salida en este caso, al ser de tipo log, por la consola. Al no existir un flujo con una fuente asignada la única forma de recibir evento es a través del simulador de eventos.

En la **consulta (query)** se ha añadido una **ventana (window)** de 3 eventos, un filtro que permite que si el *consumo* es superior a 100 o inferior a 10 no se contabilice el evento y se ha hecho uso de una **función propia** de Shiddi, en este caso **avg,** que calculará el valor medio. En caso de que los atributos del flujo de entrada y salida de la query tengan el mismo nombre se asignará automáticamente. En caso contrario se debe hacer uso de **as** para indicar a que atributo del flujo de salida se debe igualar cada parámetro seleccionado en el **select.**

A modo informativo, para facilitar la lectura del código, se puede añadir la etiqueta **info** para nombrar la consulta y usar **comentarios** monolíneas (--) o multilíneas (/**/).

Se pueden probar los diferentes escenarios únicamente pulsando el botón play y generando eventos con el simulador de eventos que viene incorporado. En la siguiente imagen se muestra con mayor detalle el simulador de eventos:

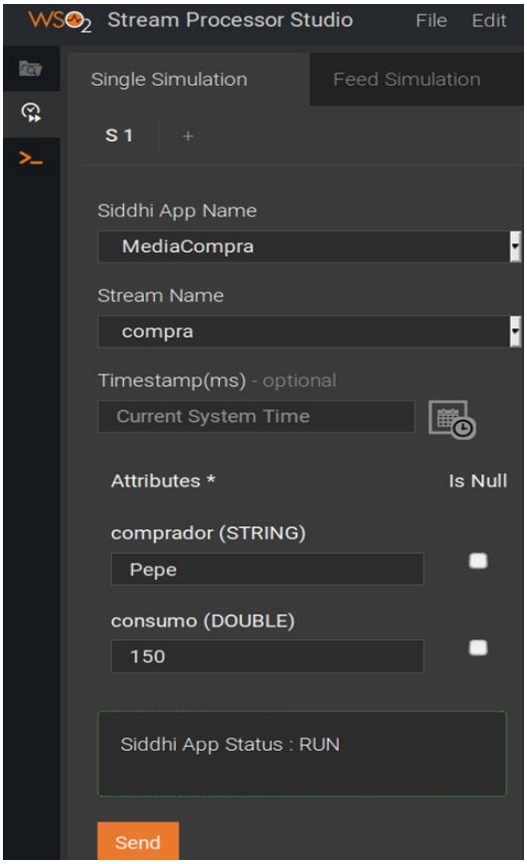


Ilustración 10: Ejemplo de configuración del simulador de eventos

3.2.1.1 Pruebas primer escenario

En este escenario se han realizado las siguientes pruebas:

1. Llegada del primer evento mediante el simulador de eventos al flujo "compra". El contenido es *Comprador:Pepe, Consumo:150*. La llegada de un evento del flujo "compra" activa la realización de la consulta "consultaMedia":
   - "consultaMedia": Comprueba que el valor de *consumo* esté entre 16 y 100, al no estar entre esos valores no provoca ninguna salida.
2. Llegada del segundo evento mediante el simulador de eventos al flujo "compra". El contenido es *Comprador:Alberto, Consumo:50*. La llegada de un evento del flujo "compra" activa la realización de la consulta "consultaMedia":

- "consultaMedia": Comprueba que el valor de *consumo* este entre 16 y 100, al estar entre esos valores calcula la media del consumo de este evento y los dos producidos anteriormente. Dado que no existe ningún otro evento la media es el valor de consumo de este evento. Este valor es incluido en el flujo "mediaConsumo" (timestamp=1563904859980) y mostrado en el terminal como podemos ver en la siguiente imagen:

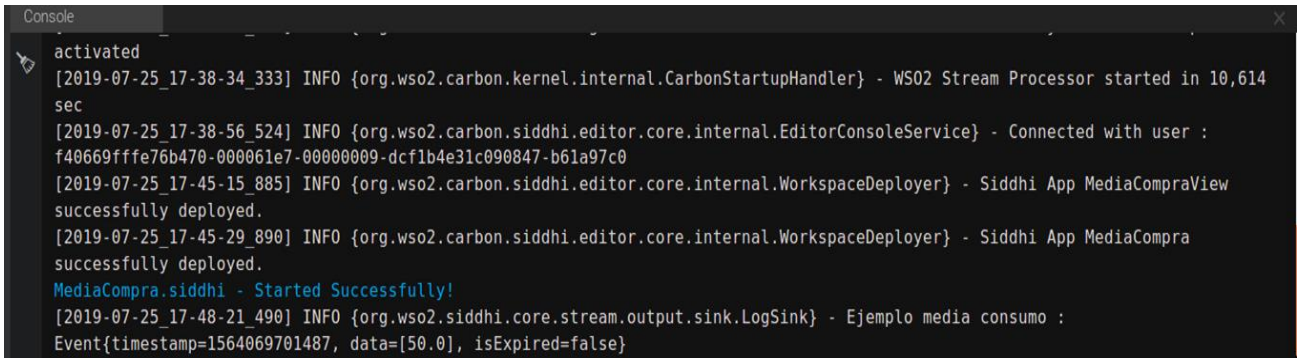


Ilustración 11: Segundo evento del primer escenario-MediaCompra

3. Llegada del tercer evento mediante el simulador de eventos al flujo "compra". El contenido es *Comprador:Luis, Consumo:17*. La llegada de un evento del flujo "compra" activa la realización de la consulta "consultaMedia":
   - "consultaMedia": Comprueba que el valor de *consumo* este entre 16 y 100, al estar entre esos valores calcula la media del *consumo* de este evento y los dos producidos anteriormente. Realiza la media del consumo de este evento y el anterior. Este valor es incluido en el flujo "mediaConsumo" (timestamp=1563904859980) y mostrado en el terminal.
4. Llegada del cuarto evento mediante el simulador de eventos al flujo "compra". El contenido es *Comprador:Marta, Consumo:18*. La llegada de un evento del flujo "compra" activa la realización de la consulta "consultaMedia":
   - "consultaMedia": Comprueba que el valor de *consumo* este entre 16 y 100, al estar entre esos valores calcula la media del *consumo* de este evento y los dos producidos anteriormente. Realiza la media del *consumo* de este evento y los dos eventos anteriores. Este valor es incluido en el flujo "mediaConsumo" (timestamp=1563904908237) y mostrado en el terminal.
5. Llegada del quinto evento mediante el simulador de eventos al flujo "compra". El contenido es *Comprador:Ana, Consumo:25*. La llegada de un evento del flujo "compra" activa la realización de la consulta "consultaMedia":
   - "consultaMedia": Comprueba que el valor de *consumo* este entre 16 y 100, al estar entre esos valores calcula la media del *consumo* de este evento y los dos producidos anteriormente, sin incluir ya el segundo evento recibido. Este valor es incluido en el flujo "mediaConsumo" (timestamp=1563904930614) y mostrado en el terminal.

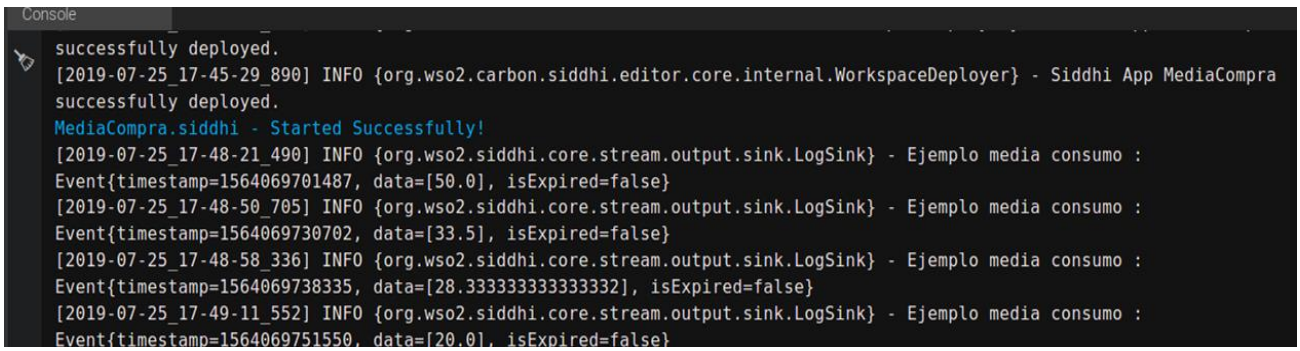


Ilustración 12: Resultado de las pruebas del primer escenario-MediaCompra

3.2.1.2 Design view primer escenario

Existe la posibilidad, como se ha comentado en el apartado 2.2.1, de realizar los programas desde una interfaz gráfica. Se puede cambiar de interfaz gráfica a código y automáticamente los cambios se reflejan en el código y viceversa.

La interfaz gráfica puede resultar interesante en los primeros desarrollos, no obstante, dada su similitud con otros lenguajes de programación se suele trabajar directamente con el código ya que es más rápido y eficiente. El objetivo principal de este interfaz, según la propia página de WSO2, no es que un programador use esa interfaz para realizar un programa, sino más bien que un usuario sin conocimientos de programación realice modificaciones en un código ya existente.

A modo de ejemplo para que el lector pueda tomar sus propias conclusiones y únicamente para este escenario se mostrará también el proceso de creación desde la ventana de desing view que permite generar código o realizar cambios desde una interfaz gráfica.

En el panel lateral tenemos diferentes iconos para arrastrar al panel:

	Source		Sink		Stream
	Table		Window		Trigger
	Aggregation		Function		Projection query
	Filter query		Window query		Function query
	Join query		Pattern query		Sequence query
	Partition				

Tabla 3: Iconos de la vista de diseño

El desarrollo del código anterior dio como resultado este diseño:

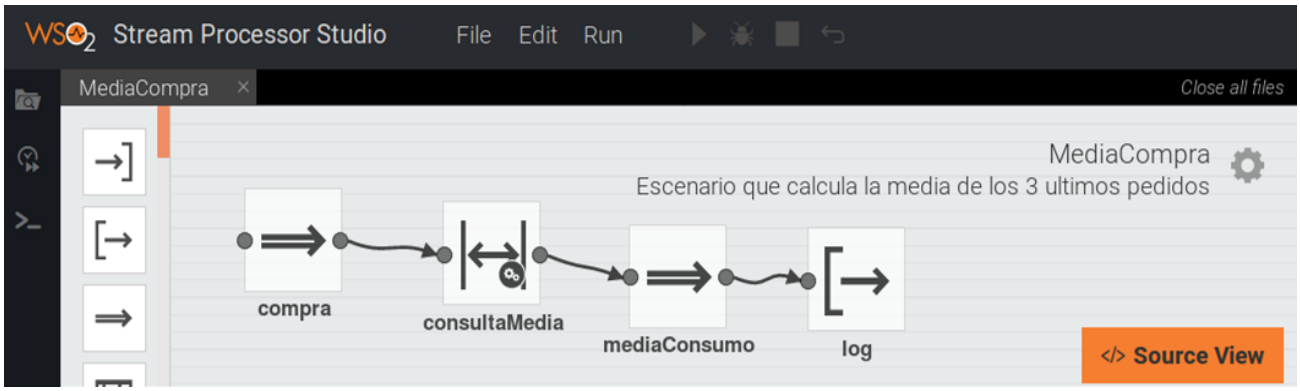


Ilustración 13: Diseño del primer escenario-MediaCompra

Tras abrir un archivo nuevo, se modifican los atributos *Name* y *Description* para lo cual se debe hacer clic en el engranaje que se muestra junto al nombre actual, SiddhiApp:

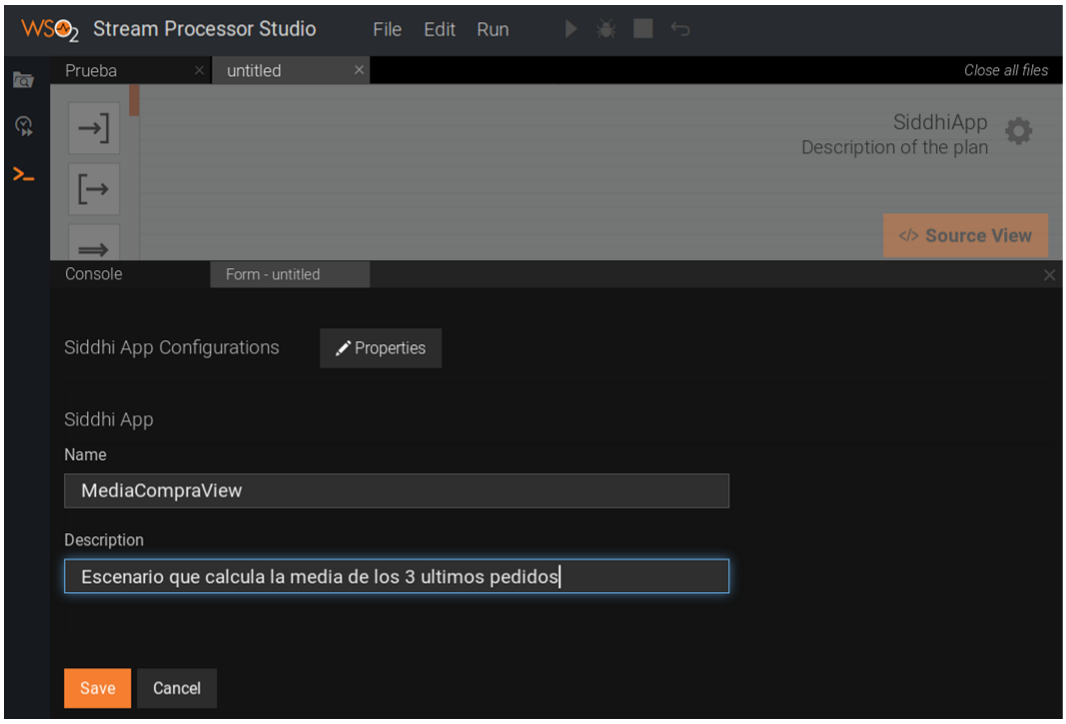


Ilustración 14: Modificando el nombre y la descripción

Tras esto se creará el flujo “compra”, para ello se arrastra su icono al panel en blanco y nos sale una pestaña donde rellenar los atributos que se quiere que tenga este flujo:

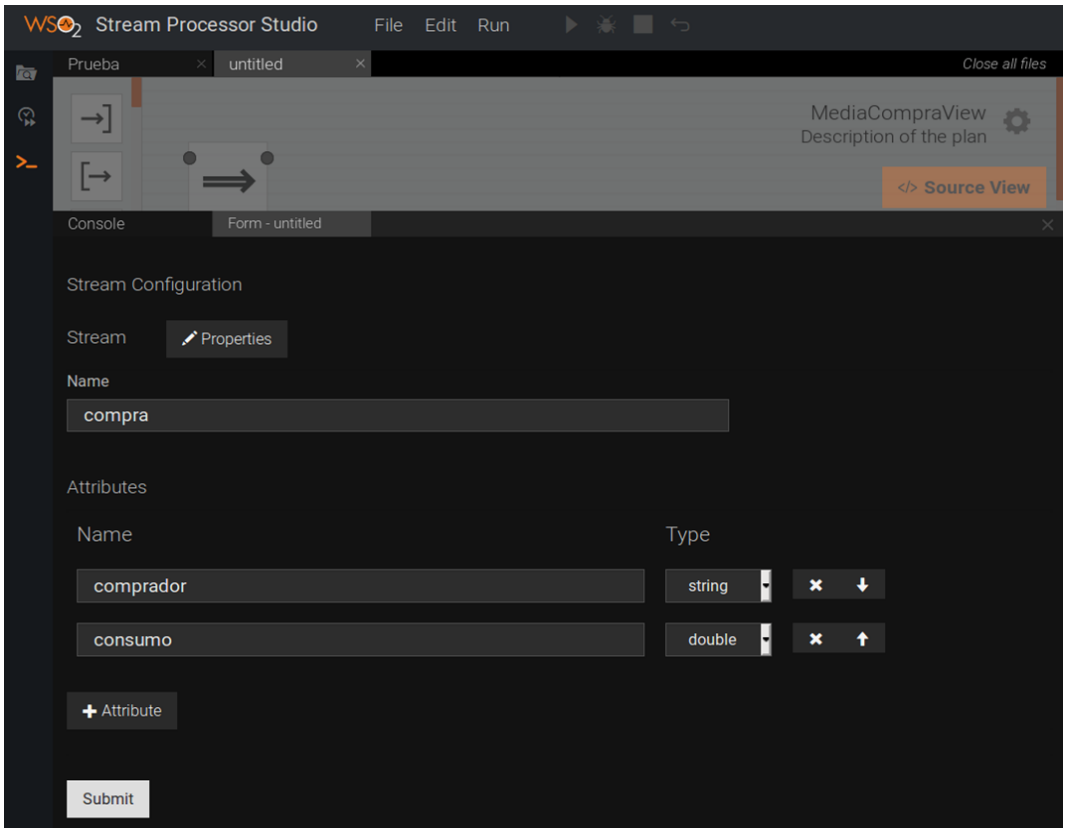


Ilustración 15: Creación flujo persona

Dado que al crear la consulta se debe indicar el flujo de entrada y de salida, se creará primero el flujo de salida:

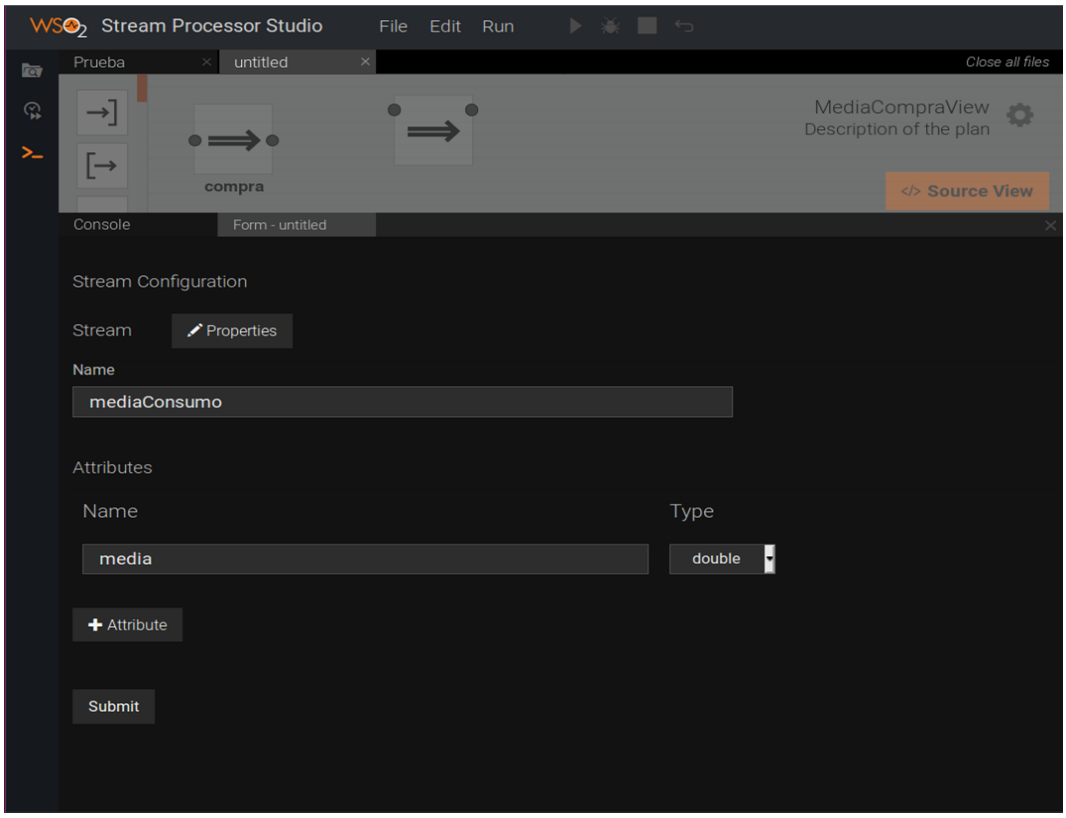
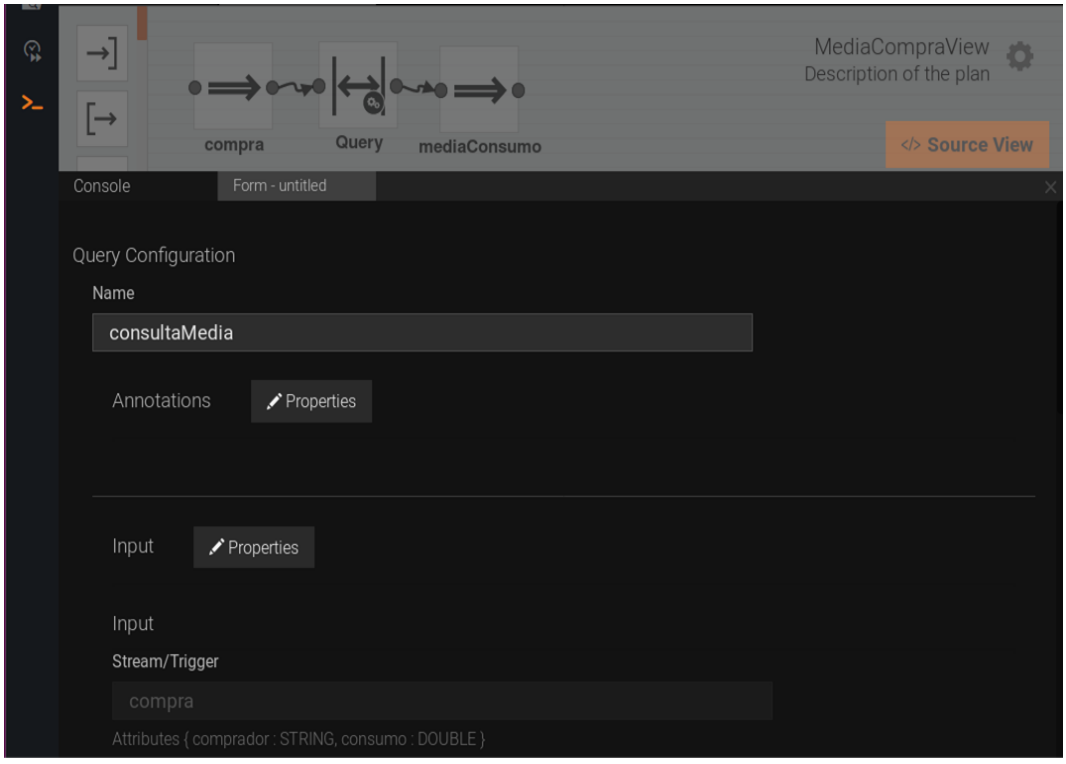


Ilustración 16: Creación flujo mediaEdad

El punto más complicado desde la interfaz web es la creación de una consulta. En este caso se va a crear la consulta de nombre "ConsultaMedia" (que incluye una ventana y un filtro):



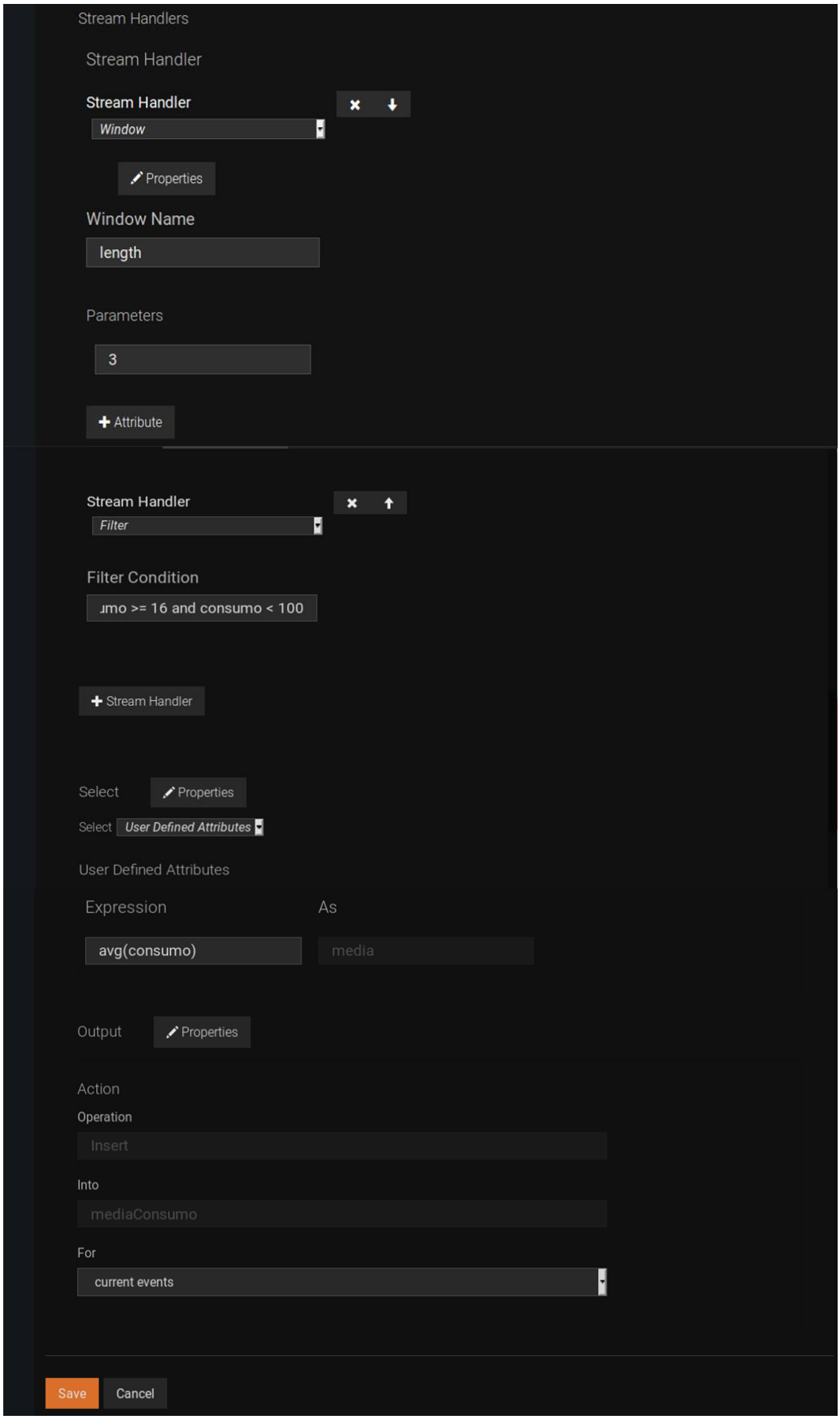


Ilustración 17: Creación ConsultaMedia

Por último, se añade el sink de tipo log al flujo “mediaConsumo” que es el flujo de salida de la query “consultaMedia”:

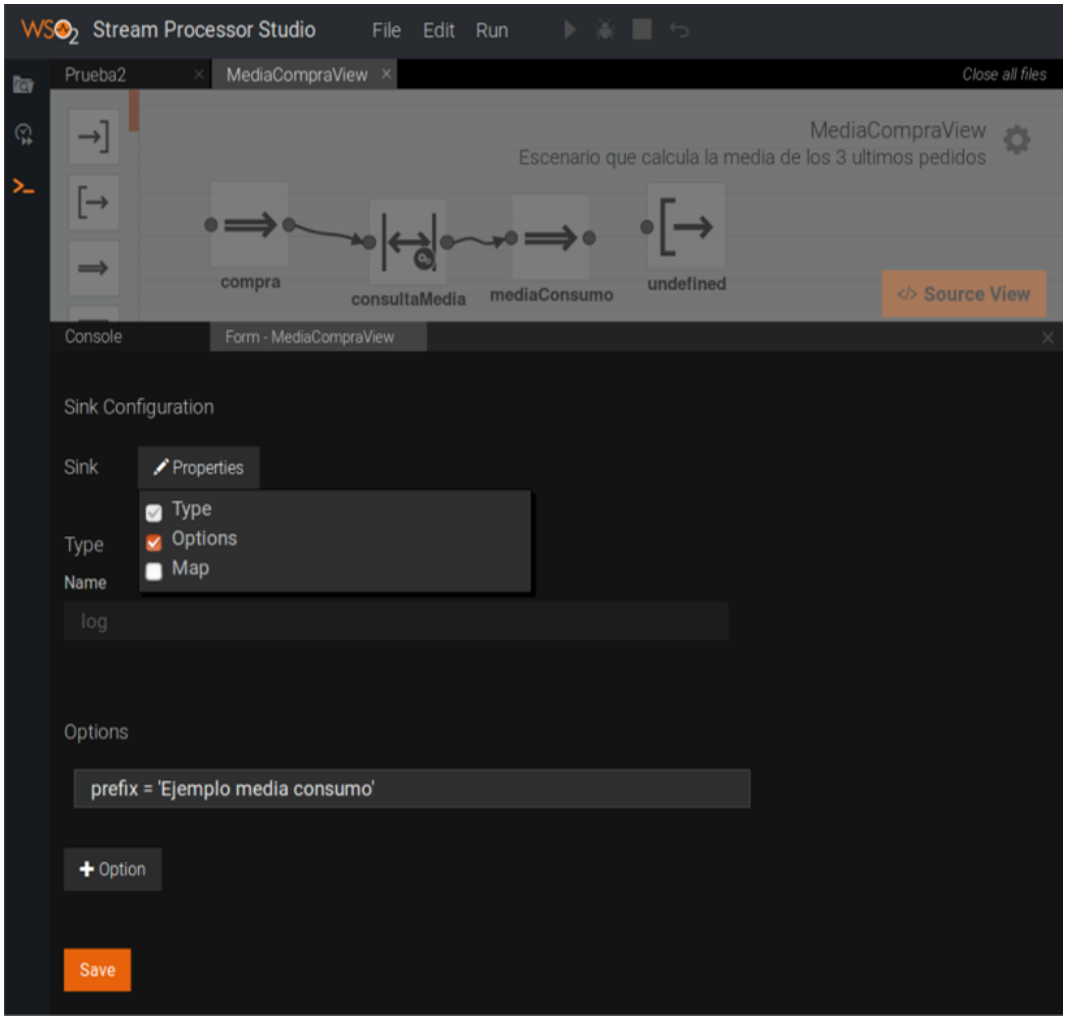


Ilustración 18: Creación de sumidero

El resultado final visualmente es el siguiente:

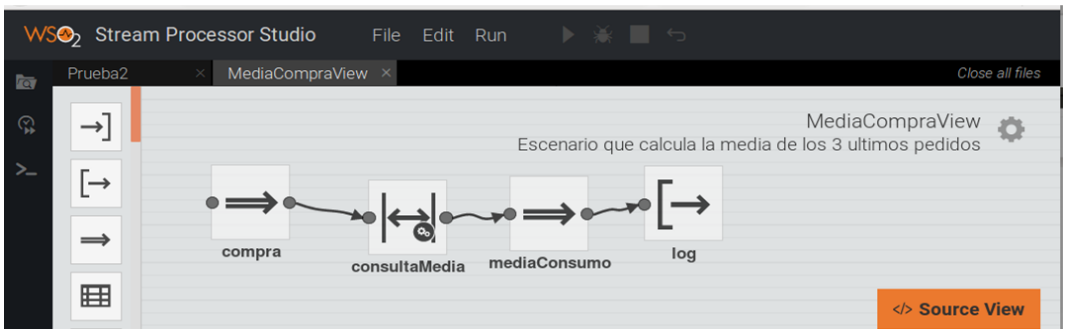


Ilustración 19: Resultado visual del primer escenario-MediaCompra

Se puede cambiar de vista siempre que no exista ningún error y se podrá comprobar que el código generado es igual al del apartado anterior salvo por el espaciado y los comentarios.

Ilustración 20: Código Stream Processor tras desarrollo visual del primer escenario-MediaCompra

3.2.2 Segundo escenario-RegistroCompra: source, group by, having, order by, limit y convert

El funcionamiento de este escenario es el siguiente, recibe eventos que incluyen un string (*comprador*) y un double (*consumo*) y devuelve un evento dando las gracias por la compra. Además, cada dos minutos mostrará un registro con los 3 usuarios que más *consumo* han realizado en estos dos minutos siempre y cuando no sobrepasen cierta cantidad.

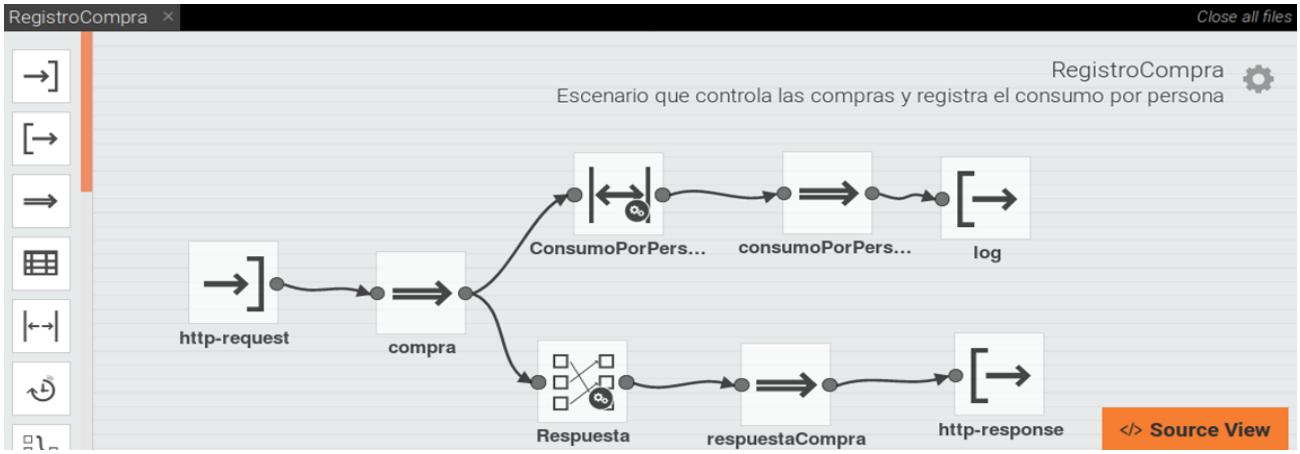


Ilustración 21: Segundo escenario-RegistroCompra vista de diseño

RegistroCompra

```
1  @App:name('RegistroCompra')
2  @App:description('Escenario que controla las compras y registra el consumo por persona')
3  /*
4  * Flujo con una fuente de tipo http-request que recibira las peticiones con body xml a la url
5  * http://localhost:5005/Compra. El xml recibido tendra dos campos comprador y consumo.
6  * Para poder responder se asigna un messageID automatico y el nombre de fuente Recibido
7  */
8  @source(type="http-request", receiver.url='http://localhost:5005/Compra', source.id="Recibido",
9  @map(type='xml',@attributes(messageID='trp:messageID', comprador='comprador', consumo='consumo')))
10 define stream compra (messageID string,comprador string, consumo double);
11 /*
12 * Flujo con un sumidero de tipo http-response que respondera las peticiones con un xml gracias al
13 * messageID y al nombre de fuente Recibido
14 */
15 @sink(type='http-response', source.id='Recibido', message.id='{{messageID}}',
16 headers="'content-type:xml'",@map(type='xml'))
17 define stream respuestaCompra (messageID string,mensaje string);
18 /*
19 * Flujo con un sumidero o sink asignado de tipo log por lo que se mostrara por el terminal
20 * Todos los mensajes que se muestren comenzaran por el prefijo: Consumo total por comprador:
21 */
22 @sink(type = 'log', prefix = 'Consumo total por comprador')
23 define stream consumoPorPersona (consumo double,comprador string, numCompras long);
24 /*
25 * Consulta que cada dos minutos calcula el consumo por persona y el numero de compras
26 * Siempre y cuando el consumo total sea menos de 100 y enviara los 3 con mayor consumo
27 * a consumoPorPersona
28 */
29 @info(name = 'ConsumoPorPersona')
30 from compra#window.timeBatch(2 min)
31 select  sum(consumo) as consumo,comprador as comprador , count() as numCompras
32 group by comprador
33 having consumo < 100
34 order by consumo desc
35 limit 3
36 insert into consumoPorPersona;
37 --Consulta que genera un mensaje agradeciendo la compra por cada evento de compra
38 @info(name='Respuesta')
39 from compra
40 select messageID as messageID, convert("Muchas gracias por su compra",'string') as mensaje
41 insert into respuestaCompra;
```

Ilustración 22: Código Stream Processor del segundo escenario-RegistroCompra

En este caso se ha incluido una **source** que recibirá peticiones HTTP de tipo POST, que son las únicas que puede recibir el Stream Processor, a la URL "http://localhost:5005/Compra" y se le asigna como source.id "Recibido" que se usara luego para poder responder el mensaje.

Esta petición tal y como indica la etiqueta **@map** debe incluir un **xml**, con dos parámetros, una cadena de nombre *comprador* y un decimal de nombre *consumo* que se asignarán a los valores del flujo con el mismo nombre. Además, para poder responder a la petición, se hace uso de ***messageID***. **MessageID** es una etiqueta que el Stream Processor puede añadir automáticamente, si se le indica, a todos los eventos que recibe desde una **source**. Gracias a este identificador, aunque se reciban diferentes peticiones, cada petición estará siempre identificada y se le podrá dar la respuesta que le corresponda. En este caso a todos se responderá con el mismo mensaje, creado mediante el uso de **convert,** que permite la conversión entre diferentes tipos de variable**,** en el que da las gracias por la compra.

Las diferentes etiquetas de un flujo que se usan para configurar si tiene asignada una fuente (**source**) o un sumidero (**sink**) se pueden configurar estáticamente mediante el uso de ' ' o " " indistintamente. También se puede configurar dinámicamente, en función de valores del propio flujo, mediante el uso de {{atributo}} usando de esta manera el contenido del evento para ese atributo.

Internamente calcula el *consumo* por cada persona usando la función **sum** y el número de compras (**count**) realizadas por cada *comprador* **(group by)** y siempre que el *consumo* no supere 100 (**having**) y, cada dos minutos mostrará por el log los tres eventos **(limit 3)** con mayor *consumo* **(order by consumo desc)**.

3.2.2.1 Pruebas segundo escenario

En este escenario se han realizado las siguientes pruebas:

1. Llegada del primer evento del flujo "compra" mediante petición POST realizada a la URL http://localhost:5005/Compra. El contenido es *Comprador:Pablo, Consumo:90*. La llegada de un evento del flujo "compra" activa la realización de dos consultas:
   - "ConsumoPorPersona" suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo "consumoPorPersona" (timestamp=1564071582093) con el contenido *Comprador:Pablo, Consumo:90, numCompras:1*.
   - "Respuesta" se genera un mensaje agradeciendo la compra que se incluye en el flujo "respuestaCompra" que se enviará como respuesta a la petición HTTP recibida anteriormente.
2. Llegada del segundo evento del flujo "compra" mediante petición POST realizada a la URL http://localhost:5005/Compra. El contenido es *Comprador:Pablo, Consumo:15*. La llegada de un evento del flujo "compra" activa la realización de dos consultas:
   - "ConsumoPorPersona" suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto no se inserta en el evento del flujo "consumoPorPersona" ya que el consumo sería superior a 100.
   - "Respuesta" se genera un mensaje agradeciendo la compra que se incluye en el flujo "respuestaCompra" que se enviará como respuesta a la petición HTTP recibida anteriormente.
3. Llegada del tercer evento del flujo "compra" mediante petición POST realizada a la URL http://localhost:5005/Compra. El contenido es *Comprador:Alba, Consumo:50*. La llegada de un evento del flujo "compra" activa la realización de dos consultas:
   - "ConsumoPorPersona" suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo "consumoPorPersona" con el contenido *Comprador:Alba, Consumo:50, numCompras:1*.
   - "Respuesta" se genera un mensaje agradeciendo la compra que se incluye en el flujo "respuestaCompra" que se enviará como respuesta a la petición HTTP recibida anteriormente.

4. Llegada del cuarto evento del flujo "compra" mediante petición POST realizada a la URL http://localhost:5005/Compra. El contenido es *Comprador:Alba, Consumo:40*. La llegada de un evento del flujo "compra" activa la realización de dos consultas:
   - "ConsumoPorPersona" suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo "consumoPorPersona"(timestamp=1564871596619) actualizando el evento anterior con el contenido *Comprador:Alba, Consumo:90, numCompras:2*.
   - "Respuesta" se genera un mensaje agradeciendo la compra que se incluye en el flujo "respuestaCompra" que se enviará como respuesta a la petición HTTP recibida anteriormente.
5. Llegada del quinto evento del flujo "compra" mediante petición POST realizada a la URL http://localhost:5005/Compra. El contenido es *Comprador:Luis, Consumo:30*. La llegada de un evento del flujo "compra" activa la realización de dos consultas:
   - "ConsumoPorPersona" suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo "consumoPorPersona" (timestamp= 1564071600506) con el contenido *Comprador:Luis, Consumo:30, numCompras:1*.
   - "Respuesta" se genera un mensaje agradeciendo la compra que se incluye en el flujo "respuestaCompra" que se enviará como respuesta a la petición HTTP recibida anteriormente.
6. Llegada del sexto evento del flujo "compra" mediante petición POST realizada a la URL http://localhost:5005/Compra. El contenido es *Comprador:Ana, Consumo:20*. La llegada de un evento del flujo "compra" activa la realización de dos consultas:
   - "ConsumoPorPersona" suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo "consumoPorPersona" con el contenido *Comprador:Ana, Consumo:20, numCompras:1*.
   - "Respuesta" se genera un mensaje agradeciendo la compra que se incluye en el flujo "respuestaCompra" que se enviará como respuesta a la petición HTTP recibida anteriormente.

Estos eventos se han enviado mediante el comando CURL como se puede ver a continuación junto a las respuestas de agradecimiento:

```
pgallegos11@pgallegos11-GE60-2QE:~$ curl -X POST   http://localhost:5005/Compra   -H 'content-ty
pe: application/xml'   -d '<event><comprador>Pablo</comprador><consumo>90</consumo></event>'
<events><event><messageID>01386561-d2ec-40e7-89af-c6607a59f2c4</messageID><mensaje>Muchas gracia
s por su compra</mensaje></event></events>pgallegos11@pgallegos11-GE60-2QE:~$
pgallegos11@pgallegos11-GE60-2QE:~$ curl -X POST   http://localhost:5005/Compra   -H 'content-ty
pe: application/xml'   -d '<event><comprador>Pablo</comprador><consumo>15</consumo></event>'
<events><event><messageID>dfa6b637-952c-4c6d-852b-124269b2325e</messageID><mensaje>Muchas gracia
s por su compra</mensaje></event></events>pgallegos11@pgallegos11-GE60-2QE:~$
pgallegos11@pgallegos11-GE60-2QE:~$ curl -X POST   http://localhost:5005/Compra   -H 'content-ty
pe: application/xml'   -d '<event><comprador>Alba</comprador><consumo>50</consumo></event>'
<events><event><messageID>413f5724-64ad-4da6-aa27-80dabb3c9218</messageID><mensaje>Muchas gracia
s por su compra</mensaje></event></events>pgallegos11@pgallegos11-GE60-2QE:~$
pgallegos11@pgallegos11-GE60-2QE:~$ curl -X POST   http://localhost:5005/Compra   -H 'content-ty
pe: application/xml'   -d '<event><comprador>Alba</comprador><consumo>40</consumo></event>'
<events><event><messageID>5d58bc12-dac1-44e7-97d7-2167122face8</messageID><mensaje>Muchas gracia
s por su compra</mensaje></event></events>pgallegos11@pgallegos11-GE60-2QE:~$
pgallegos11@pgallegos11-GE60-2QE:~$ curl -X POST   http://localhost:5005/Compra   -H 'content-ty
pe: application/xml'   -d '<event><comprador>Luis</comprador><consumo>30</consumo></event>'
<events><event><messageID>97353d2b-deab-46ab-8f90-c9adccd1d4eb</messageID><mensaje>Muchas gracia
s por su compra</mensaje></event></events>pgallegos11@pgallegos11-GE60-2QE:~$
pgallegos11@pgallegos11-GE60-2QE:~$ curl -X POST   http://localhost:5005/Compra   -H 'content-ty
pe: application/xml'   -d '<event><comprador>Ana</comprador><consumo>20</consumo></event>'
<events><event><messageID>ce416bc5-08e3-47c7-b6db-b7fb09ce4b63</messageID><mensaje>Muchas gracia
s por su compra</mensaje></event></events>pgallegos11@pgallegos11-GE60-2QE:~$
```

Ilustración 23: Eventos segundo escenario-RegistroCompra

Pasados dos minutos se muestra por el terminal lo siguiente:

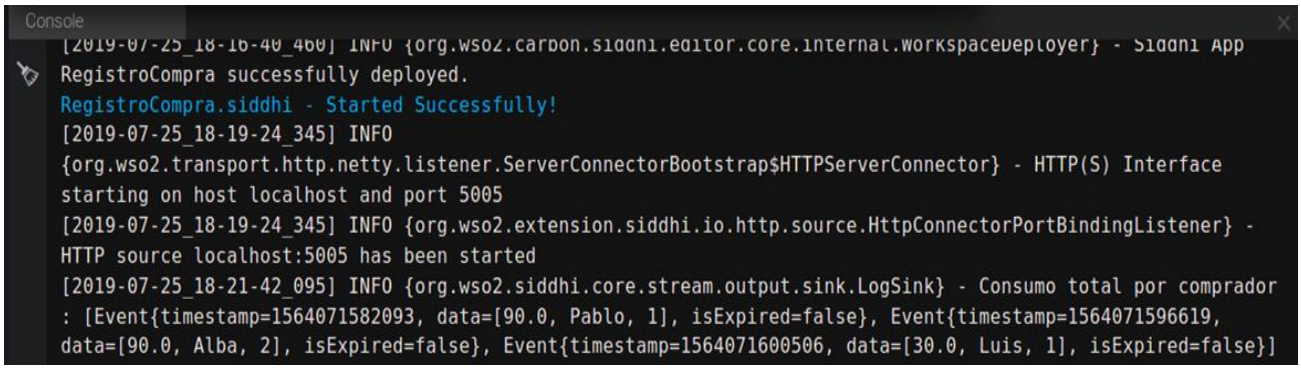


Ilustración 24: Salida terminal segundo escenario-RegistroCompra

Como se puede observar se han mostrado los tres eventos con mayor consumo que no sobrepasan el valor 100: El primer evento de *Pablo*, el segundo evento de *Alba* y el primer evento de *Luis*, dejando fuera el evento de *Ana* ya que el consumo es el más bajo.

3.2.3 Tercer escenario-DescuentosTuTienda: table, join, pattern, script y trigger

En este escenario se reciben desde el simulador eventos de compra similares a los que se recibían en el escenario anterior, y muestra por el log el número de compras y el consumo total. Este escenario tendrá en cuenta el número de compras y en función de su valor se procederá de forma diferente.

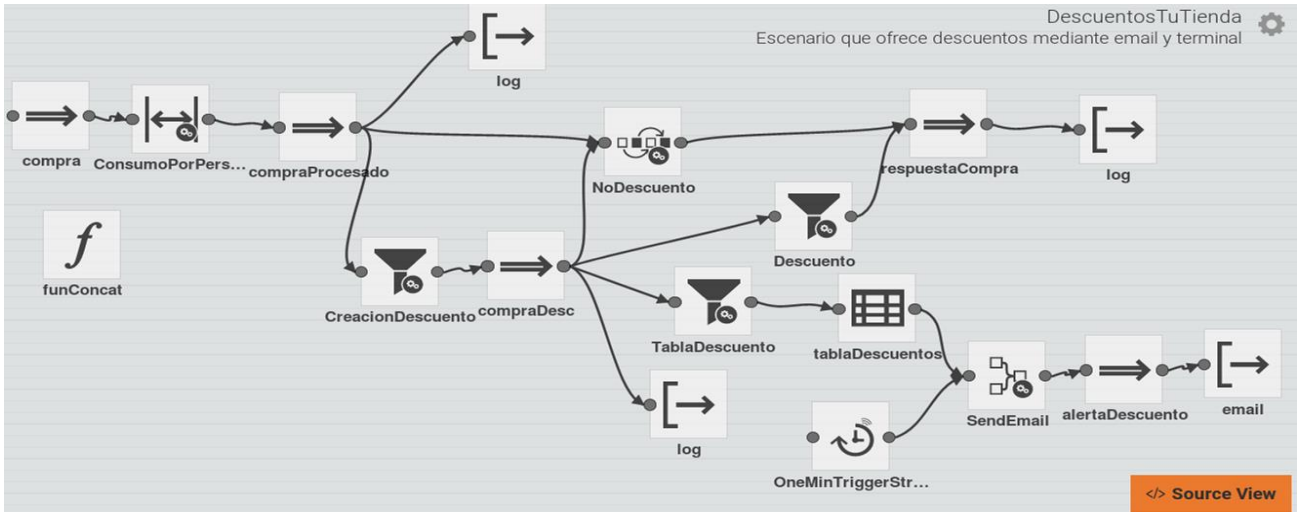


Ilustración 25: Tercer escenario-DescuentosTuTienda vista de diseño

DescuentosTuTienda

```
@App:name('DescuentosTuTienda')
@App:description('Escenario que ofrece descuentos mediante email y terminal')
-- Flujo basico con dos atributos
define stream compra (comprador string, consumo double);
/*
* Flujo con un sumidero o sink asignado de tipo log por lo que se mostrara por el terminal
* Todos los mensajes que se muestren comenzaran por el prefijo: Respuesta:
*/
@sink(type = 'log', prefix = 'Respuesta')
define stream respuestaCompra (mensaje string);
/*
* Flujo con un sumidero o sink asignado de tipo log por lo que se mostrara por el terminal
* Todos los mensajes que se muestren comenzaran por el prefijo: Compra envio log:
*/
@sink(type = 'log', prefix = 'Compra envio log')
define stream compraProcesado(consumo double,comprador string, numCompras long);
/*
* Flujo con un sumidero o sink asignado de tipo log por lo que se mostrara por el terminal
* Todos los mensajes que se muestren comenzaran por el prefijo: Compra envio log:
*/
@sink(type = 'log', prefix = 'Compra con Descuento envio log')
define stream compraDesc(comprador string, descuento bool, porcentaje double);
/*
* Flujo con un sumidero o sink asignado de email, se enviaran con el asunto
* Recordatorio de descuentos-NombreComprador al email pgallegosj11@gmail.com
*/
@sink(type='email', @map(type ='xml'), username='descuentos.tutienda',
address='descuentos.tutienda@gmail.com',password='7ti\a2@j',host='smtp.gmail.com',
port='465',ssl.enable='true',auth='true',content.type='text/html',
subject='Recordatorio de descuentos-{{comprador}}',to='pgallegosj11@gmail.com')
define stream alertaDescuento(triggered_time long,comprador string,porcentaje double);
--Tabla que almacena todos los compradores que tienen descuento y su porcentaje
@PrimaryKey('comprador')
define table tablaDescuentos(comprador string,porcentaje double);
--Funcion que genera el mensaje concatenando dos cadenas y el valor de descuento calculado
define function funConcat[javascript] return string {
    return "Ha recibido un descuento del "+ data[0]+" por su fidelidad";
};
--Evento que se producira cada minuto solo con el timestamp como informacion
define trigger Aviso1Minuto at every 1 min;
```

```
41
42  -- Consulta que calcula el consumo y el numero de compras por comprador
43  @info(name = 'ConsumoPorPersona')
44  from compra#window.length(20)
45  select  sum(consumo) as consumo,comprador as comprador,count() as numCompras
46  group by comprador
47  insert into compraProcesado;
48  /*
49  * Consulta que comprueba si se ha realizado mas de una compra en cuyo caso
50  * calcula un descuento
51  */
52  @info(name='CreacionDescuento')
53  from compraProcesado[numCompras >=2 ]
54  select comprador,true as descuento, ((numCompras-1)*0.1) as porcentaje
55  insert into compraDesc;
56
57  --Consulta que usando funcConcat genera el mensaje de descuento
58  @info(name='Descuento')
59  from compraDesc
60  select  funConcat(porcentaje) as mensaje
61  insert into respuestaCompra;
62  /*
63  * Consulta que cuando se produce un evento de compraProcesado espera 10 segundos
64  * si no se produce un evento de compraDesc gennera un mensaje sin descuento
65  */
66  @info(name='NoDescuento')
67  from not compraDesc for 10 sec and compraProcesado
68  select  convert("Muchas gracias por su compra",'string') as mensaje
69  insert into respuestaCompra;
70  /*
71  * Consulta que almacena en tablaDescuentos todos los compradores con descuento
72  */
73  @info(name='TablaDescuento')
74  from compraDesc
75  select comprador,porcentaje
76  update or insert into tablaDescuentos
77  on tablaDescuentos.comprador==comprador;
78  /*
79  * Consulta que cada vez que se genera un evento Aviso1Minuto genera
80  * un evento alertaDescuento que enviara un email
81  */
82  @info(name='SendEmail')
83  from Aviso1Minuto join tablaDescuentos
84  select *
85  insert into alertaDescuento;
```

Ilustración 26: Código Stream Processor del tercer escenario-DescuentosTuTienda

En función del número de compras realizadas se producirán dos escenarios diferentes. Si es la primera compra, se mostrará un mensaje dando las gracias por su compra, siempre que ese *consumidor* no realice otra compra en los siguientes 10 segundos **(not compraDesc for 10 sec)**. En cambio, si no se trata de la primera compra, se mostrará un mensaje para el cual se hace uso de un **script de JavaScript** indicando que se ha producido un descuento por su fidelidad.

Además, se guarda un registro en una **tabla**, similar a las de una base datos SQL, con el atributo *comprador* como clave primaria (**primary key**) de todos los descuentos que se producen. Mediante el uso de un **trigger**, flujo que solo incluye el timestamp, que genera un evento cada minuto, se realiza un **join** con la tabla y se manda un **email** indicando los descuentos que se han producido.

Para diferenciar los escenarios hace uso de **pattern** o patrón. Si en un evento del stream "compraProcesado" el número de compra es mayor o igual a 2, se genera un evento en el flujo "compraDesc". En caso de que 10 segundos después de un evento del flujo "compraProcesado" no se ha generado un evento del flujo "compraDesc" se genera una respuesta con el mensaje sin el anuncio de descuento. En los patrones "not condicion1 and condicion2" se ejecuta primero la segunda condición del and por lo que el orden escrito es el necesario para que cumpla esta función.

Aunque para este caso se ha hecho uso de un script de Javascript, son multitud de lenguajes los que el Stream Processor permite usar, entre ellos Javascript, phyton, R y scala. Pero sólo se puede usar una función en cada archivo. Para acceder a los diferentes parámetros de las funciones se hace uso de data[i], siendo i la posición del parámetro empezando por la posición cero.

3.2.3.1 Pruebas tercer escenario

En este escenario se han realizado las siguientes pruebas:

1. Llegada del primer evento mediante el simulador de eventos al flujo "compra". El contenido es *Comprador:Pablo, Consumo:100*. La llegada de un evento del flujo "compra" activa la realización de la consulta "ConsumoPorPersona":
   - "ConsumoPorPersona": suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo "compraProcesado" (timestamp=1564078748699) con el contenido *Comprador:Pablo, Consumo:100, numCompras:1*. La llegada de un evento del flujo "compraProcesado" provoca que se muestre por el terminal con el prefijo "Compra envio log" y la realización de la consulta "CreacionDescuento" y "NoDescuento".
     - "CreacionDescuento": comprueba si *numCompras* es mayor o igual a dos. En caso de ser así se inserta en el flujo "compraDesc".
     - "NoDescuento": espera 10 segundos si se produce un evento del flujo "CompraDesc". Dado que no se produce, inserta en el flujo "respuestaCompra" el mensaje de "Muchas gracias por su compra" que se mostrará por el terminal con el prefijo "Compra envio log" cómo se puede ver en la siguiente imagen:

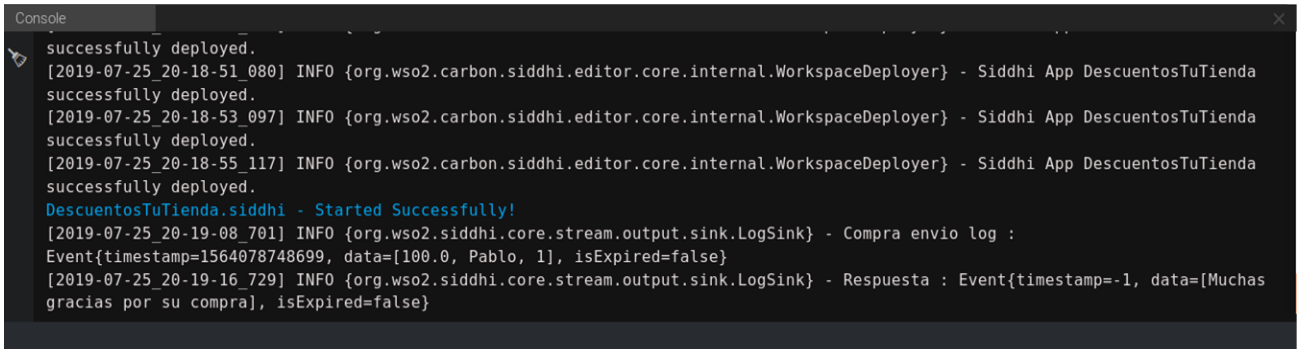


Ilustración 27: Primer evento del tercer escenario-DescuentosTuTienda

2. Llegada del segundo evento mediante el simulador de eventos al flujo "compra". El contenido es *Comprador:Pablo, Consumo:50*. La llegada de un evento del flujo "compra" activa la realización de la consulta "ConsumoPorPersona":
   - "ConsumoPorPersona": suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo "compraProcesado" (timestamp=1564078762785) con el contenido *Comprador:Pablo, Consumo:150, numCompras:2*. La llegada de un evento del flujo "compraProcesado" provoca que se muestre por el terminal con el prefijo "Compra envio log" y la realización de la consulta "CreacionDescuento" y "NoDescuento".

- “CreacionDescuento”: comprueba si *numCompras* es mayor o igual a dos. En este caso, al ser igual a dos, se inserta en el flujo “compraDesc” que se mostrará por el log con el prefijo “Compra con descuento envio log”. Un evento del flujo “compraDesc” provoca la realización de dos consultas: “TablaDescuento” y “Descuento”
  - “TablaDescuento”: guarda el descuento asignado al comprador en la tabla “tablaDescuento”.
  - “Descuento”: genera un evento del flujo “respuestaCompra” mediante la función *funConcat*. Este flujo se mostrará en el log ofreciendo un descuento por su compra, como se puede ver en la siguiente imagen:

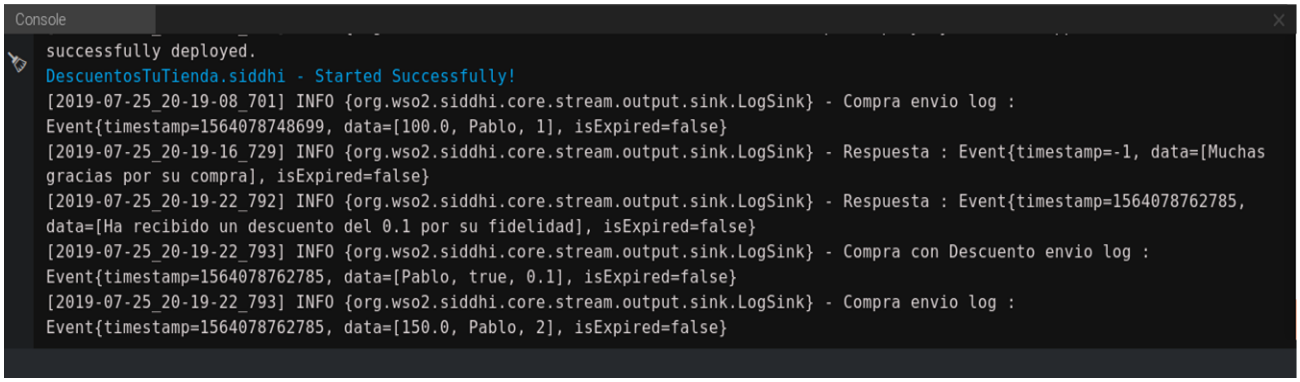


Ilustración 28: Segundo evento del tercer escenario-DescuentosTuTienda

- “NoDescuento”: espera 10 segundos si se produce un evento del flujo “CompraDesc”, dado que se produce no se ejecuta.

3. Llegada del tercer evento mediante el simulador de eventos al flujo “compra”. El contenido es *Comprador:Alba, Consumo:40*. La llegada de un evento del flujo “compra” activa la realización de la consulta “ConsumoPorPersona”:
   - “ConsumoPorPersona”: suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo “compraProcesado” (timestamp=1564078771979) con el contenido *Comprador:Alba, Consumo:40, numCompras:1*. La llegada de un evento del flujo “compraProcesado” provoca que se muestre por el terminal con el prefijo “Compra envio log” y la realización de la consulta “CreacionDescuento” y “NoDescuento”.
     - “CreacionDescuento”: comprueba si *numCompras* es mayor o igual a dos. En caso de ser así se inserta en el flujo “compraDesc”.
     - “NoDescuento”: espera 10 segundos si se produce un evento del flujo “CompraDesc”. **En este caso antes de que pasen 10 segundos se ha producido un evento del flujo “CompraDesc” provocado por el siguiente evento.**
4. Llegada del cuarto evento **(solo unos segundos después del evento anterior)** mediante el simulador de eventos al flujo “compra”. El contenido es *Comprador:Alba, Consumo:30*. La llegada de un evento del flujo “compra” activa la realización de la consulta “ConsumoPorPersona”:
   - “ConsumoPorPersona”: suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo “compraProcesado” (timestamp=1564078777875) con el contenido *Comprador:Alba, Consumo:70, numCompras:2*. La llegada de un evento del flujo “compraProcesado” provoca que se muestre por el terminal con el prefijo “Compra envio log” y la realización de la consulta “CreacionDescuento” y “NoDescuento”.
     - “CreacionDescuento”: comprueba si *numCompras* es mayor o igual a dos. En este caso, al ser igual a dos, se inserta en el flujo “compraDesc” que se mostrará por el log con el

prefijo “Compra con descuento envio log”. Un evento del flujo “compraDesc” **(este evento es el que impide interviene en la query “NoDescuento” anterior)** provoca la realización de dos consultas: “TablaDescuento” y “Descuento”

- “TablaDescuento”: guarda el nuevo descuento asignado al comprador en la tabla “tablaDescuento”.
- “Descuento”: genera un evento del flujo “respuestaCompra” mediante la función *funConcat*. Este flujo se mostrará en el log ofreciendo un descuento por su compra, como se puede ver en la siguiente imagen:

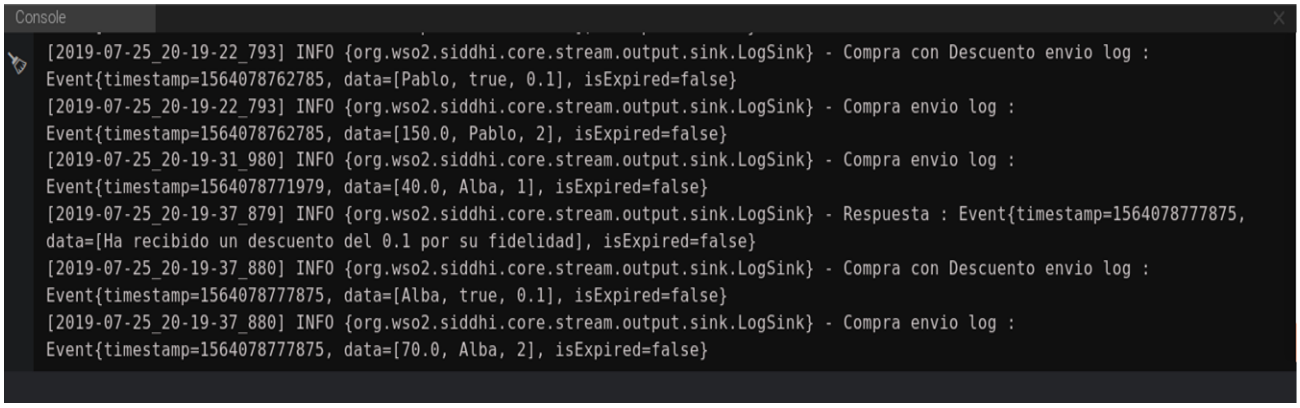


Ilustración 29: Tercer y cuarto evento del tercer escenario-DescuentosTuTienda

- “NoDescuento”: espera 10 segundos si se produce un evento del flujo “CompraDesc”, dado que se produce no se ejecuta.

5. Llegada del quinto evento mediante el simulador de eventos al flujo “compra”. El contenido es *Comprador:Pablo, Consumo:200*. La llegada de un evento del flujo “compra” activa la realización de la consulta “ConsumoPorPersona”:

- “ConsumoPorPersona”: suma el consumo de todos los flujos recibidos cuyo *Comprador* es el mismo y contabiliza cuantos flujos está sumando. Esto se inserta en el flujo “compraProcesado” (timestamp=1564078790946) con el contenido *Comprador:Pablo, Consumo:350, numCompras:3*. La llegada de un evento del flujo “compraProcesado” provoca que se muestre por el terminal con el prefijo “Compra envio log” y la realización de la consulta “CreacionDescuento” y “NoDescuento”.
  - “CreacionDescuento”: comprueba si *numCompras* es mayor o igual a dos. En este caso, al ser tres, se inserta en el flujo “compraDesc” que se mostrará por el log con el prefijo “Compra con descuento envio log”. Un evento del flujo “compraDesc” provoca la realización de dos consultas: “TablaDescuento” y “Descuento”
    - “TablaDescuento”: guarda el nuevo descuento asignado al comprador en la tabla “tablaDescuento”.
    - “Descuento”: genera un evento del flujo “respuestaCompra” mediante la funcion *funConcat*. Este flujo se mostrará en el log ofreciendo un descuento por su compra, como se puede ver en la siguiente imagen:

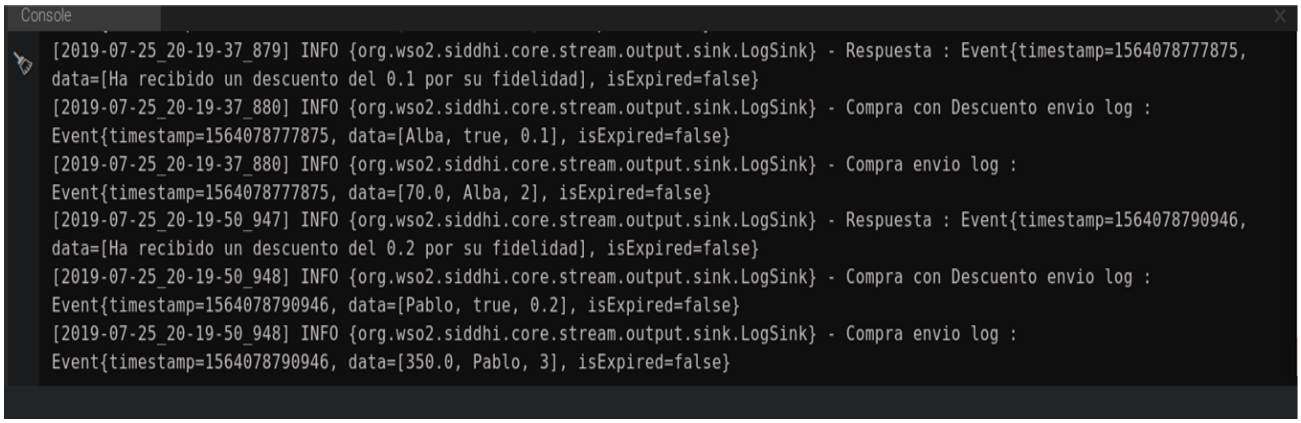


Ilustración 30: Quinto evento del tercer escenario-DescuentosTuTienda

- "NoDescuento": espera 10 segundos si se produce un evento del flujo "CompraDesc", dado que se produce no se ejecuta.

De manera paralela a lo comentado anteriormente cada minuto se genera un evento del flujo "Alerta1Minuto" que se utiliza para recorrer la tabla de descuentos y mandar un email como recordatorio de los descuentos existentes con únicamente la información temporal, el nombre del comprador y el número de compras realizadas:

Ilustración 31: Email recibido del tercer escenario-DescuentosTuTienda

Ilustración 32: Detalle email recibido del tercer escenario-DescuentosTuTienda

3.3 Puesta en marcha del Stream Processor-Worker

Para conocer otro de los modos de funcionamiento que posee el Stream Processor se utilizará el script worker para el escenario real. Este modo posee diversas ventajas como por ejemplo una capacidad de cómputo superior debido, entre otras cosas, a que no necesita cargar una interfaz gráfica.

En primer lugar, creamos el archivo shiddi, para ello, aunque también se podría escribir el código en un editor de texto independiente del WSO2, la opción más recomendable es utilizar el editor comentado anteriormente y exportar el archivo tal y como se mostrará en la siguiente imagen:

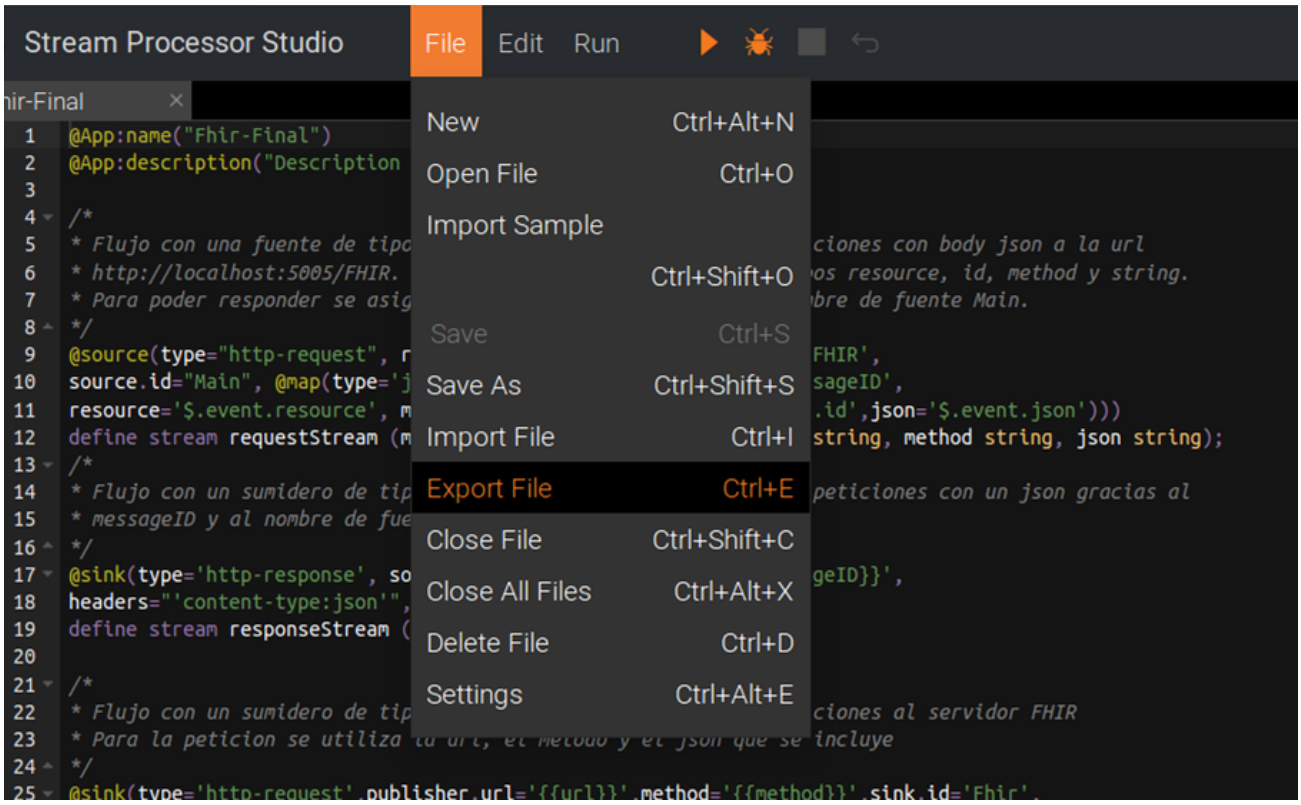
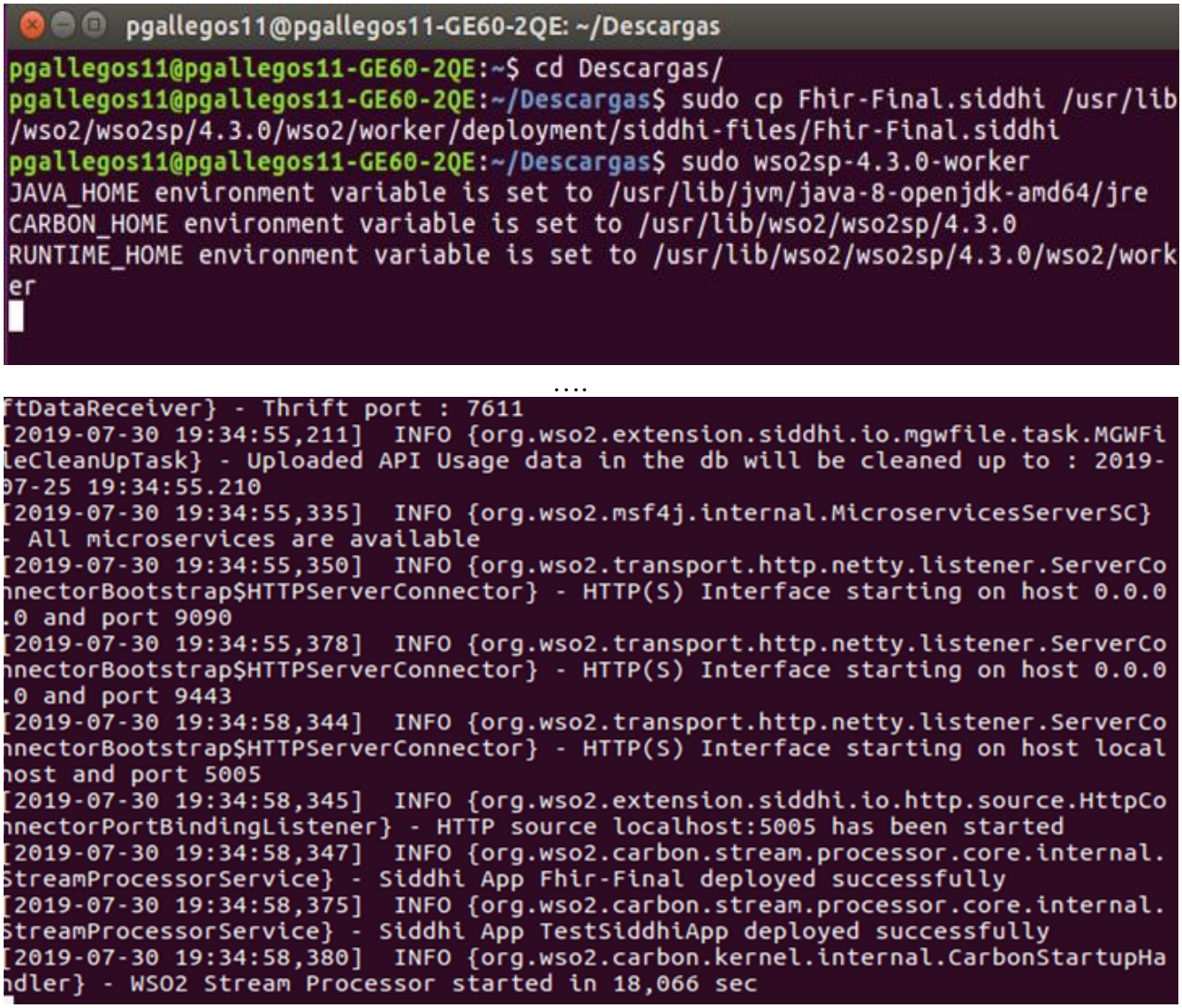


Ilustración 33: Exportando el archivo Fhir-Final.siddhi

Tras hacer clic en Export File se mostrará la siguiente ventana emergente para guardar o abrir el archivo:

Ilustración 34: Ventana emergente al exportar archivo Fhir-Final.siddhi

Una vez obtenido este archivo existen dos opciones para ejecutarlo. La más sencilla es copiar el archivo a la ruta de despliegue: <WSO2-PATH>/wso2sp/4.3.0/wso2/worker/deployment/siddhi-files:



....

Ilustración 35: Ejecución Worker Stream Processor

No obstante, también se podría arrancar el Worker y posteriormente enviar el archivo mediante una petición POST al servidor, en vez de copiar el archivo en la ruta indicada. No obstante, para este envío, al menos en esta versión, los ficheros de código deben estar escritos sin comentarios ni saltos de línea. Dado que Fhir-Final se ha desplegado mediante la copia del archivo, en este caso se va a desplegar MediaCompra, el primer escenario de pruebas. El formato de la petición debe ser la siguiente:

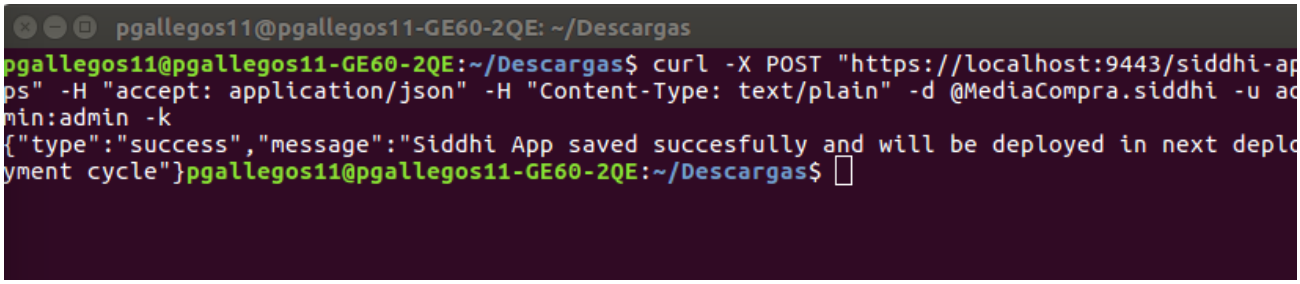


Ilustración 36: Envío de MediaCompra a Worker

En el terminal donde se está ejecutando el worker, nos mostrará los mensajes indicando que MediaCompra se ha guardado en el sistema de archivos y posteriormente que se ha desplegado:

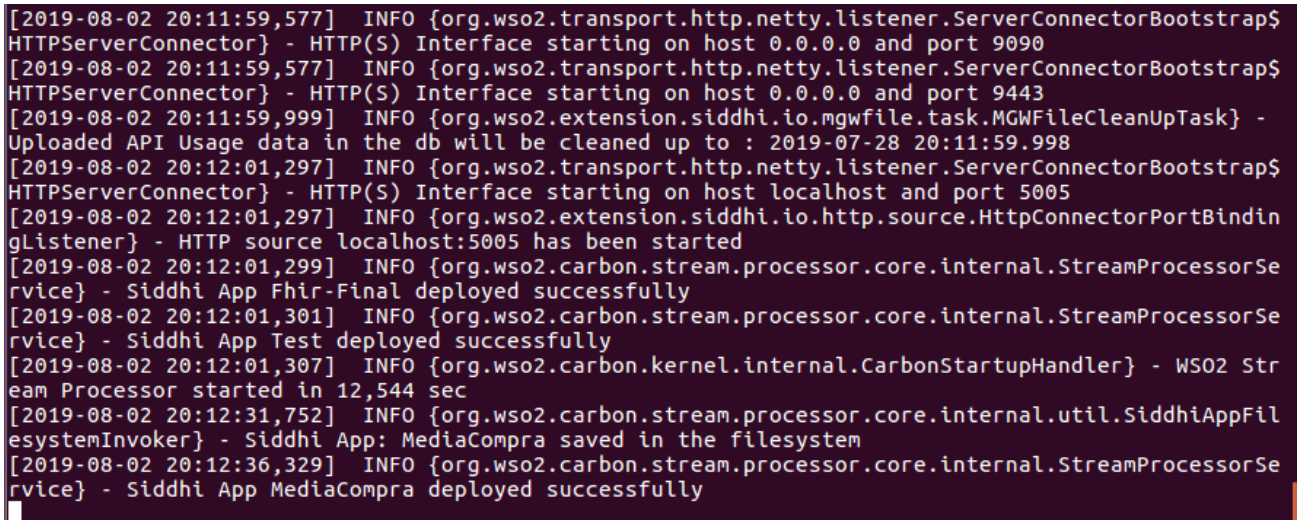


Ilustración 37: Terminal worker tras envío MediaCompra

3.4 Escenario real

Este es el escenario real que se planteó como objetivo al inicio del proyecto, añade nuevos elementos que serán explicados como se explicaron los escenarios anteriores.

3.4.1 Funcionamiento

Este escenario modela lo que se podría dar en un futuro en un hospital o centro sanitario. En este escenario final intervienen 5 elementos:

- Agente de usuario: En este caso usaremos como agente de usuario un terminal y el comando curl. En el caso real lo habitual será usar una aplicación especifica que actúe como cliente FHIR.
- Proxy: Dado que el Stream Processor tiene unos requisitos en cuanto a las peticiones que puede tramitar, se ha añadido este elemento para que realice la conversión del formato FHIR al formato que entiende el Stream Processor y viceversa.
- Stream Processor (2.2.1): Gracias al proxy es transparente para los usuarios, permite acceder al servidor FHIR bajo determinadas condiciones.
- Servidor FHIR (2.4): Servidor donde se almacena toda la información.
- Agente de administrador: Se usarán dos tipos de agentes de administrador. Un terminal que actuará como suscriptor a eventos de interés MQTT y un buzón de correo electrónico. Con anterioridad a la realización de las diferentes pruebas se ha suscrito mediante el uso de mosquitto_sub a la cola de mensajes. Para ver un ejemplo de suscripción consultar el Anexo C.1 Mosquitto.

Se pueden producir dos interacciones diferentes:

- Comunicación normal

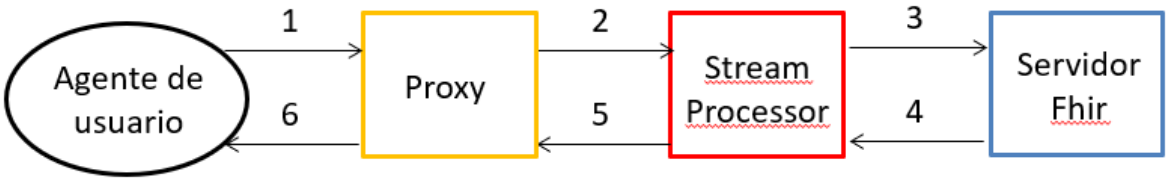


Ilustración 38: Diagrama de interacciones 1

En esta comunicación el usuario realiza una petición siguiendo el formato FHIR al Proxy(1) que posteriormente lo transforma para que el Stream Processor lo pueda procesar y se lo envía(2). Este almacena y procesa información sobre esta petición y la reenvía al servidor FHIR (3). El servidor FHIR responde(4) al Stream Processor que reenvía la respuesta al Proxy(5), que tras darle de nuevo el formato FHIR, responde al usuario que realizó la petición inicial(6).

- Comunicación con error

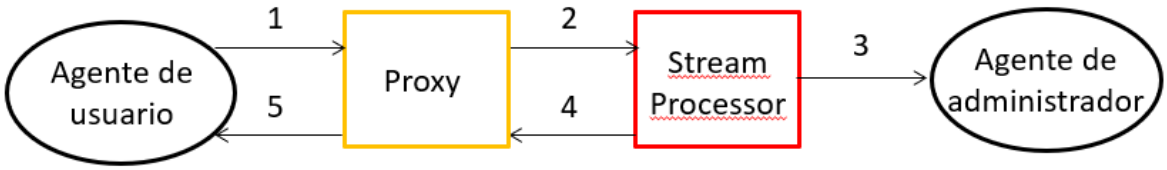


Ilustración 39: Diagrama de interacciones 2

En esta comunicación el usuario realiza una petición siguiendo el formato Fhir al Proxy(1) que posteriormente lo transforma para que el Stream Processor lo pueda procesar y se lo envía(2). Al procesar esta petición detecta que ha superado el umbral de peticiones permitidas al recurso y responde al usuario, previo paso por el

Proxy(4), indicándoselo(5). Simultáneamente notifica esta incidencia a los agentes de administrador.

3.4.1.1 Proxy

```php
<?php
//Funcion write_log: Escribira en su archivo de log el json y la fecha
function write_log($cadena,$tipo)
{
        $arch = fopen("logs/intermediario.txt", "a+");

        fwrite($arch, "[".date("Y-m-d H:i:s.u")." "." " - $tipo ] ".$cadena."\n");
        fclose($arch);
}
//URL Stream Processor
$url = 'http://localhost:5005/FHIR';
//Creacion de recurso cURL
$ch = curl_init($url);
//Creacion de json en funcion de los parametros aceptados por Stream Processor
$url_completa = $_SERVER['REQUEST_URI'];
$method = $_SERVER['REQUEST_METHOD'];
$partes = explode("/", $url_completa);
if (strcmp($method,"GET") == 0 || strcmp($method,"DELETE")==0){
$json='null';
} else if(strcmp($method,"POST")==0){
$json='['.file_get_contents('php://input').']';
$partes[3]="";
} else{
$json='['.file_get_contents('php://input').']';
}
if($partes[2]==null)
$partes[2]="";
if($partes[3]==null)
$partes[3]="";
$data = array(
        'resource' => $partes[2],
        'id' => $partes[3],
        'method' => $method,
        'json' => $json
);

$payload = json_encode(array("event" => $data));
//Escribimos en el log el json
write_log($payload,"INFO-JSON");
//Añadir el json a la peticion
curl_setopt($ch, CURLOPT_POSTFIELDS, $payload);
//Añadir las diferentes opciones
curl_setopt($ch, CURLOPT_HTTPHEADER, array('Content-Type:application/json'));
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
//Ejecutamos la peticion
$result = curl_exec($ch);
//Procesado de la respuesta para que se muestre de la forma mas similar a si
//se realiza la peticion directamente al servidor FHIR
$str= str_replace("\\n","\n",$result);
$str= str_replace("\\r","\r",$str);
$str= str_replace("\\t","\t",$str);
$str= str_replace("  ","\t",$str);
$str= str_replace("\\","",$str);
$str=substr($str,0,-3);
$pos = strpos($str, "{\n\t\"resourceType");
if($pos ===false)
{
//Procesado de la respuesta personalizada de error
$pos = strpos($str, "Error: ");
}
$str=substr($str,$pos);
echo $str."\n";
//Cerrado el recurso cURL
curl_close($ch);
?>
```

Ilustración 40: Código proxy

El Stream Processor tiene algunas limitaciones que hacen necesaria la intervención de un servidor que actúe como proxy. Este proxy adaptará las peticiones y las respuestas de tal manera que el conjunto Proxy-Stream Processor sea transparente para el usuario y no se perciba ningún cambio frente a cuando realiza la petición directamente al servidor FHIR.

Algunas de las limitaciones más importantes del Stream Processor y que el proxy busca solventar son la imposibilidad de incluir recursos en la URI o de usar métodos diferentes a POST.

- Peticiones

Las peticiones FHIR siguen el estándar REST y responden al siguiente formato:

Se realiza una petición HTTP usando el método correspondiente a la dirección http://hapi.fhir.org/baseDstu3/tipoRecurso/idRecurso y en caso de ser necesario incluye un JSON en el cuerpo de la petición.

Este formato es incompatible con el Stream Processor por lo que el proxy enviará una petición POST con el siguiente JSON:

- resource: Indica el tipo de recurso FHIR sobre el que se quiere interactuar. Es el primer campo de la URI en la petición que recibe el Proxy, nombrado anteriormente como "tipoRecurso".
- id: Indica el identificador interno de FHIR del elemento sobre el que se quiera actuar. Cuando el método es POST y se quiere crear un nuevo recurso, lo dejara en blanco. Es el segundo campo de la URI en la petición que recibe el Proxy, nombrado anteriormente como "idRecurso".
- method: Indica el método HTTP que se quiere ejecutar.
- json: En caso de que la petición recibida incluyera un JSON y que el método requiera de este JSON para ejecutarse correctamente, lo incluirá en este campo con el formato necesario para el Stream Processor, simplemente el JSON recibido entre corchetes "[ ]". En caso contrario le asignará el valor null.

- Respuestas

El Stream Processor no muestra correctamente los caracteres especiales incluidos en alguno de sus campos, por ejemplo, el carácter '\n' que equivaldría a un salto de línea es escrito como texto en lugar de como carácter especial. Además, a todas las respuestas les añade el messageID y le proporciona el formato evento.

El Proxy permitirá que estos caracteres sean tratados como caracteres especiales y eliminará toda la información que no provenga del servidor FHIR. En el caso mostrado en el segundo diagrama de interacciones, únicamente dejará el mensaje de alerta.

3.4.1.2 Código Stream Processor

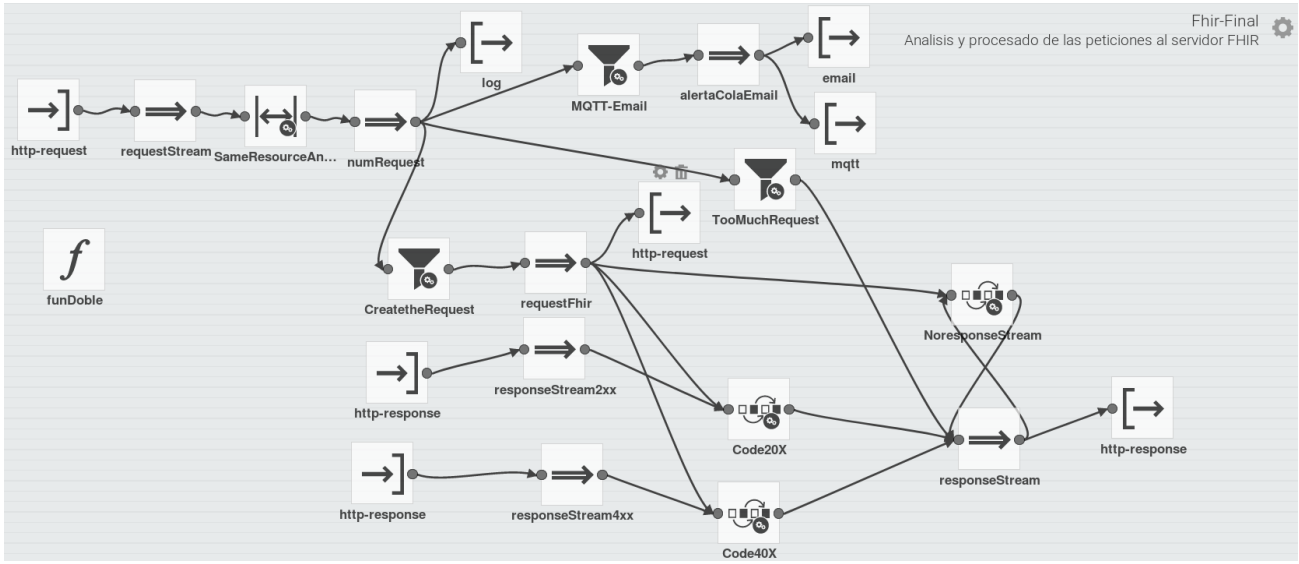
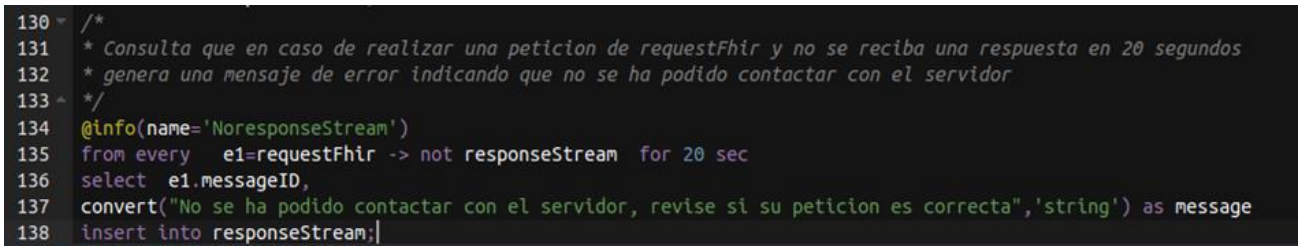


Ilustración 41: Escenario real vista diseño

```
@App:name("Fhir-Final")
@App:description("Analisis y procesado de las peticiones al servidor FHIR")
/*
* Flujo con una fuente de tipo http-request que recibira las peticiones con body json a la url
* http://localhost:5005/FHIR. El json recibido tendra cuatro campos resource, id, method y string.
* Para poder responder se asigna un messageID automatico y el nombre de fuente Main.
*/
@source(type="http-request", receiver.url='http://localhost:5005/FHIR',
source.id="Main", @map(type='json',@attributes(messageID='trp:messageID',
resource='$.event.resource', method='$.event.method', id='$.event.id',json='$.event.json')))
define stream requestStream (messageID string,resource string,id string, method string, json string);
/*
* Flujo con un sumidero de tipo http-response que respondera las peticiones con un json gracias al
* messageID y al nombre de fuente Main
*/
@sink(type='http-response', source.id='Main', message.id='{{messageID}}',
headers="'content-type:json'",@map(type='json'))
define stream responseStream (messageID string,message string);

/*
* Flujo con un sumidero de tipo http-request que realiza las peticiones al servidor FHIR
* Para la peticion se utiliza la url, el metodo y el json que se incluye
*/
@sink(type='http-request',publisher.url='{{url}}',method='{{method}}',sink.id='Fhir',
headers="'Content-type:application/json','User-Agent:Stream Procesor','Host:hapi.fhir.org','Accept: */*'",
@map(type='json', @payload("{{json}}") ) )
define stream requestFhir (url string,method string,json string,messageID string);
/*
* Flujo con una fuente de tipo http-response que recibe las respuestas de las peticiones
* que se realicen al servidor FHIR siempre y cuando el codigo http es 2XX
*/
@source(type='http-response', sink.id='Fhir', http.status.code='2\d+', @map(type='text',
regex.A='((.|\n)*)', @attributes(message='A[1]')))
define stream responseStream2xx(message string);
/*
* Flujo con una fuente de tipo http-response que recibe las respuestas de las peticiones
* que se realicen al servidor FHIR siempre y cuando el codigo http es 4XX
* en este caso es unicamente a modo demostrativo y se tratara de igual forma que si es 2XX
*/
@source(type='http-response', sink.id='Fhir', http.status.code='4\d+', @map(type='text',
regex.A='((.|\n)*)', @attributes(message='A[1]')))
define stream responseStream4xx(message string);
/*
* Flujo con una fuente de tipo log
* Calcula el numero de peticiones que se han realizado a un determinado recurso
*/
@sink(type = 'log', prefix = 'Num Request:')
define stream numRequest (resource string,id string,method string, num long, json string, messageID string);
/*
* Flujo con un sumidero o sink asignado de tipo mqtt y tipo email,
* se enviaran con el asunto Alerta de WSO2 Stream Processor-NombreRecuro al email pgallegosj11@gmail.com
* Se inclura en la cola con el topic 'Alerta'.
* El resto de parametros de la sink mqtt son los por defecto para poder explicarlos con mayor claridad.
* En ambos caso se incluira tanto el tipo de recurso como el identificador
*/
@sink(type='email', @map(type ='json'), username='hospital.prueba11', address='hospital.prueba11@gmail.com',
password='l#QT9iSuasuT',host='smtp.gmail.com',port='465',ssl.enable='true',auth='true',
content.type='text/html',subject='Alerta de Wso2 Stream Processor-{{resource}}',to='pgallegosj11@gmail.com')
@sink(type='mqtt', url= 'tcp://localhost:1883', topic='Alerta', clean.session='true', message.retain='false',
quality.of.service= '1', keep.alive= '60',connection.timeout='30', @map(type='json'))
define stream alertaColaEmail(resource string, id string, num long);
```

```
/*
* Dado que solo se puede incluir un script esta funcion realiza dos tareas:
* 1 Correccion de formato: Adaptar el json al formato correcto
* 2 CreateUrl: Construir la url para el servidor fhir
*/
define function funDoble[JavaScript] return string {
    var response = "";
    var  resourceType = data[0];
    if (resourceType.charAt(0).equals('[')) {
        //Correccion de formato
        response = resourceType.substring(1, resourceType.length() - 1);
    } else if(!resourceType.equals("null")) {
        //CreateUrl
        response = "http://hapi.fhir.org/baseDstu3/" + resourceType + "/" + data[1];
    }
return response;
};
/*
* Consulta que contabiliza cuantas peticiones de las ultimas 10 se realizan al cada id y recurso
* y lo inserta en el flujo numRequest
*/
@info(name = 'SameResourceAndIDCount')
from requestStream#window.time(10 min)
select resource,id,method,  count() as num,json,messageID
    group by  resource, id
insert into numRequest;
/*
* Consulta que en funcion del numero de peticiones, obviando el id vacio que es para
* la creacion de un nuevo id, genera un evento de tipo alertaColaEmail que generara
* un email y un mensaje a una cola MQTT
*/
@info(name = 'MQTT-Email')
from numRequest[num >= 5 and id!=""]
select  resource, id,num
insert into alertaColaEmail;
/*
* Consulta que si no se sobrepasan las 3 peticiones al recurso-id, genera un evento del flujo
* requestFhir que se enviara al servidor Fhir
*/
@info(name='CreatetheRequest')
from numRequest [id=="" or num < 5]
select funDoble(resource,id) as url, method as method,funDoble(json) as json, messageID
insert into requestFhir;
/*
* Consulta que si se sobrepasan las 3 peticiones al recurso-id, genera un mensaje de alerta
* indicando que se han realizado demsaiadas peticiones al recurso
*/
@info(name='TooMuchRequest')
from numRequest[num >= 5 and id!=""]
select messageID,
convert("Error: Peticion bloqueada, demasiadas peticiones al recurso",'string') as message
insert into responseStream;
/*
* Consulta que procesa las respuestas con el codigo 2XX y las inserta en el flujo
* responseStream que respondera a la peticion que recibio el stream processor
*/
@info(name='Code20X')
from every  e1=requestFhir  ->  e2=responseStream2xx
select e1.messageID  as messageID, e2.message as message
insert into responseStream;
/*
* Consulta que procesa las respuestas con el codigo 4XX y las inserta en el flujo
* responseStream que respondera a la peticion que recibio el stream processor
*/
@info(name='Code40X')
from every e1=requestFhir  ->  e2=responseStream4xx
select e1.messageID as messageID, e2.message as message
insert into responseStream;
```



```
/*
* Consulta que en caso de realizar una peticion de requestFhir y no se reciba una respuesta en 20 segundos
* genera una mensaje de error indicando que no se ha podido contactar con el servidor
*/
@info(name='NoresponseStream')
from every   e1=requestFhir -> not responseStream  for 20 sec
select  e1.messageID,
convert("No se ha podido contactar con el servidor, revise si su peticion es correcta",'string') as message
insert into responseStream;
```

Ilustración 42: Código Stream Processor escenario final

La fuente de entrada de eventos es mediante peticiones HTTP POST y recibe un JSON, con el formato explicado en el apartado anterior.

Este evento es procesado y se comprueba cuántas veces se ha producido algún acceso, independientemente de si este acceso es para lectura o modificación, al recurso solicitado. En función del número de consultas pueden darse las situaciones explicadas en los diagramas uno y dos del apartado anterior:

- Si se han producido al menos 5 peticiones al recurso en los últimos 10 minutos (**window.time(10 min)**), se responde a la petición HTTP con un mensaje indicando que se ha producido un error. Además, para registrar esta infracción y a modo de alerta, se inserta en una **cola MQTT** con el topic "Alerta" y se envía un email.
- En caso de que no haberse alcanzado las cinco peticiones, se construye la URI para la petición, se modifica el JSON en caso de existir para que cumpla los criterios necesarios de FHIR. Se realiza la **petición HTTP** al servidor FHIR y la **respuesta,** si existe (en caso de no recibirla en 20 segundos se considerará un error y se mostrará como tal), se procesa en función de si el **código recibido es 20X o 40X,** aunque en este caso se actúa de igual manera para ambos casos y solo se añade a modo explicativo. En caso de no recibir respuesta pasados 20 segundos, lanzará un mensaje de error. Por último, la respuesta recibida desde el servidor FHIR es devuelta como respuesta al cliente que solicitó la petición inicialmente.

3.4.1.2.1 Explicación del código Stream Processor

Existen algunas funciones del Stream Processor que no se han visto en los ejemplos anteriores y debido a su complejidad requieren una explicación más detallada:

- Como se indicó en el segundo escenario de prueba, cada código del Stream Processor únicamente puede incluir un script. Dado que para este escenario era necesario el uso de dos funcionalidades diferentes ambas se han incluido en la misma función usando un **if** para diferenciar su funcionamiento.
    - Corrección de formato: Elimina los "[ ]" incluidos en el campo json en el Proxy para que sean tratados correctamente por el servidor FHIR.
    - CreateURL: En función de los campos del json creado en el Proxy, construye la URL siguiendo el estándar FHIR.
- El sumidero o sink del flujo "requestFhir" tiene configurados varios parámetros dinámicamente, en función de los valores del evento que se procese. El método, la URL y el json que se incluirá se leerán del evento que se está procesando en ese momento.
- El sumidero o sink del flujo "alertaTipoCola" permite su publicación en una cola MQTT. Para ello hay que configurar los diferentes parámetros, al menos los siguientes:
    - URL: URL del bróker de MQTT. Obligatorio.
    - topic: Topic en el que será publicado en la cola MQTT. Obligatorio.
    - quality.of.service:
        - 0: El evento solo se enviará una vez. No hay confirmación de que llegue a la cola

        - 1: El evento es enviado al menos una vez. Puede llegar más de una vez a la cola.
        - 2: El evento se recibirá solo una vez.
    - clean.session: Si su valor es true cada vez que se detenga la conexión limpiará la información anterior sobre la cola.
    - message.retain: Si su valor es true, el último mensaje que se envíe a la cola se almacenará.
    - keep.alive: Tiempo máximo que la conexión entre el cliente y el bróker está encendida sin que se produzca ningún nuevo evento.
    - connection.timeout: Tiempo máximo que se intentará conectar con el bróker MQTT.

- Aunque los flujos "alertaTipoEmail" y "alertaTipoCola" reciban la información de peticiones idénticas, éstas no pueden ser unidas ya que de una petición solo pueda darse una salida y un flujo no puede tener dos sumideros asignados.
- Las consultas "Code20X" y "Code40X" se realizan utilizando un tipo de patrón distinto "from every flujo e1=requestFhir→ flujo e2= responseStream2xx". El funcionamiento de este patrón es el siguiente: para cada evento del flujo requestFhir esperará un evento del flujo responseStream2xx, si este se produce antes de un nuevo evento del flujo requestFhir se realizará la consulta, en caso contrario volverá a esperar a un nuevo evento del flujo responseStream2xx. Si se quisiera que solo se diera para el primer evento del flujo requestFhir se podría eliminar el every y esta consulta solo se realizaría una vez. Para acceder a atributos se debe indicar a que flujo pertenece de la siguiente manera: nombreFlujo.nombreAtributo en este caso requestFhir.messageID. Para facilitar la lectura de la sentencia se asignan nombres a los flujos con el objetivo de no tener que escribir el nombre del flujo completo, por lo que en este caso es lo mismo usar requestFhir.messageID que e1.messageID.

3.4.2 Pruebas realizadas

Las pruebas y el desarrollo se han realizado teniendo en cuenta que el objetivo principal con el que se quería trabajar era el recurso Person. No obstante, es extrapolable para cualquier otro recurso del servidor FHIR y, sin ningún cambio en el código, se pueden realizar las mismas operaciones con resultados satisfactorios para cualquier otro recurso. A modo de ejemplo se incluirán las pruebas con el recurso Patient. Todas las peticiones al intermediario tienen el mismo formato que tendrían si se lanzaran directamente al servidor FHIR.

Para estas pruebas, como se ha comentado anteriormente se usará como agente de usuario un terminal y el comando curl que realizará las peticiones al Proxy.

3.4.2.1 Probando los diferentes métodos HTTP

- GET

Cuando un agente de usuario quiere obtener información de un recurso del servidor FHIR, debe realizar una petición tipo GET al servidor intermediario. Para ello en la URI se debe incluir el tipo de recurso y el identificador de este, como si se realizara al servidor FHIR directamente. Esta petición se realiza mediante el comando curl a la url http://localhost/intermediario.php/Person/2036654.

El proxy recibe esta petición y genera una petición POST a la url http://localhost:5005/FHIR del Stream Processor con el siguiente JSON:

```
{
      "event":
      {
            "resource": "Person",
            "id": "2036654",
            "method": "GET",
            "json": "null"
      }
}
```

Esta petición es recibida como un evento del flujo "requestStream". El contenido es *"messageID": "082b601d-76fa-4f73-80c4-bb24ee00f70a","resource":"Person","id:"2036654","method":"GET", "json":"null"*.La llegada de un evento del flujo "requestStream" activa la realización de la consulta "SameResourceAndIDCount", consulta que únicamente añade un campo que contabiliza el número de peticiones recibidas a este recurso quedando con el siguiente contenido: *"messageID": "082b601d-76fa-4f73-80c4-bb24ee00f70a", "resource": "Person", "id": "2036654","method":"GET", "num": "1", "json": "null"* y lo inserta en el flujo "numRequest" que lo muestra por el terminal ya que tiene asignado un sumidero de tipo log. La llegada de un evento del flujo "numRquest" provoca las siguientes consultas:

- "TooMuchRequest": No se ejecuta porque el número de peticiones es menor que 5 y el campo id no está vacío.
- "MQTT-Email": No se ejecuta porque el número de peticiones es menor que 5 y el campo id no está vacío.
- "CreateTheRequest": Dado que el campo num es menor que 5, mediante el uso de la función "funDoble" genera la url para posteriormente insertarlo como evento del flujo "requestFhir" con el siguiente contenido: *"url":"http://hapi.fhir.org/baseDstu3/Person/2036654","method": "GET","json":"null","messageID": "082b601d-76fa-4f73-80c4-bb24ee00f70a"*. Al tener asignado un sumidero de tipo http-request con campos publish.url y method dinámicos realizará una petición del tipo indicado en el campo method, en este caso GET y a la url indicada en el

campo url, en este caso http://hapi.fhir.org/baseDstu3/Person/2036654. La llegada de un evento del flujo "requestFhir" provoca la realización de las siguientes consultas:

- "Code20X": Dado que se produce un evento del flujo "responseSteam2xx" recibido mediante la fuente de tipo http-response con la respuesta del servidor FHIR a la petición realizada por el sumidero asignado al flujo "requestFhir" se ejecuta. Recoge la respuesta recibida del evento del flujo "responseStream2xx" junto al messageID del evento del flujo "requestFhir" que provoco esta respuesta y lo inserta en el flujo "reponseStream". El flujo "responseStream" tiene asignado un sumidero de tipo http-response, por lo tanto, cuando se produce un evento de este flujo se responde a la petición que tenía el mismo source.id (Main) y el mismo messageID que este evento (082b601d-76fa-4f73-80c4-bb24ee00f70a)
- "Code40X": No se ejecuta dado que no recibe ningún evento del flujo responseStream4XX.
- "NoresponseStream": No se ejecuta dado que en menos de 20 segundos desde que se produjo el evento del flujo "requestFhir" se produce un evento del flujo "responseStream".

El evento del flujo "responseStream" ha sido enviado al Proxy como respuesta a la petición que realizó anteriormente. El proxy procesa esta respuesta y elimina campos del JSON innecesarios, el titulo event y el campo messageID, y activa correctamente los caracteres especiales como '\n' y la envía al agente de usuario tal y como podemos ver en la siguiente imagen:

Ilustración 43: Petición GET realizada por el agente de usuario al intermediario y respuesta

- PUT

Cuando un agente de usuario quiere modificar un recurso en el servidor FHIR, debe realizar una petición tipo PUT al servidor intermediario. Para ello en la URI se debe incluir el tipo de recurso junto a su identificador y un JSON con el contenido del recurso que quiere modificar, como si se realizara al servidor FHIR directamente. Esta petición se realiza mediante el comando curl a la url http://localhost/intermediario.php/Person/2036654 y usando como cuerpo del mensaje el contenido del archivo

editPerson.json[2]:

```
{
  "resourceType": "Person",
  "id": "2036654",
  "name": [
    {
      "use": "official",
      "text": "Jackie Burton",
      "family": "Burton",
      "given": [ "Jackie" ],
      "prefix": [
        "Ms"
      ]
    }
  ],
  "gender": "female"
}
```

El proxy recibe esta petición y genera una petición POST al Stream Processor a la url http://localhost:5005/FHIR con el siguiente JSON:

```
{
      "event":
      {
            "resource": "Person",
            "id": "2036654",
            "method": "PUT",
            "json": "editPerson.json"
      }
}
```

Esta petición es recibida como un evento del flujo "requestStream". El contenido es *"messageID": "5fe10dd5-2e4e-4a7e-af93-cb27c0d09f76","resource":"Person","id:"2036654","method":"PUT", "json":"editPerson.json"*. La llegada de un evento del flujo "requestStream" activa la realización de la consulta "SameResourceAndIDCount", consulta que únicamente añade un campo que contabiliza el número de peticiones recibidas a este recurso quedando con el siguiente contenido: *"messageID": "5fe10dd5-2e4e-4a7e-af93-cb27c0d09f76", "resource": "Person", "id": "2036654","method":"PUT", "num": "2", "json": "editPerson.json"* y lo inserta en el flujo "numRequest" que lo muestra por el terminal ya que tiene asignado un sumidero de tipo log. La llegada de un evento del flujo "numRquest" provoca las siguientes consultas:

[2] A partir de ahora cuando se use el nombre editPerson.json se refiere a su contenido completo ya que con el fin de no molestar la lectura no se va a copiar de nuevo este archivo.

- "TooMuchRequest": No se ejecuta porque el número de peticiones es menor que 5 y el campo id no está vacío.
- "MQTT-Email": No se ejecuta porque el número de peticiones es menor que 5 y el campo id no está vacío.
- "CreateTheRequest": Dado que el campo num es menor que 5, mediante el uso de la función "funDoble" genera la url para posteriormente insertarlo como evento del flujo "requestFhir" con el siguiente contenido: *"url":"http://hapi.fhir.org/baseDstu3/Person/2036654","method": "PUT","json":"editPerson.json","messageID": "5fe10dd5-2e4e-4a7e-af93-cb27c0d09f76"*. Al tener asignado un sumidero de tipo http-request con campos publish.url y method dinámicos realizará una petición del tipo indicado en el campo method, en este caso PUT y a la url indicada en el campo url, en este caso http://hapi.fhir.org/baseDstu3/Person/2036654. La llegada de un evento del flujo "requestFhir" provoca la realización de las siguientes consultas:
    - "Code20X": Dado que se produce un evento del flujo "responseSteam2xx" recibido mediante la fuente de tipo http-response con la respuesta del servidor FHIR a la petición realizada por el sumidero asignado al flujo "requestFhir" se ejecuta. Recoge la respuesta recibida del evento del flujo "responseStream2xx" junto al messageID del evento del flujo "requestFhir" que provocó esta respuesta y lo inserta en el flujo "reponseStream". El flujo "responseStream" tiene asignado un sumidero de tipo http-response, por lo tanto, cuando se produce un evento de este flujo se responde a la petición que tenía el mismo source.id (Main) y el mismo messageID que este evento (082b601d-76fa-4f73-80c4-bb24ee00f70a)
    - "Code40X": No se ejecuta dado que no recibe ningún evento del flujo responseStream4XX.
    - "NoresponseStream": No se ejecuta dado que en menos de 20 segundos desde que se produjo el evento del flujo "requestFhir" se produce un evento del flujo "responseStream".

El evento del flujo "responseStream" ha sido enviado al Proxy como respuesta a la petición que realizó anteriormente. El proxy procesa esta respuesta y elimina campos del JSON innecesarios, el titulo event y el campo messageID, y activa correctamente los caracteres especiales como '\n' y la envía al agente de usuario tal y como podemos ver en la siguiente imagen:

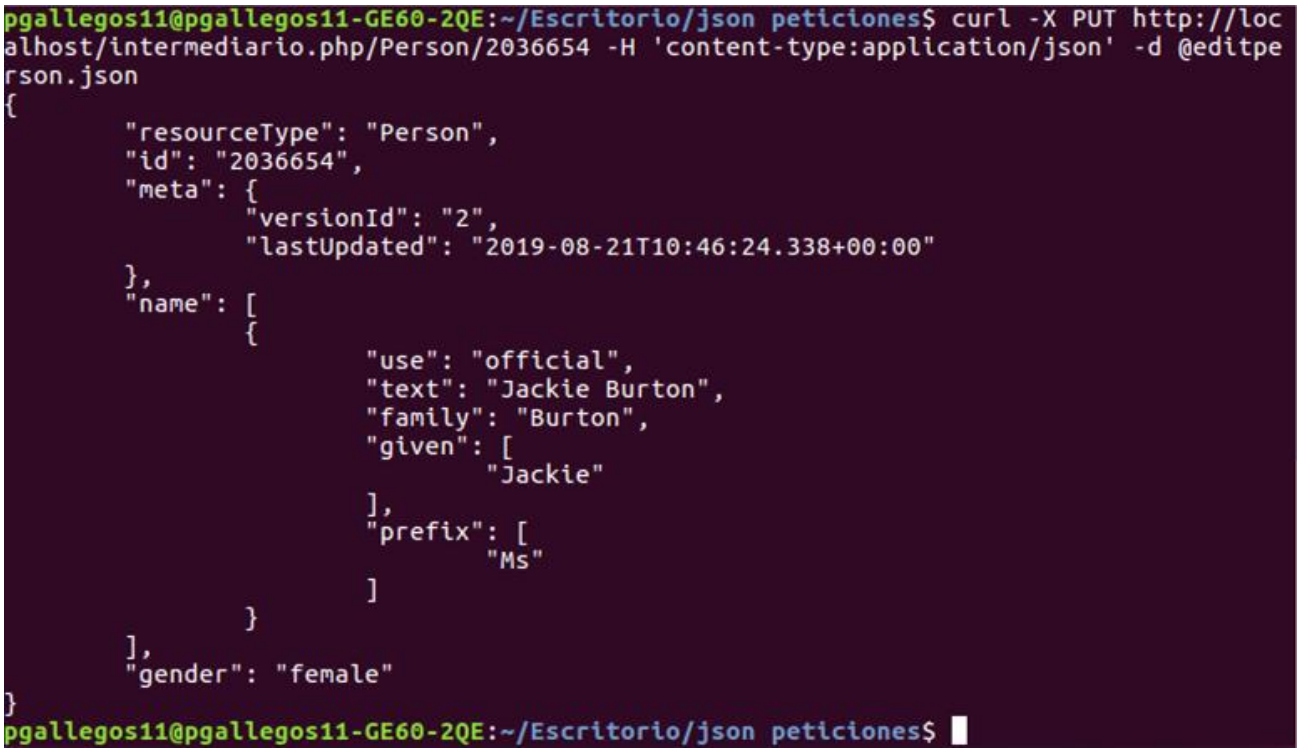


Ilustración 44: Petición PUT realizada por el agente de usuario al intermediario y respuesta

- POST

Cuando un agente de usuario quiere crear un nuevo recurso en el servidor FHIR, debe realizar una petición tipo POST al servidor intermediario. Para ello en la URI se debe incluir el tipo de recurso y JSON con el contenido del recurso que quiere crear, como si se realizara al servidor FHIR directamente:

Cuando un agente de usuario quiere modificar un recurso en el servidor FHIR, debe realizar una petición tipo POST al servidor intermediario. Para ello en la URI se debe incluir el tipo de recurso y un JSON con el contenido del recurso que quiere crear, como si se realizara al servidor FHIR directamente. Esta petición se realiza mediante el comando curl a la url http://localhost/intermediario.php/Person y usando como cuerpo del mensaje el contenido del archivo newperson.json[3]:

```
{
  "resourceType": "Person",
  "name": [
    {
      "use": "official",
      "text": "Jack Burton",
      "family": "Burton",
      "given": ["Jack" ],
      "prefix": [
        "Mr"
      ]
    }
```

[3] A partir de ahora cuando se use el nombre newperson.json se refiere a su contenido completo ya que con el fin de no molestar la lectura no se va a copiar de nuevo este archivo.

```
  ],
  "gender": "male"
}
```

El proxy recibe esta petición y genera una petición POST al Stream Processor a la url http://localhost:5005/FHIR con el siguiente JSON:

```
{
      "event":
      {
            "resource": "Person",
            "id": "",
            "method": "POST",
            "json": "newperson.json"
      }
}
```

Esta petición es recibida como un evento del flujo "requestStream". El contenido es *"messageID": "684cef9c5-4c17-4262-9c42-a53ecacee2e7","resource":"Person","id:"","method":"POST", "json":"newperson.json"*. La llegada de un evento del flujo "requestStream" activa la realización de la consulta "SameResourceAndIDCount", consulta que únicamente añade un campo que contabiliza el número de peticiones recibidas a este recurso quedando con el siguiente contenido: *"messageID": "684cef9c5-4c17-4262-9c42-a53ecacee2e7", "resource": "Person", "id": "","method":"POST", "num": "1", "json": "newperson.json"* y lo inserta en el flujo "numRequest" que lo muestra por el terminal ya que tiene asignado un sumidero de tipo log. La llegada de un evento del flujo "numRquest" provoca las siguientes consultas:

- "TooMuchRequest": No se ejecuta porque el campo id está vacío.
- "MQTT-Email": No se ejecuta porque el campo id está vacío.
- "CreateTheRequest": Dado que el campo id está vacío, mediante el uso de la función "funDoble" genera la url para posteriormente insertarlo como evento del flujo "requestFhir" con el siguiente contenido: *"url":"http://hapi.fhir.org/baseDstu3/Person","method": "POST","json":"newperson.json","messageID": "684cef9c5-4c17-4262-9c42-a53ecacee2e7"*. Al tener asignado un sumidero de tipo http-request con campos publish.url y method dinámicos realizará una petición del tipo indicado en el campo method, en este caso POST y a la url indicada en el campo url, en este caso http://hapi.fhir.org/baseDstu3/Person. La llegada de un evento del flujo "requestFhir" provoca la realización de las siguientes consultas:
    - "Code20X": Dado que se produce un evento del flujo "responseSteam2xx" recibido mediante la fuente de tipo http-response con la respuesta del servidor FHIR a la petición realizada por el sumidero asignado al flujo "requestFhir" se ejecuta. Recoge la respuesta recibida del evento del flujo "responseStream2xx" junto al messageID del evento del flujo "requestFhir" que provoco esta respuesta y lo inserta en el flujo "reponseStream". El flujo "responseStream" tiene asignado un sumidero de tipo http-response, por lo tanto, cuando se produce un evento de este flujo se responde a la petición que tenía el mismo source.id (Main) y el mismo messageID que este evento (684cef9c5-4c17-4262-9c42-a53ecacee2e7)
    - "Code40X": No se ejecuta dado que no recibe ningún evento del flujo responseStream4XX.

- “NoresponseStream”: No se ejecuta dado que en menos de 20 segundos desde que se produjo el evento del flujo “requestFhir” se produce un evento del flujo “responseStream”.

El evento del flujo “responseStream” ha sido enviado al Proxy como respuesta a la petición que realizó anteriormente. El proxy procesa esta respuesta y elimina campos del JSON innecesarios, el titulo event y el campo messageID, y activa correctamente los caracteres especiales como ‘\n’ y la envía al agente de usuario tal y como podemos ver en la siguiente imagen:

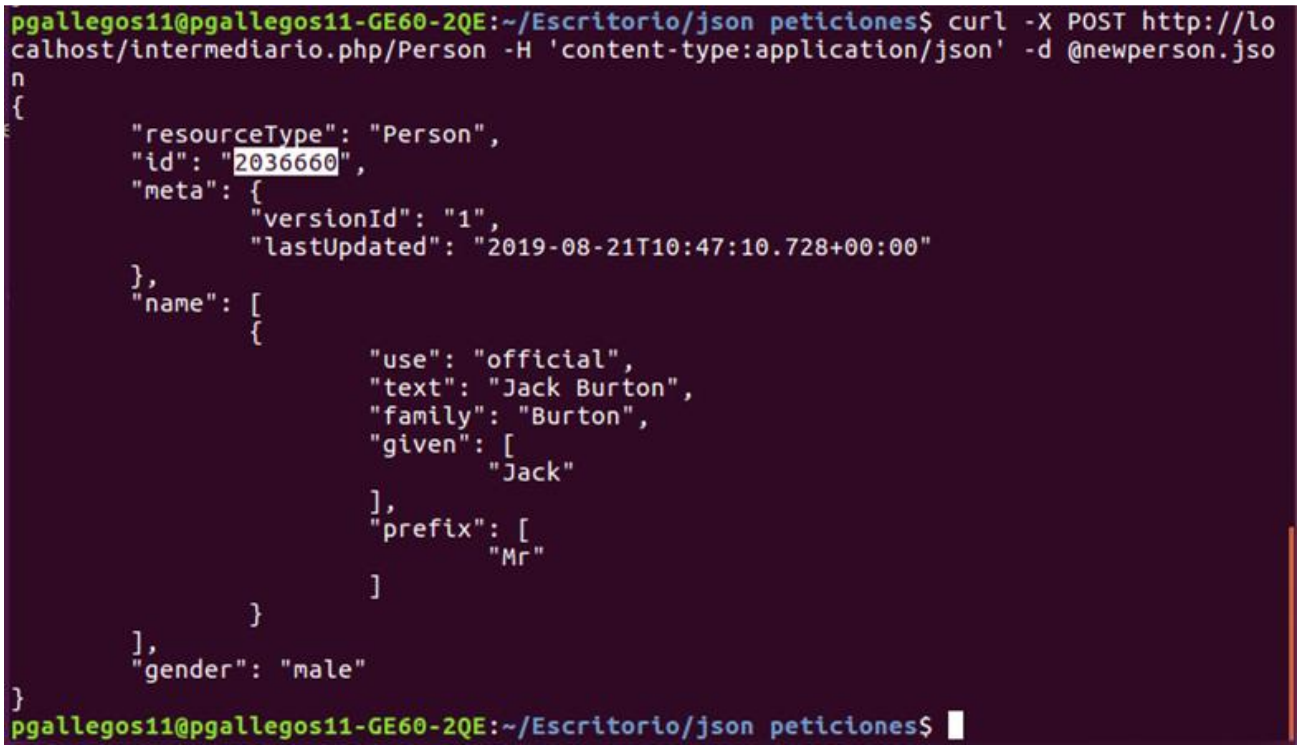


Ilustración 45: Petición POST realizada por el agente de usuario al intermediario y respuesta

Una vez creado, el agente de usuario puede comprobar que el recurso se ha creado correctamente realizando otra prueba GET, que seguiría el mismo flujo que en la prueba GET anterior.

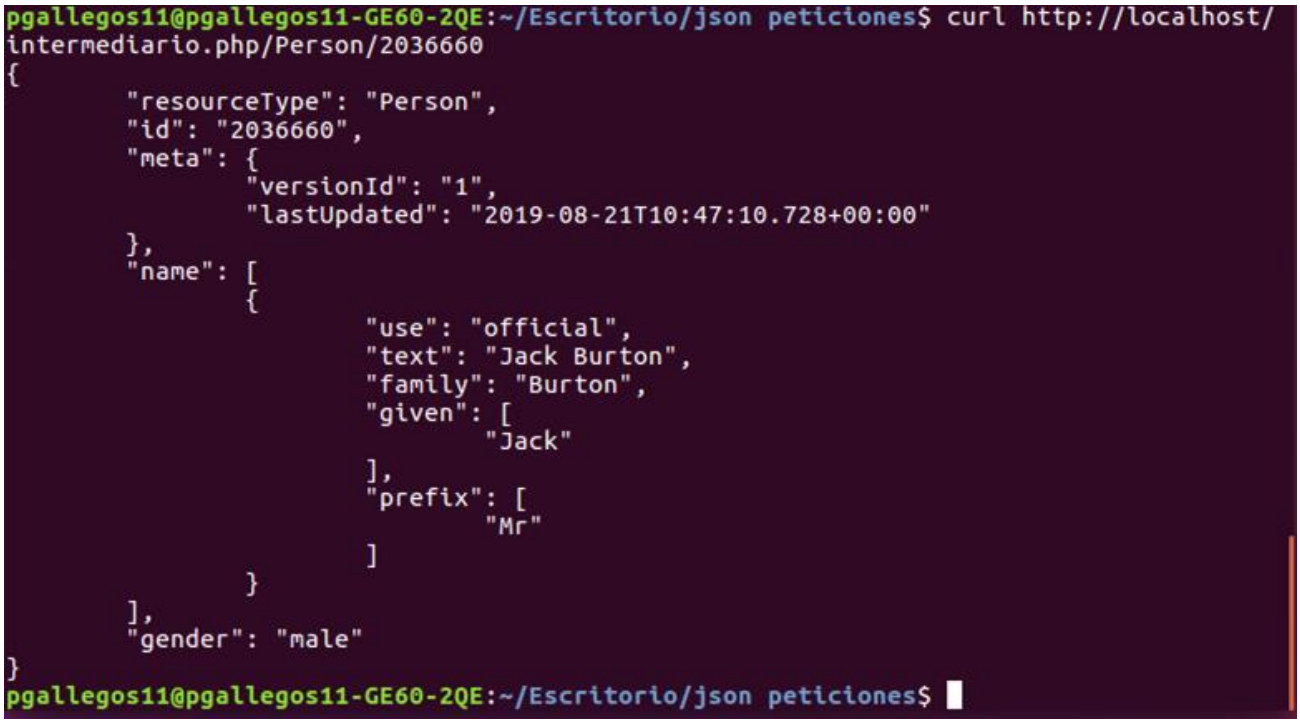


Ilustración 46: Petición GET realizada por el agente de usuario al intermediario para comprobar el POST anterior y respuesta

- DELETE

Cuando un agente de usuario quiere eliminar un recurso del servidor FHIR, debe realizar una petición tipo DELETE al servidor intermediario. Para ello en la URI se debe incluir el tipo de recurso y el identificador de este, como si se realizara al servidor FHIR directamente. Esta petición se realiza mediante el comando curl a la url http://localhost/intermediario.php/Person/2036660.

El proxy recibe esta petición y genera una petición POST a la url http://localhost:5005/FHIR del Stream Processor con el siguiente JSON:

```
{
      "event":
      {
            "resource": "Person",
            "id": "2036660",
            "method": "DELETE",
            "json": "null"
      }
}
```

Esta petición es recibida como un evento del flujo "requestStream". El contenido es *"messageID": "16709ba2-a2a4-4f98-bb7f-02b5cef19d17","resource":"Person","id:"2036660","method":"DELETE", "json":"null".*La llegada de un evento del flujo "requestStream" activa la realización de la consulta "SameResourceAndIDCount", consulta que únicamente añade un campo que contabiliza el número de peticiones recibidas a este recurso quedando con el siguiente contenido: *"messageID": "16709ba2-a2a4-4f98-bb7f-02b5cef19d17", "resource": "Person", "id": "2036660","method":"DELETE", "num": "1", "json": "null"* y lo inserta en el flujo "numRequest" que lo muestra por el terminal ya que tiene asignado un sumidero de tipo log. La llegada de un evento del flujo "numRquest" provoca las siguientes consultas:

- "TooMuchRequest": No se ejecuta porque el número de peticiones es menor que 5 y el campo id no está vacío.
- "MQTT-Email": No se ejecuta porque el número de peticiones es menor que 5 y el campo id no está vacío.
- "CreateTheRequest": Dado que el campo num es menor que 5, mediante el uso de la función "funDoble" genera la url para posteriormente insertarlo como evento del flujo "requestFhir" con el siguiente contenido: *"url":"http://hapi.fhir.org/baseDstu3/Person/2036660","method": "DELETE","json":"null","messageID": "16709ba2-a2a4-4f98-bb7f-02b5cef19d17"*. Al tener asignado un sumidero de tipo http-request con campos publish.url y method dinámicos realizará una petición del tipo indicado en el campo method, en este caso DELETE y a la url indicada en el campo url, en este caso http://hapi.fhir.org/baseDstu3/Person/2036654. La llegada de un evento del flujo "requestFhir" provoca la realización de las siguientes consultas:
    - "Code20X": Dado que se produce un evento del flujo "responseSteam2xx" recibido mediante la fuente de tipo http-response con la respuesta del servidor FHIR a la petición realizada por el sumidero asignado al flujo "requestFhir" se ejecuta. Recoge la respuesta recibida del evento del flujo "responseStream2xx" junto al messageID del evento del flujo "requestFhir" que provoco esta respuesta y lo inserta en el flujo "reponseStream". El flujo "responseStream" tiene asignado un sumidero de tipo http-response, por lo tanto, cuando se produce un evento de este flujo se responde a la petición que tenía el mismo source.id (Main) y el mismo messageID que este evento (16709ba2-a2a4-4f98-bb7f-02b5cef19d17)
    - "Code40X": No se ejecuta dado que no recibe ningún evento del flujo responseStream4XX.
    - "NoresponseStream": No se ejecuta dado que en menos de 20 segundos desde que se produjo el evento del flujo "requestFhir" se produce un evento del flujo "responseStream".

El evento del flujo "responseStream" ha sido enviado al Proxy como respuesta a la petición que realizó anteriormente. El proxy procesa esta respuesta y elimina campos del JSON innecesarios, el titulo event y el campo messageID, y activa correctamente los caracteres especiales como '\n' y la envía al agente de usuario tal y como podemos ver en la siguiente imagen:

```
pgallegos11@pgallegos11-GE60-2QE:~/Escritorio/json peticiones$ curl -X DELETE http://
localhost/intermediario.php/Person/2036660
{
        "resourceType": "OperationOutcome",
        "text": {
                "status": "generated",
                "div": "<div xmlns="http://www.w3.org/1999/xhtml"><h1>Operation Outco
me</h1><table border="0"><tr><td style="font-weight: bold;">INFORMATION</td><td>[]</t
d><td><pre>Successfully deleted 1 resource(s) in 53ms</pre></td>


                        </tr>
                </table>
        </div>"
        },
        "issue": [
                {
                        "severity": "information",
                        "code": "informational",
                        "diagnostics": "Successfully deleted 1 resource(s) in 53ms"
                }
        ]
}
pgallegos11@pgallegos11-GE60-2QE:~/Escritorio/json peticiones$
```

Ilustración 47: Petición DELETE realizada por el agente de usuario al intermediario y respuesta

Mientras se realizan estas pruebas se puede consultar el log del Stream Processor. Este log nos mostrara el flujo "numRequest" ya que es el que tiene asignada un sumidero de tipo log. Este flujo contiene la información del flujo de entrada junto al número de peticiones realizadas a un mismo recurso, valor que se usará para generar alertas.

```
[2019-08-21 12:44:59,722]  INFO {org.wso2.carbon.stream.processor.core.internal.StreamProcessorService} - Siddh
i App MediaCompra deployed successfully
[2019-08-21 12:44:59,724]  INFO {org.wso2.carbon.stream.processor.core.internal.StreamProcessorService} - Siddh
i App Test deployed successfully
[2019-08-21 12:44:59,727]  INFO {org.wso2.carbon.kernel.internal.CarbonStartupHandler} - WSO2 Stream Processor
started in 13,418 sec
[2019-08-21 12:45:07,277]  INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - Num Request: : Event{timest
amp=1566384307168, data=[Person, 2036654, GET, 1, null, 6aa0d1be-1cea-45e3-af0b-1c8dfdbd5337], isExpired=false}
[2019-08-21 12:46:24,035]  INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - Num Request: : Event{timest
amp=1566384384016, data=[Person, 2036654, PUT, 2, [      {  "resourceType": "Person",  "id": "2036654",  "name"
: [    {      "use": "official",      "text": "Jackie Burton",      "family": "Burton",      "given": [
"Jackie"      ],      "prefix": [        "Ms"      ]    }  ],  "gender": "female"}], 1b36a234-5bb5-48bd-ae2c-0f
27ce92a60a], isExpired=false}
[2019-08-21 12:47:10,467]  INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - Num Request: : Event{timest
amp=1566384430455, data=[Person, , POST, 1, [   {  "resourceType": "Person",  "name": [    {      "use": "offic
ial",      "text": "Jack Burton",      "family": "Burton",      "given": [        "Jack"      ],      "prefix":
 [        "Mr"      ]    }  ],  "gender": "male"}], 16143526-2741-49b1-a95d-38406344983c], isExpired=false}
[2019-08-21 12:48:01,690]  INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - Num Request: : Event{timest
amp=1566384481682, data=[Person, 2036660, GET, 1, null, b1beda63-6019-4a3c-86e7-1d88b99ee6fe], isExpired=false}
[2019-08-21 12:48:42,614]  INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - Num Request: : Event{timest
amp=1566384522571, data=[Person, 2036660, DELETE, 2, null, 9d336174-b7bb-40a2-8438-c6e82dce18a3], isExpired=fal
se}
```

Ilustración 48: Log Stream Processor-numRequest

3.4.2.2 Probando demasiadas peticiones

Como se indicó anteriormente, si se producen cinco peticiones al mismo recurso se genera una situación de alerta. Para comprobar que esto funciona correctamente se han realizado previamente cuatro peticiones GET al mismo recurso, esto hace que el ultimo evento del flujo "numRequest" tenga el valor 4 en el campo num.

Posteriormente se realiza la quinta petición. Para ello en la URI se debe incluir el tipo de recurso y el identificador de este, como si se realizara al servidor FHIR directamente. Esta petición se realiza mediante el comando curl a la url http://localhost/intermediario.php/Person/2036654.

El proxy recibe esta petición y genera una petición POST a la url http://localhost:5005/FHIR del Stream Processor con el siguiente JSON:

```
{
      "event":
      {
            "resource": "Person",
            "id": "2036654",
            "method": "GET",
            "json": "null"
      }
}
```

Esta petición es recibida como un evento del flujo "requestStream". El contenido es *"messageID": "45dfd8bc-9c9b-45f0-b329-5aa750e02121","resource":"Person","id:"2036654","method":"GET", "json":"null"*.La llegada de un evento del flujo "requestStream" activa la realización de la consulta "SameResourceAndIDCount", consulta que únicamente añade un campo que contabiliza el número de peticiones recibidas a este recurso quedando con el siguiente contenido: *"messageID": "45dfd8bc-9c9b-45f0-b329-5aa750e02121", "resource": "Person", "id": "2036654","method":"GET", "num": "5", "json": "null"* y lo inserta en el flujo "numRequest" que lo muestra por el terminal ya que tiene asignado un sumidero de tipo log. La llegada de un evento del flujo "numRquest" provoca las siguientes consultas:

- "TooMuchRequest": Se ejecuta porque el número de peticiones es igual a 5. Genera una respuesta de error mediante el uso de convert "Peticion bloqueda, demasiadas peticiones al recurso" y junto al messageID inserta un evento en el flujo "reponseStream". El flujo "responseStream" tiene asignado un sumidero de tipo http-response, por lo tanto, cuando se produce un evento de este flujo se responde a la petición que tenía el mismo source.id (Main) y el mismo messageID que este evento (45dfd8bc-9c9b-45f0-b329-5aa750e02121)
- "MQTT-Email": Se ejecuta porque el número de peticiones es igual a 5. Inserta todo el contenido del evento en un evento del flujo alertaColaEmail. Este flujo tiene asignado dos sumideros, el primero de tipo MQTT, lo cual hace que se envié un mensaje a la cola MQTT asignada y otro de tipo email, lo que provoca que se envié un email al correo indicado.
- "CreateTheRequest": No se ejecuta porque el número de peticiones es igual o mayor que 5.

El evento del flujo "responseStream" ha sido enviado al Proxy como respuesta a la petición que realizó anteriormente. El proxy procesa esta respuesta y elimina campos del JSON innecesarios, el titulo event y el campo messageID, y la envía al agente de usuario tal y como podemos ver en la siguiente imagen:

```
pgallegos11@pgallegos11-GE60-2QE:~/Escritorio/json peticiones$ curl http://localhost/
intermediario.php/Person/2036654
Error: Peticion bloqueada, demasiadas peticiones al recurso
pgallegos11@pgallegos11-GE60-2QE:~/Escritorio/json peticiones$ curl http://localhost/
intermediario.php/Person/2036654
Error: Peticion bloqueada, demasiadas peticiones al recurso
```

Ilustración 49: Petición del agente de usuario al intermediario cuando se producen demasiadas peticiones y respuesta

Como se ha comentado anteriormente las consecuencias de sobrepasar el número de peticiones máximas a un determinado recurso son la inclusión de esta información en una cola MQTT y el envío de un email al administrador:

```
pgallegos11@pgallegos11-GE60-2QE:~$ sudo service mosquitto status
● mosquitto.service - LSB: mosquitto MQTT v3.1 message broker
   Loaded: loaded (/etc/init.d/mosquitto; bad; vendor preset: enabled)
   Active: active (running) since mié 2019-08-21 11:15:49 CEST; 1h 34min ago
     Docs: man:systemd-sysv-generator(8)
  Process: 865 ExecStart=/etc/init.d/mosquitto start (code=exited, status=0/SUCC
   CGroup: /system.slice/mosquitto.service
           └─1028 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

ago 21 11:15:46 pgallegos11-GE60-2QE systemd[1]: Starting LSB: mosquitto MQTT v3
ago 21 11:15:49 pgallegos11-GE60-2QE mosquitto[865]:  * Starting network daemon:
ago 21 11:15:49 pgallegos11-GE60-2QE mosquitto[865]:    ...done.
ago 21 11:15:49 pgallegos11-GE60-2QE systemd[1]: Started LSB: mosquitto MQTT v3.
pgallegos11@pgallegos11-GE60-2QE:~$ mosquitto_sub -h localhost -t "Alerta" -v
Alerta {"event":{"resource":"Person","id":"2036654","num":5}}
Alerta {"event":{"resource":"Person","id":"2036654","num":6}}
```

Ilustración 50: Mensaje recibido por el agente de administrador tras suscribirse al topic “Alerta” mediante mosquitto_sub de la cola MQTT

Ilustración 51: Email recibido por el agente de administrador

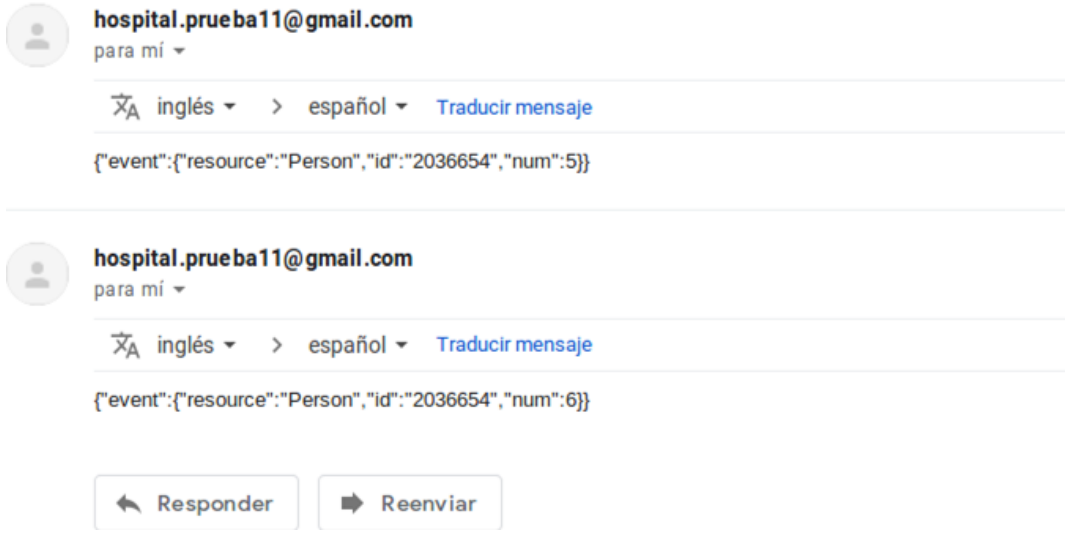


Ilustración 52: Detalle email recibido por el agente de administrador

3.4.2.3 Probando otro recurso: Patient

Este escenario funciona perfectamente para cualquier otro recurso con el que trabaje el servidor FHIR. A modo de ejemplo, se han realizado pruebas similares a las anteriores, pero con el recurso Patient. Para ello únicamente se debe modificar el JSON y la URI. Únicamente se van a incluir las peticiones del agente de usuario y la respuesta que reciben. Ppara más detalle del flujo intermedio consultar el apartado anterior (3.4.2.1).

- POST

El único cambio que se debe realizar frente al ejemplo anterior es adjuntar un JSON con el contenido adecuado para el recurso Patient y modificar la URI.

```
pgallegos11@pgallegos11-GE60-2QE:~/Escritorio/json peticiones$ curl -X POST http://lo
calhost/intermediario.php/Patient -H 'content-type:application/json' -d @newPatient.j
son
{
        "resourceType": "Patient",
        "id": "2036677",
        "meta": {
                "versionId": "1",
                "lastUpdated": "2019-08-21T11:12:27.151+00:00"
        },
        "text": {
                "status": "generated",
                "div": "<div xmlns="http://www.w3.org/1999/xhtml"> </div>"
        },
        "identifier": [
                {
                        "use": "usual",
                        "value": "100065799",
                        "assigner": {
                                "display": "AccMgr"
                        }
                }
        ],
        "active": false,
        "gender": "male",
        "birthDate": "1993-10-24",
        "deceasedBoolean": false
}
```

Ilustración 53: Petición POST patient realizada por el agente de usuario al intermediario y respuesta

- GET

Para realizar esta prueba únicamente se indica el recurso y el identificador en la URI, para que coincida con el recurso del cual se quiere obtener información:

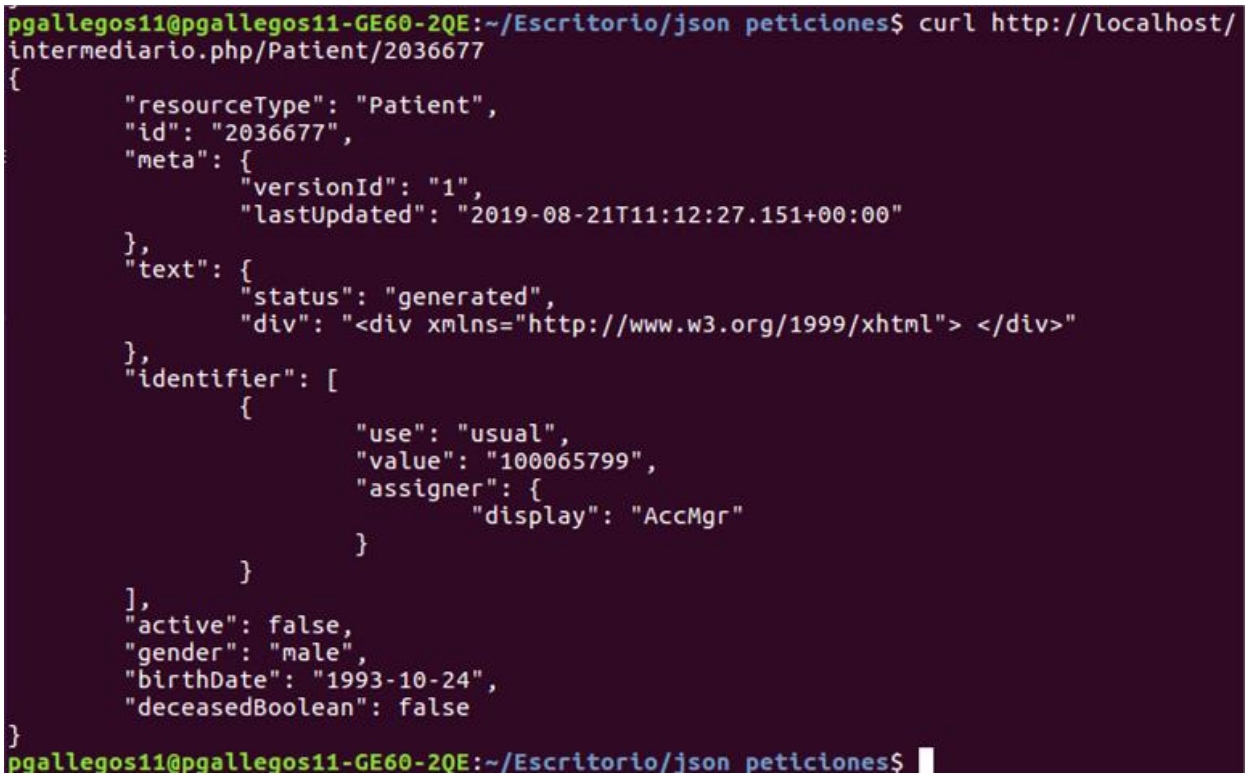


Ilustración 54: Petición GET patient realizada por el agente de usuario al intermediario y respuesta

- PUT

Para realizar esta prueba se modifica el tipo recurso, el identificador y el JSON para que coincida con el formato de los recursos tipo Patient.

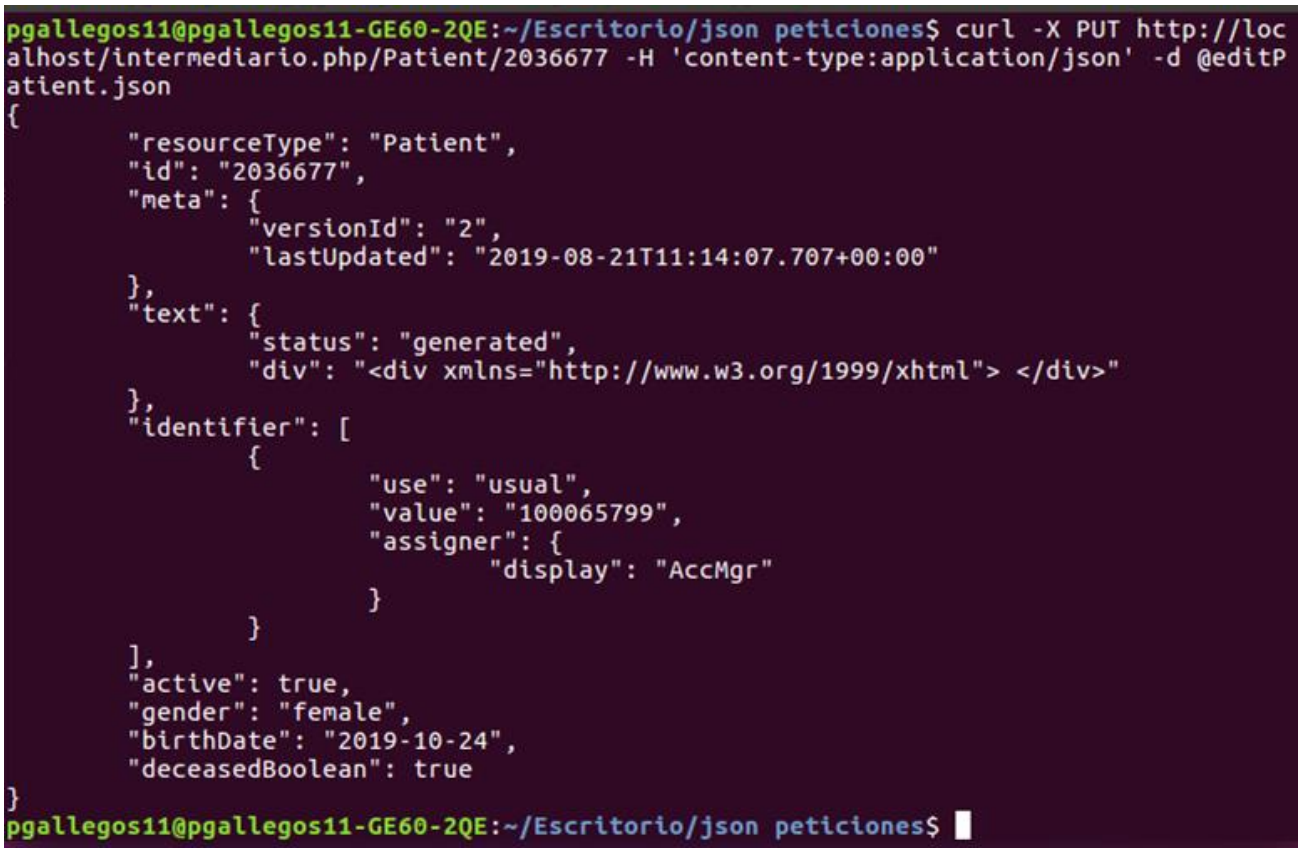


Ilustración 55: Petición PUT patient realizada por el agente de usuario al intermediario y respuesta

- DELETE

Para realizar esta prueba únicamente se modifica el tipo recurso y el identificador para que coincida con el recurso que se desea eliminar:

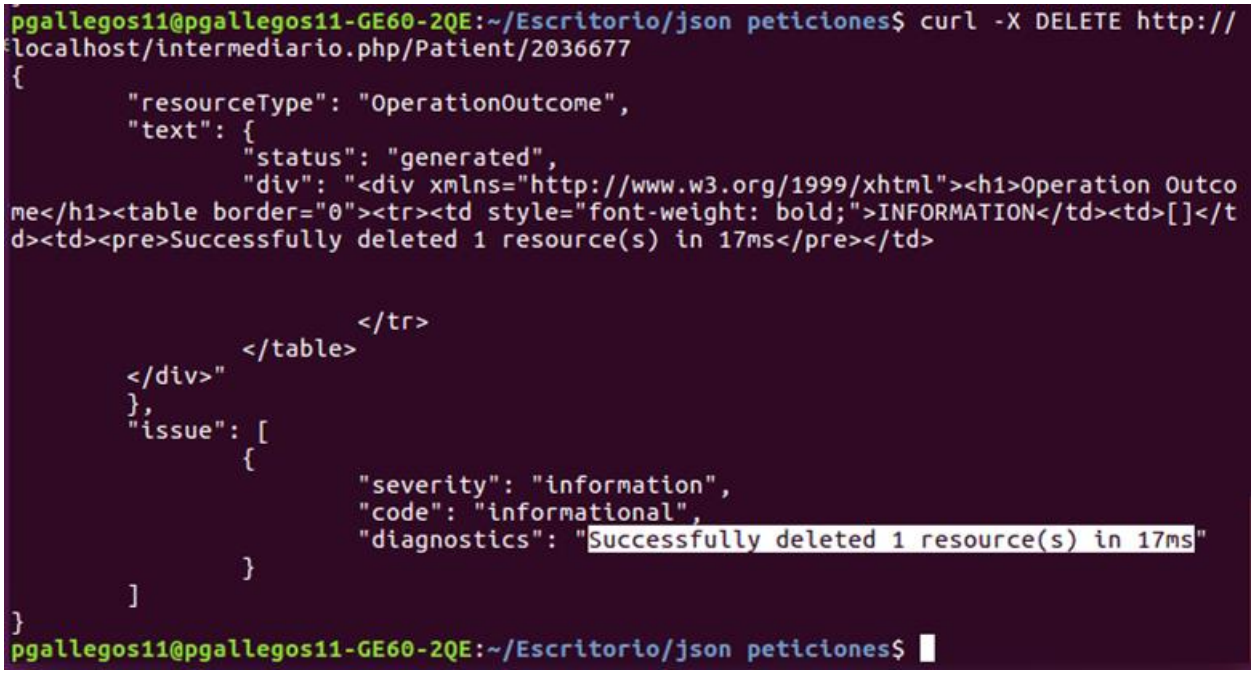


Ilustración 56: Petición DELETE patient realizada por el agente de usuario al intermediario y respuesta

- Demasiadas peticiones

Por último, se puede comprobar que al realizar demasiadas peticiones ocurre lo mismo que para el recurso Person, con la única diferencia de que en el correo y en la cola MQTT se indica que ahora el recurso afectado es de tipo Patient:

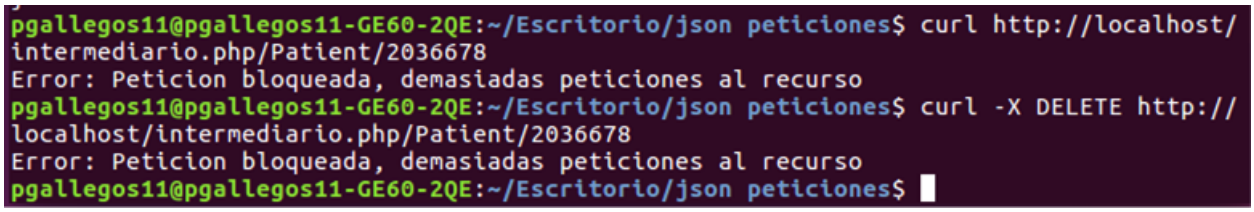


Ilustración 57: Demasiadas peticiones a recurso Patient realizadas por el agente de usuario al intermediario

Ilustración 58: Email recibido por el agente de administrador tras consultas a recurso Patient

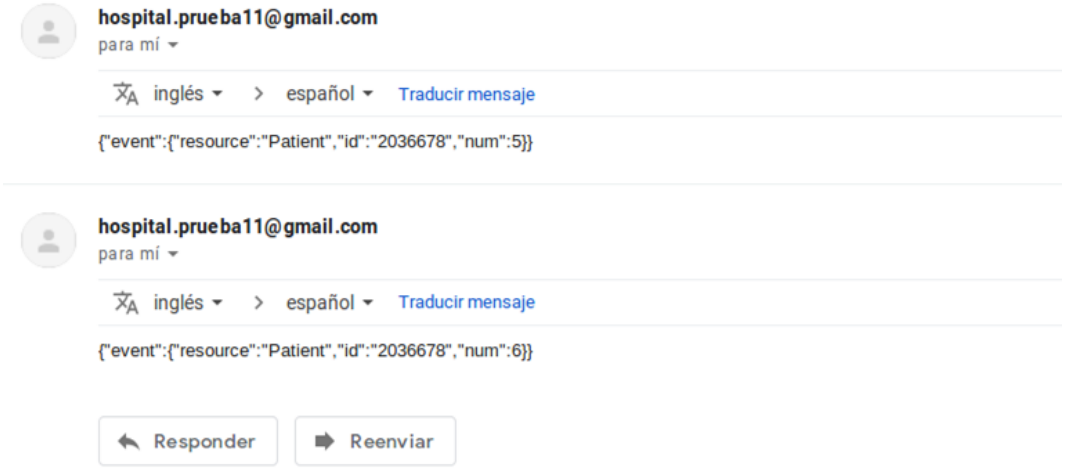


Ilustración 59: Detalle email recibido por el agente de administrador tras consultas a recurso Patient

```
pgallegos11@pgallegos11-GE60-2QE:~$ sudo service mosquitto status
● mosquitto.service - LSB: mosquitto MQTT v3.1 message broker
   Loaded: loaded (/etc/init.d/mosquitto; bad; vendor preset: enabled)
   Active: active (running) since mié 2019-08-21 11:15:49 CEST; 1h 34min ago
     Docs: man:systemd-sysv-generator(8)
  Process: 865 ExecStart=/etc/init.d/mosquitto start (code=exited, status=0/SUCC
   CGroup: /system.slice/mosquitto.service
           └─1028 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

ago 21 11:15:46 pgallegos11-GE60-2QE systemd[1]: Starting LSB: mosquitto MQTT v3
ago 21 11:15:49 pgallegos11-GE60-2QE mosquitto[865]:  * Starting network daemon:
ago 21 11:15:49 pgallegos11-GE60-2QE mosquitto[865]:    ...done.
ago 21 11:15:49 pgallegos11-GE60-2QE systemd[1]: Started LSB: mosquitto MQTT v3.
pgallegos11@pgallegos11-GE60-2QE:~$ mosquitto_sub -h localhost -t "Alerta" -v
Alerta {"event":{"resource":"Person","id":"2036654","num":5}}
Alerta {"event":{"resource":"Person","id":"2036654","num":6}}
Alerta {"event":{"resource":"Patient","id":"2036678","num":5}}
Alerta {"event":{"resource":"Patient","id":"2036678","num":6}}
```

Ilustración 60: Mensaje recibido por el agente de administrador tras suscribirse al topic “Alerta” de la cola MQTT tras peticiones a recurso Person y Patient

3.4.2.4 Probando flujos de error

En todas las pruebas anteriores se han probado casos en los que todo funcionaba correctamente y no existía ningún problema, no obstante, tanto el servidor FHIR como el resto de los elementos están preparado para controlar los posibles errores que pueden producirse. El Stream Processor trata todos los mensajes, se produzcan o no algunos de estos errores, de la misma forma. En caso de que se quiera detallar en el Stream Processor, se podrían añadir nuevos sumideros de tipo log o nuevas consultas para procesar estas respuestas. El Proxy trata las respuestas de manera similar, pero en el caso de que no se produzca un error grave, y no incluya el campo "resourceType" adapta el procesado del texto.

Los principales errores que pueden producirse vienen derivados de la inclusión de parámetros incorrectos en la petición inicial:

- Error en el recurso: Al introducir en la URI un tipo de recurso que no coincide con ningún recurso FHIR.

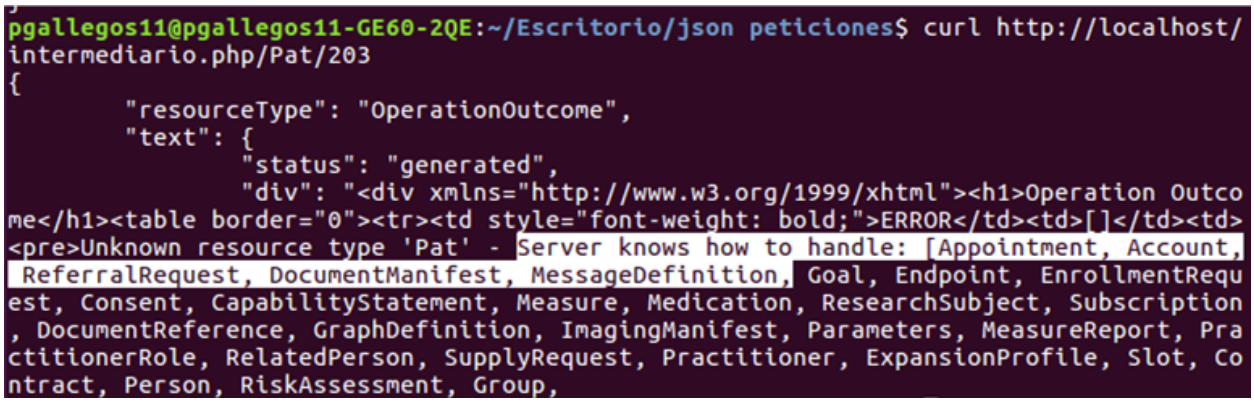


Ilustración 61: Petición del agente de usuario con error de recurso inexistente y respuesta

Se ha introducido un recurso llamado 'Pat' y en la respuesta se indica que este tipo de recurso no existe.

- Error en el id: Al introducir como id una cadena que no coincide con ningún identificador válido para el tipo de recurso solicitado.

Ilustración 62: Petición del agente de usuario con error de id inexistente y respuesta

Se ha introducido en el campo id el valor ‘203’, valor que no coincide con ningún paciente, como puede verse en la respuesta se indica que el identificador no existe para el tipo de recurso solicitado.

- Error en el JSON: Al introducir como JSON un archivo que no coincide con el formato JSON que el servidor espera recibir para el recurso especificado en la URI:

```
pgallegos11@pgallegos11-GE60-2QE:~/Escritorio/json peticiones$ curl -X POST http://lo
calhost/intermediario.php/Patient -H 'content-type:application/json' -d @newperson.js
on
{
        "resourceType": "OperationOutcome",
        "text": {
                "status": "generated",
                "div": "<div xmlns="http://www.w3.org/1999/xhtml"><h1>Operation Outco
me</h1><table border="0"><tr><td style="font-weight: bold;">ERROR</td><td>[]</td><td>
<pre>Failed to parse request body as JSON resource. Error was: Incorrect resource typ
e found, expected "Patient" but found "Person"</pre></td>


                        </tr>
                </table>
        </div>"
        },
        "issue": [
                {
                        "severity": "error",
                        "code": "processing",
                        "diagnostics": "Failed to parse request body as JSON resource
. Error was: Incorrect resource type found, expected "Patient" but found "Person""
                }
        ]
}
pgallegos11@pgallegos11-GE60-2QE:~/Escritorio/json peticiones$ curl -X POST http://lo
calhost/intermediario.php/Patient -H 'content-type:application/json' -d @error.json
{
        "resourceType": "OperationOutcome",
        "text": {
                "status": "generated",
                "div": "<div xmlns="http://www.w3.org/1999/xhtml"><h1>Operation Outco
me</h1><table border="0"><tr><td style="font-weight: bold;">ERROR</td><td>[]</td><td>
<pre>Failed to parse request body as JSON resource. Error was: Invalid JSON content d
etected, missing required element: 'resourceType'</pre></td>


                        </tr>
                </table>
        </div>"
        },
        "issue": [
                {
                        "severity": "error",
                        "code": "processing",
                        "diagnostics": "Failed to parse request body as JSON resource
. Error was: Invalid JSON content detected, missing required element: 'resourceType'"
                }
        ]
}
```

Ilustración 63: Peticiones del agente de usuario con error de JSON no valido y respuestas

Se han realizado dos pruebas, la primera con un recurso incorrecto para el JSON especificado donde se indica este error y la segunda donde falta el campo obligatorio resourceType.

4 CONCLUSIONES

La realización de este proyecto ha supuesto un gran reto al tener que realizar un estudio a fondo de una herramienta sobre la cual la información pública disponible es muy básica. Además, durante el desarrollo de este proyecto los componentes de WSO2 han sufrido diversos cambios que han obligado a modificar parte del contenido de éste. Algunos cambios, como la aparición de patrones, han facilitado su desarrollo y otros, como la desaparición del DAS lo complicaron.

A modo de conclusión del proyecto se hará un repaso de los objetivos planteados al principio y se mostrarán diferentes caminos por los que se podría continuar trabajando con el Stream Processor.

4.1 Objetivos logrados

Se puede decir que los principales objetivos que se plantearon al inicio del proyecto se han alcanzado satisfactoriamente, ya que:

- Se ha desarrollado un sistema que permite interceptar peticiones a un servidor FHIR con el fin de procesar tanto consultas como respuestas para obtener información de éstas y actuar en consecuencia.
- El procesado de este flujo provoca salidas tanto por email como por una tecnología que actualmente está en auge como son las colas MQTT.
- Esta memoria puede servir de ejemplo para facilitar que nuevos desarrolladores puedan comprender el funcionamiento del Stream Processor y realizar diferentes proyectos con el mismo, motivo por el cual se han incluido códigos de pruebas utilizando algunas funcionalidades no utilizadas en el trabajo principal. Estas funcionalidades son útiles, pero no tenían cabida en el escenario real que estábamos trabajando

4.2 Líneas de continuación

Existen diferentes líneas que podrían abarcarse en nuevos trabajos:

- Aumentar el número de análisis realizados con el fin de obtener más información, aunque para ello quizás se deban personalizar más las fuentes de eventos en lugar de usar una general como se hace actualmente.
- Añadir alguna funcionalidad más del Stream Processor.
- Incluir un parser externo que modifique el contenido de los emails de tal forma que tengan un formato más atractivo para el usuario.
- Utilizar Kafka como intermediario, tal y como recomiendan desde la página oficial de WSO2, para los eventos de los diferentes flujos de entrada para evitar que se colapse si superan los 100k eventos por segundo.
- WSO2 posee diferentes componentes que pueden interactuar fácilmente con el objetivo de dar una solución global. Una continuación lógica del proyecto es interconectarla con otros componentes. Para ello se podría hacer uso del WSO2 Enterprise Integrator. Una vez integrado con el Entreprise Integrator se podría integrar con el Identity Server que cubriría la seguridad, dejada en un segundo

plano en este proyecto.

- Como se ha dicho anteriormente este proyecto busca servir de guía para nuevos proyectos por lo que puede servir de punto de partida para cualquier otro sistema de análisis de flujo, en cualquier entorno distinto al sanitario.

5 BIBLIOGRAFÍA

[1] B. D. Sanidad. [En línea]. Available: http://gacetasanitaria.org/es-big-data-sanidad-espana-oportunidad-articulo-S0213911115002095.

[2] BigMedilytics. [En línea]. Available: https://www.bigmedilytics.eu/.

[3] H. 2020. [En línea]. Available: https://ec.europa.eu/eurostat/cros/content/big-data-and-data-science_en.

[4] Savana. [En línea]. Available: https://www.savanamed.com/.

[5] G. S. España. [En línea]. Available: http://gacetasanitaria.org/es-big-data-sanidad-espana-oportunidad-articulo-S0213911115002095.

[6] S. P. WSO2. [En línea]. Available: https://wso2.com/analytics-and-stream-processing/.

[7] Siddhi. [En línea]. Available: https://siddhi.io/en/v5.0/.

[8] Gartner, «Big data,» [En línea]. Available: https://www.gartner.com/it-glossary/big-data/.

[9] Oracle, «Big data,» [En línea]. Available: https://www.oracle.com/es/big-data/guide/what-is-big-data.html.

[10] Wikipedia, «Big Data,» [En línea]. Available: https://es.wikipedia.org/wiki/Macrodatos#Caracter%C3%ADsticas.

[11] WSO2. [En línea]. Available: https://wso2.com.

[12] REST. [En línea]. Available: https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional.

[13] R. Standar. [En línea]. Available: https://standards.rest/.

[14] FHIR. [En línea]. Available: https://www.hl7.org/fhir/.

[15] FHIR-Docs. [En línea]. Available: http://wiki.hl7.org/index.php?title=FHIR.

[16] MQTT. [En línea]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[17] MQTT. [En línea]. Available: http://mqtt.org/.

[18] MQTT. [En línea]. Available: https://en.wikipedia.org/wiki/MQTT.

[19] Mosquitto. [En línea]. Available: https://mosquitto.org/.

[20] M. Siddhi. [En línea]. Available: https://siddhi-io.github.io/siddhi/documentation/siddhi-5.x/query-guide-5.x/ .

[21] WSO2-Quick-Start. [En línea]. Available: https://docs.wso2.com/display/SP400/Quick+Start+Guide.

[22] D. SP. [En línea]. Available: https://wso2.com/analytics-and-stream-processing/.

ANEXO A: CÓDIGO ESCENARIO REAL

En este anexo se va a incluir el código del escenario real, el código del proxy intermediario y los diferentes JSON que se han usado para las pruebas.

A.1 Código escenario real

```
@App:name("Fhir-Final")
@App:description("Analisis y procesado de las peticiones al servidor
FHIR")
/*
* Flujo con una fuente de tipo http-request que recibira las
peticiones con body json a la url
* http://localhost:5005/FHIR. El json recibido tendra cuatro campos
resource, id, method y string.
* Para poder responder se asigna un messageID automatico y el nombre
de fuente Main.
*/
@source(type="http-request",
receiver.url='http://localhost:5005/FHIR',
source.id="Main",
@map(type='json',@attributes(messageID='trp:messageID',
resource='$.event.resource',                    method='$.event.method',
id='$.event.id',json='$.event.json')))
define stream requestStream (messageID string,resource string,id
string, method string, json string);
/*
* Flujo con un sumidero de tipo http-response que respondera las
peticiones con un json gracias al
* messageID y al nombre de fuente Main
*/
@sink(type='http-response',                              source.id='Main',
message.id='{{messageID}}',
headers="'content-type:json'",@map(type='json'))
define stream responseStream (messageID string,message string);

/*
* Flujo con un sumidero de tipo http-request que realiza las
peticiones al servidor FHIR
* Para la peticion se utiliza la url, el metodo y el json que se
incluye
*/
@sink(type='http-
request',publisher.url='{{url}}',method='{{method}}',sink.id='Fhir',
headers="'Content-type:application/json','User-Agent:Stream
Procesor','Host:hapi.fhir.org','Accept:        */*'",@map(type='json',
@payload("{{json}}") ) )
define    stream    requestFhir    (url    string,method    string,json
string,messageID string);
/*
* Flujo con una fuente de tipo http-response que recibe las respuestas
de las peticiones
```

```
* que se realicen al servidor FHIR siempre y cuando el codigo http es
2XX
*/
@source(type='http-response', sink.id='Fhir', http.status.code='2\d+',
@map(type='text',
regex.A='((.|\n)*)', @attributes(message='A[1]')))
define stream responseStream2xx(message string);
/*
* Flujo con una fuente de tipo http-response que recibe las respuestas
de las peticiones
* que se realicen al servidor FHIR siempre y cuando el codigo http es
4XX
* en este caso es unicamente a modo demostrativo y se tratara de igual
forma que si es 2XX
*/
@source(type='http-response', sink.id='Fhir', http.status.code='4\d+',
@map(type='text',
regex.A='((.|\n)*)', @attributes(message='A[1]')))
define stream responseStream4xx(message string);

/*
* Flujo con una fuente de tipo log
*  Calcula el numero de peticiones que se han realizado a un
determinado recurso
*/
@sink(type = 'log', prefix = 'Num Request:')
define stream numRequest (resource string,id string,method string, num
long, json string, messageID string);
/*
* Flujo con un sumidero o sink asignado de tipo mqtt y tipo email,
*  se enviaran con el asunto Alerta de WSO2 Stream Processor-
NombreRecuro al email pgallegosj11@gmail.com
* Se inclura en la cola con el topic 'Alerta'.
*  El resto de parametros de la sink mqtt son los por defecto para
poder explicarlos con mayor claridad.
*  En ambos caso se incluira tanto el tipo de recurso como el
identificador
*/
@sink(type='email', @map(type ='json'), username='hospital.prueba11',
address='hospital.prueba11@gmail.com',
password='l#QT9iSuasuT',host='smtp.gmail.com',port='465',ssl.enable='t
rue',auth='true',
content.type='text/html',subject='Alerta de Wso2 Stream Processor-
{{resource}}',to='pgallegosj11@gmail.com')
@sink(type='mqtt', url= 'tcp://localhost:1883', topic='Alerta',
clean.session='true', message.retain='false',
quality.of.service= '1', keep.alive= '60',connection.timeout='30',
@map(type='json'))
define stream alertaColaEmail(resource string, id string, num long);
/*
* Dado que solo se puede incluir un script esta funcion realiza dos
tareas:
* 1 Correccion de formato: Adaptar el json al formato correcto
* 2 CreateUrl: Construir la url para el servidor fhir
*/
define function funDoble[JavaScript] return string {
```

```
    var response = "";
    var  resourceType = data[0];
    if (resourceType.charAt(0).equals('[')) {
        //Correccion de formato
        response  =  resourceType.substring(1,  resourceType.length()  -
1);
    } else if(!resourceType.equals("null")) {
        //CreateUrl
        response = "http://hapi.fhir.org/baseDstu3/" + resourceType +
"/" + data[1];
    }
return response;
};

/*
* Consulta que contabiliza cuantas peticiones de las ultimas 10 se
realizan al cada id y recurso
* y lo inserta en el flujo numRequest
*/
@info(name = 'SameResourceAndIDCount')
from requestStream#window.time(10 min)
select resource,id,method,  count() as num,json,messageID
     group by  resource, id
insert into numRequest;

/*
* Consulta que en funcion del numero de peticiones, obviando el id
vacio que es para
* la creacion de un nuevo id, genera un evento de tipo alertaColaEmail
que generara
* un email y un mensaje a una cola MQTT
*/
@info(name = 'MQTT-Email')
from numRequest[num >= 5 and id!=""]
select  resource, id,num
insert into alertaColaEmail;
/*
* Consulta que si no se sobrepasan las 3 peticiones al recurso-id,
genera un evento del flujo
* requestFhir que se enviara al servidor Fhir
*/
@info(name='CreatetheRequest')
from numRequest [id=="" or num < 5]
select funDoble(resource,id) as url, method as method,funDoble(json)
as json, messageID
insert into requestFhir;
/*
* Consulta que si se sobrepasan las 3 peticiones al recurso-id, genera
un mensaje de alerta
* indicando que se han realizado demsaiadas peticiones al recurso
*/
@info(name='TooMuchRequest')
from numRequest[num >= 5 and id!=""]
select  messageID,  convert("Error:  Peticion  bloqueada,  demasiadas
peticiones al recurso",'string') as message
insert into responseStream;
/*
```

```
* Consulta que procesa las respuestas con el codigo 2XX y las inserta
en el flujo
* responseStream que respondera a la peticion que recibio el stream
processor
*/
@info(name='Code20X')
from every  e1=requestFhir  ->  e2=responseStream2xx
select e1.messageID  as messageID, e2.message as message
insert into responseStream;
/*
* Consulta que procesa las respuestas con el codigo 4XX y las inserta
en el flujo
* responseStream que respondera a la peticion que recibio el stream
processor
*/
@info(name='Code40X')
from every e1=requestFhir  ->  e2=responseStream4xx
select e1.messageID as messageID, e2.message as message
insert into responseStream;
/*
* Consulta que en caso de realizar una peticion de requestFhir y no se
reciba una respuesta en 20 segundos
* genera una mensaje de error indicando que no se ha podido contactar
con el servidor
*/
@info(name='NoresponseStream')
from every   e1=requestFhir -> not responseStream  for 20 sec
select   e1.messageID, convert("No se ha podido contactar con el
servidor, revise si su peticion es correcta",'string') as message
insert into responseStream;
```

A.2 Intermediario.php

```
<?php
//Funcion write_log: Escribira en su archivo de log el json y la fecha
function write_log($cadena,$tipo)
{
	$arch = fopen("logs/intermediario.txt", "a+");

	fwrite($arch, "[".date("Y-m-d H:i:s.u")." "." "." - $tipo ]
".$cadena."\n");
	fclose($arch);
}
//URL Stream Processor
$url = 'http://localhost:5005/FHIR';
//Creacion de recurso cURL
$ch = curl_init($url);
//Creacion de json en funcion de los parametros aceptados por Stream
Processor
$url_completa = $_SERVER['REQUEST_URI'];
$method = $_SERVER['REQUEST_METHOD'];
$partes = explode("/", $url_completa);
if (strcmp($method,"GET") == 0 || strcmp($method,"DELETE")==0){
$json='null';
} else if(strcmp($method,"POST")==0){
$json='['.file_get_contents('php://input').']';
$partes[3]="";
} else{
$json='['.file_get_contents('php://input').']';
}
if($partes[2]==null)
$partes[2]="";
if($partes[3]==null)
$partes[3]="";
$data = array(
	'resource' => $partes[2],
	'id' => $partes[3],
	'method' => $method,
	'json' => $json
);

$payload = json_encode(array("event" => $data));
//Escribimos en el log el json
write_log($payload,"INFO-JSON");
//Añadir el json a la peticion
curl_setopt($ch, CURLOPT_POSTFIELDS, $payload);
//Añadir las diferentes opciones
curl_setopt($ch, CURLOPT_HTTPHEADER, array('Content-
Type:application/json'));
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
//Ejecutamos la peticion
$result = curl_exec($ch);
//Procesado de la respuesta para que se muestre de la forma mas
similar a si
//se realiza la peticion directamente al servidor FHIR
$str= str_replace("\\n","\n",$result);
$str= str_replace("\\r","\r",$str);
```

```
$str= str_replace("\\t","\t",$str);
$str= str_replace("  ","\t",$str);
$str= str_replace("\\","",$str);
$str=substr($str,0,-3);
$pos = strpos($str, "{\n\t\"resourceType");
if($pos ===false)
{
//Procesado de la respuesta personalizada de error
$pos = strpos($str, "Error: ");
}
$str=substr($str,$pos);
echo $str."\n";
//Cerrado el recurso cURL
curl_close($ch);
?>
```

A.3 JSON

A.3.1 newperson.json

```
{
  "resourceType": "Person",
  "name": [
    {
      "use": "official",
      "text": "Jack Burton",
      "family": "Burton",
      "given": ["Jack" ],
      "prefix": [
        "Mr"
      ]
    }
  ],
  "gender": "male"
}
```

A.3.2 editperson.json

```
{
  "resourceType": "Person",
  "id": "2036654",
  "name": [
    {
      "use": "official",
      "text": "Jackie Burton",
      "family": "Burton",
      "given": [ "Jackie" ],
      "prefix": [
        "Ms"
      ]
    }
  ],
  "gender": "female"
}
```

A.3.3 newPatient.json

```
{
     "resourceType": "Patient",
     "text": {
          "status": "generated",
          "div": "<div xmlns=\"http://www.w3.org/1999/xhtml\"> </div>"
          },
     "identifier": [            {
               "use": "usual",
               "value": "100065799",
               "assigner": { "display": "AccMgr"}
               }          ],
     "active": false,
     "gender": "male",
     "birthDate": "1993-10-24",
     "deceasedBoolean": false
}
```

A.3.4 editPatient.json

```
{
     "resourceType": "Patient",
     "id":"2036677",
     "text": {
          "status": "generated",
          "div": "<div xmlns=\"http://www.w3.org/1999/xhtml\"> </div>"
          },
     "identifier": [            {
               "use": "usual",
               "value": "100065799",
               "assigner": { "display": "AccMgr"      }
               }          ],
     "active": true,
     "gender": "female",
     "birthDate": "2019-10-24",
     "deceasedBoolean": true
}
```

ANEXO B: CÓDIGO ESCENARIO PRUEBAS

En este anexo se va a incluir el código de los diferentes escenarios de prueba

B.1 Primer escenario de pruebas: MediaCompra

```
@App:name('MediaCompra')
@App:description('Escenario que calcula la media de los 3 ultimos
pedidos')

-- Flujo basico con dos atributos
define stream compra (comprador string,consumo double);

/*
* Flujo con un sumidero o sink asignado de tipo log por lo que se
mostrara por el terminal
* Todos los mensajes que se muestren comenzaran por el prefijo:
Ejemplo media consumo:
*/
@sink(type = 'log', prefix = 'Ejemplo media consumo')
define stream mediaConsumo (media double);

/*
* Realiza la media de lo que han consumido las ultimas 3 compras
* siempre y cuando el importe sea mayor o igual a 16 y menos de 100
*/
@info(name= 'consultaMedia')
from compra[(consumo >= 16 and consumo < 100)]#window.length(3)
select avg(consumo) as media
insert into mediaConsumo;
```

B.2 Segundo escenario de prueba: RegistroCompra

```
@App:name('RegistroCompra')
@App:description('Escenario que controla las compras y registra el
consumo por persona')
/*
* Flujo con una fuente de tipo http-request que recibira las
peticiones con body xml a la url
* http://localhost:5005/Compra. El xml recibido tendra dos campos
comprador y consumo.
* Para poder responder se asigna un messageID automatico y el nombre
de fuente Recibido
*/
@source(type="http-request",
receiver.url='http://localhost:5005/Compra', source.id="Recibido",
@map(type='xml',@attributes(messageID='trp:messageID',
comprador='comprador', consumo='consumo')))
define stream compra (messageID string,comprador string, consumo
double);
/*
* Flujo con un sumidero de tipo http-response que respondera las
peticiones con un xml gracias al
* messageID y al nombre de fuente Recibido
*/
@sink(type='http-response', source.id='Recibido',
message.id='{{messageID}}',
headers="'content-type:xml'",@map(type='xml'))
define stream respuestaCompra (messageID string,mensaje string);
/*
* Flujo con un sumidero o sink asignado de tipo log por lo que se
mostrara por el terminal
* Todos los mensajes que se muestren comenzaran por el prefijo:
Consumo total por comprador:
*/
@sink(type = 'log', prefix = 'Consumo total por comprador')
define stream consumoPorPersona (consumo double,comprador string,
numCompras long);
/*
* Consulta que cada dos minutos calcula el consumo por persona y el
numero de compras
* Siempre y cuando el consumo total sea menos de 100 y enviara los 3
con mayor consumo
* a consumoPorPersona
*/
@info(name = 'ConsumoPorPersona')
from compra#window.timeBatch(2 min)
select  sum(consumo) as consumo,comprador as comprador , count() as
numCompras
group by comprador
having consumo < 100
order by consumo desc
limit 3
insert into consumoPorPersona;
--Consulta que genera un mensaje agradeciendo la compra por cada
evento de compra
@info(name='Respuesta')
```

```
from compra
select messageID as messageID, convert("Muchas gracias por su
compra",'string') as mensaje
insert into respuestaCompra;
```

B.3 Tercer escenario de prueba: DescuentosTuTienda

```
@App:name('DescuentosTuTienda')
@App:description('Escenario que ofrece descuentos mediante email y
terminal')
-- Flujo basico con dos atributos
define stream compra (comprador string, consumo double);
/*
* Flujo con un sumidero o sink asignado de tipo log por lo que se
mostrara por el terminal
* Todos los mensajes que se muestren comenzaran por el prefijo:
Respuesta:
*/
@sink(type = 'log', prefix = 'Respuesta')
define stream respuestaCompra (mensaje string);
/*
* Flujo con un sumidero o sink asignado de tipo log por lo que se
mostrara por el terminal
* Todos los mensajes que se muestren comenzaran por el prefijo: Compra
envio log:
*/
@sink(type = 'log', prefix = 'Compra envio log')
define stream compraProcesado(consumo double,comprador string,
numCompras long);
/*
* Flujo con un sumidero o sink asignado de tipo log por lo que se
mostrara por el terminal
* Todos los mensajes que se muestren comenzaran por el prefijo: Compra
envio log:
*/
@sink(type = 'log', prefix = 'Compra con Descuento envio log')
define stream compraDesc(comprador string, descuento bool, porcentaje
double);
/*
* Flujo con un sumidero o sink asignado de email, se enviaran con el
asunto
* Recordatorio de descuentos-NombreComprador al email
pgallegosj11@gmail.com
*/
@sink(type='email', @map(type ='xml'), username='descuentos.tutienda',
address='descuentos.tutienda@gmail.com',password='7ti\a2@j',host='smtp
.gmail.com',
port='465',ssl.enable='true',auth='true',content.type='text/html',
subject='Recordatorio de descuentos-
{{comprador}}',to='pgallegosj11@gmail.com')
define stream alertaDescuento(triggered_time long,comprador
string,porcentaje double);
--Tabla que almacena todos los compradores que tienen descuento y su
porcentaje
@PrimaryKey('comprador')
define table tablaDescuentos(comprador string,porcentaje double);
--Funcion que genera el mensaje concatenando dos cadenas y el valor de
descuento calculado
define function funConcat[javascript] return string {
    return "Ha recibido un descuento del "+ data[0]+" por su
fidelidad";
```

```
};
--Evento que se producira cada minuto solo con el timestamp como
informacion
define trigger Aviso1Minuto at every 1 min;

-- Consulta que calcula el consumo y el numero de compras por
comprador
@info(name = 'ConsumoPorPersona')
from compra#window.length(20)
select  sum(consumo) as consumo,comprador as comprador,count() as
numCompras
group by comprador
insert into compraProcesado;
/*
* Consulta que comprueba si se ha realizado mas de una compra en cuyo
caso
* calcula un descuento
*/
@info(name='CreacionDescuento')
from compraProcesado[numCompras >=2 ]
select comprador,true as descuento, ((numCompras-1)*0.1) as porcentaje
insert into compraDesc;

--Consulta que usando funcConcat genera el mensaje de descuento
@info(name='Descuento')
from compraDesc
select  funConcat(porcentaje) as mensaje
insert into respuestaCompra;
/*
* Consulta que cuando se produce un evento de compraProcesado espera
10 segundos
* si no se produce un evento de compraDesc gennera un mensaje sin
descuento
*/
@info(name='NoDescuento')
from not compraDesc for 10 sec and compraProcesado
select  convert("Muchas gracias por su compra",'string') as mensaje
insert into respuestaCompra;
/*
* Consulta que almacena en tablaDescuentos todos los compradores con
descuento
*/
@info(name='TablaDescuento')
from compraDesc
select comprador,porcentaje
update or insert into tablaDescuentos
on tablaDescuentos.comprador==comprador;
/*
* Consulta que cada vez que se genera un evento Aviso1Minuto genera
* un evento alertaDescuento que enviara un email
*/
@info(name='SendEmail')
from Aviso1Minuto join tablaDescuentos
select *
insert into alertaDescuento;
```

ANEXO C: MANUAL DE INSTALACIÓN

En este manual se explicarán los procesos de instalación tanto de Mosquitto (broker usado para la cola MQTT) como del Stream Processor en Ubuntu.

C.1 Mosquitto

Mosquitto es muy sencillo de instalar, solo se deben seguir los siguientes pasos:

1. Agregar repositorio mosquitto dev

```
sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
```

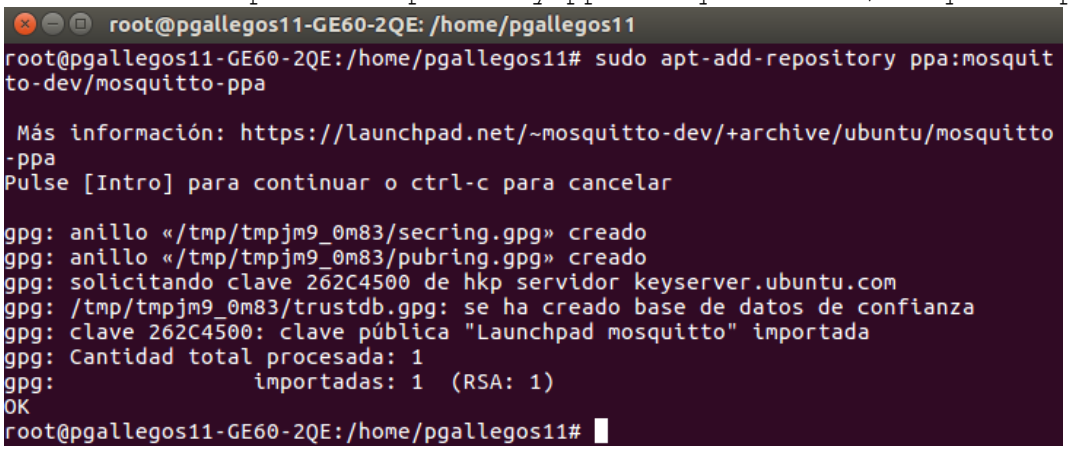


Ilustración 64: Añadiendo repositorio mosquitto

2. Actualizar los repositorios Linux

```
sudo apt-get update
```

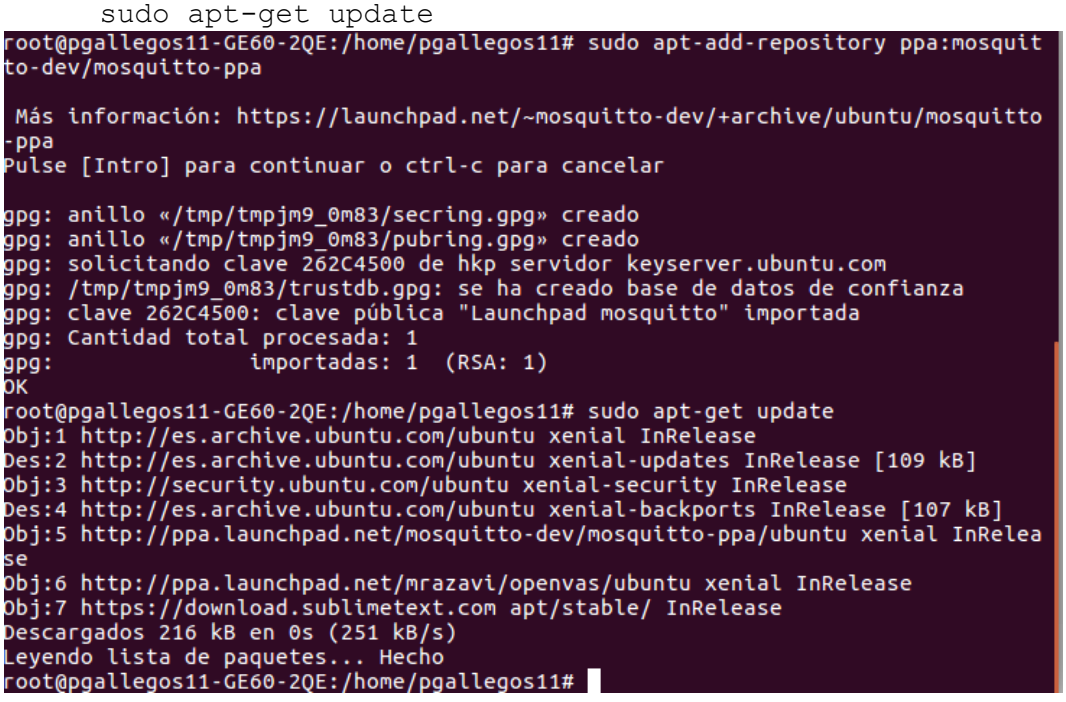


Ilustración 65: Actualizando repositorio

3. Instalar mosquito broker

```
sudo apt-get install mosquitto
```

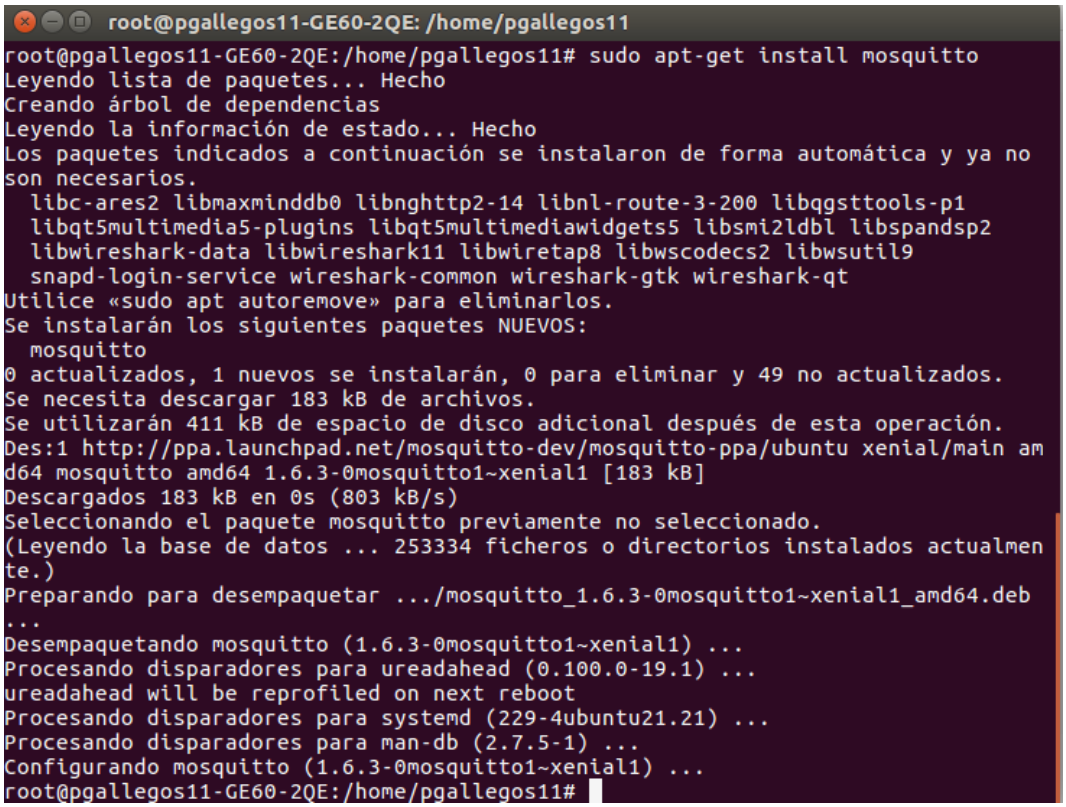


Ilustración 66: Instalación de mosquitto

4. Instalar librerías de desarrollo (innecesario para este proyecto, no obstante, se indican por si se desea trabajar más a fondo con mosquitto)

```
sudo apt-get install libmosquitto-dev
```

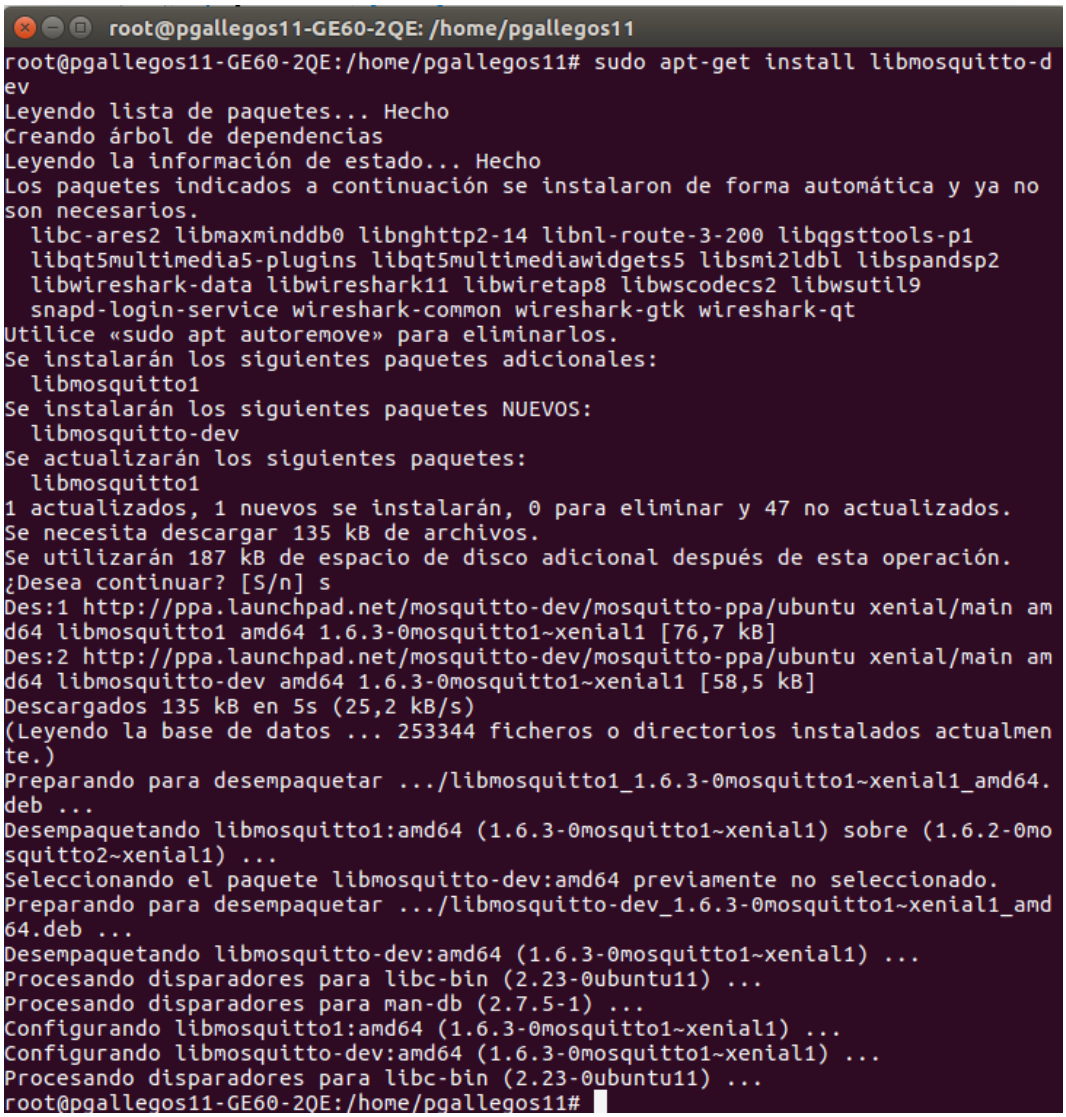


Ilustración 67: Instalación mosquitto-dev

5. Instalar cliente MQTT

```
sudo apt-get install mosquitto-clients
```

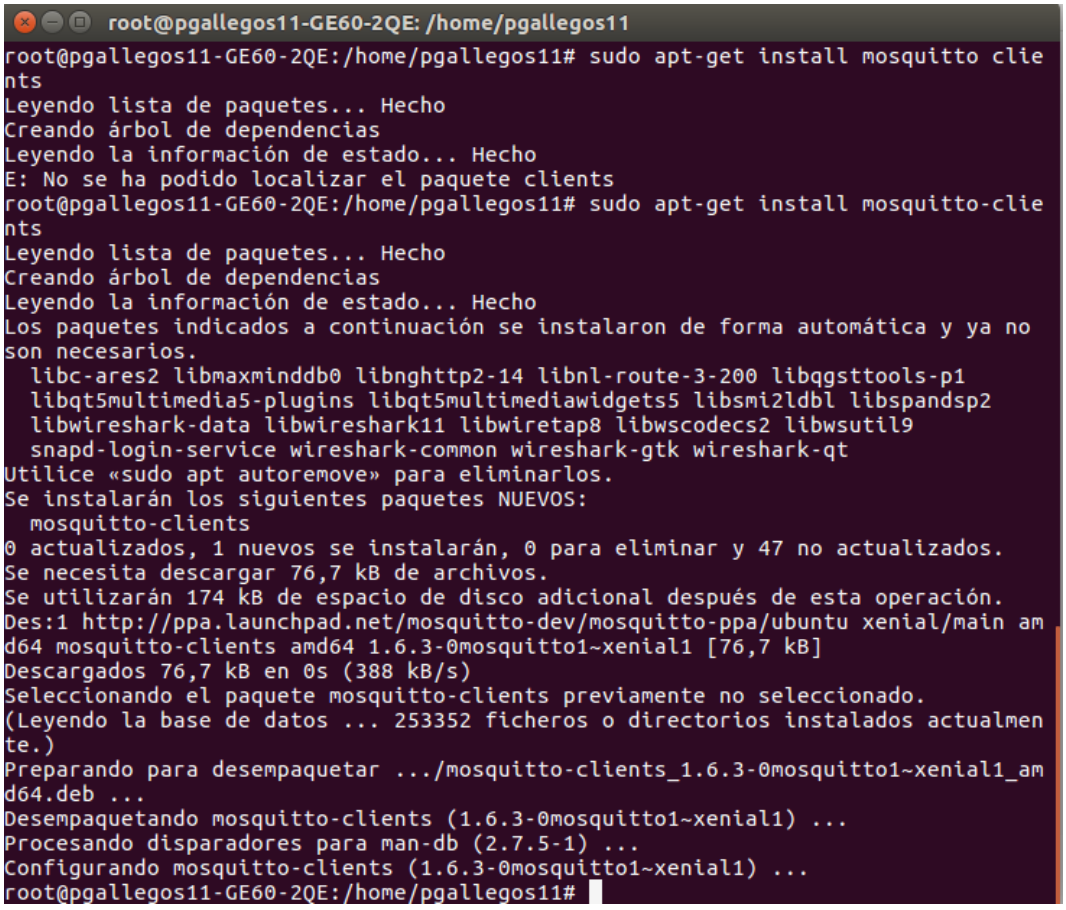


Ilustración 68: Instalación cliente mosquitto

Ya está instalado, para comprobarlo podemos realizar la siguiente prueba:

1. Creamos un subscriptor, en este caso lo hemos subscrito a la dirección localhost y al topic MQTT, el terminal se quedará esperando la llegada de algún mensaje con ese topic.

```
mosquitto_sub -h localhost -t "MQTT" -v
```

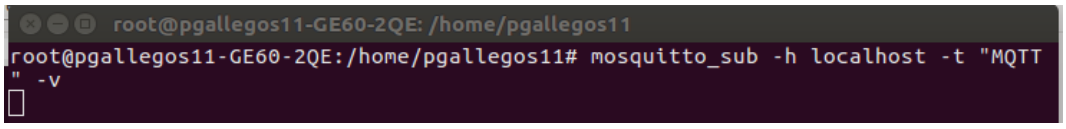


Ilustración 69: Creación subscriptor MQTT

2. Podemos verificar que mosquitto está corriendo

```
sudo service mosquito status
```

Ilustración 70: Comprobación de estado de la cola MQTT

3. Enviamos un mensaje para comprobar si el terminal con el subscriptor lo recibe.

```
mosquitto_pub -h localhost -t "MQTT" -m "Error_Sensor"
```

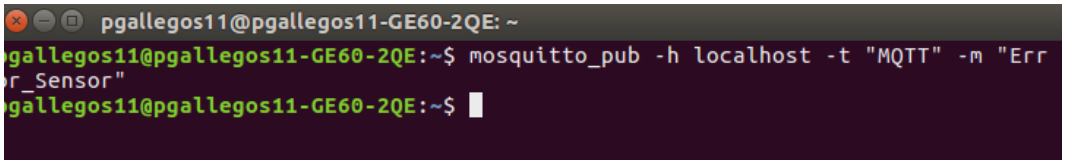


Ilustración 71: Creación mensaje en la cola MQTT

4. Por último, comprobamos que el terminal ha recibido el mensaje.

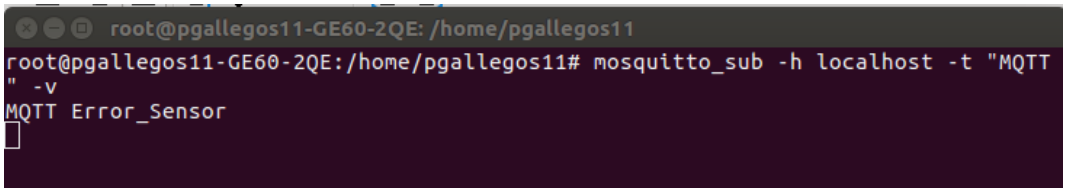


Ilustración 72: Lectura de mensaje de la cola MQTT

C.2 Stream Processor

La instalación del Stream Processor es muy sencilla, al ser free source, desde la propia página web oficial de WSO2 podemos descargarlo [22]:

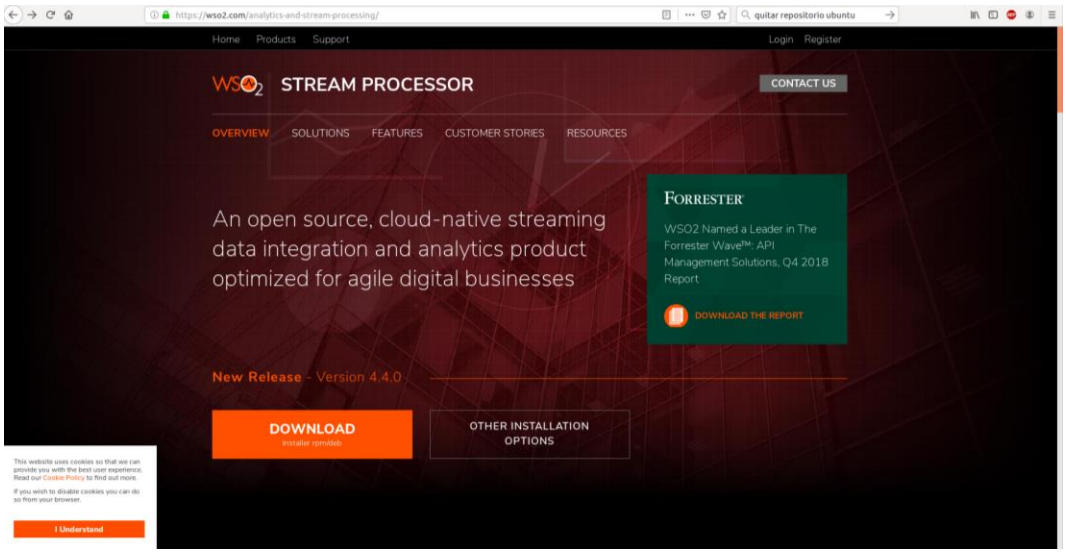


Ilustración 73: Pagina Stream Processor

Incluye diferentes formas de descarga:

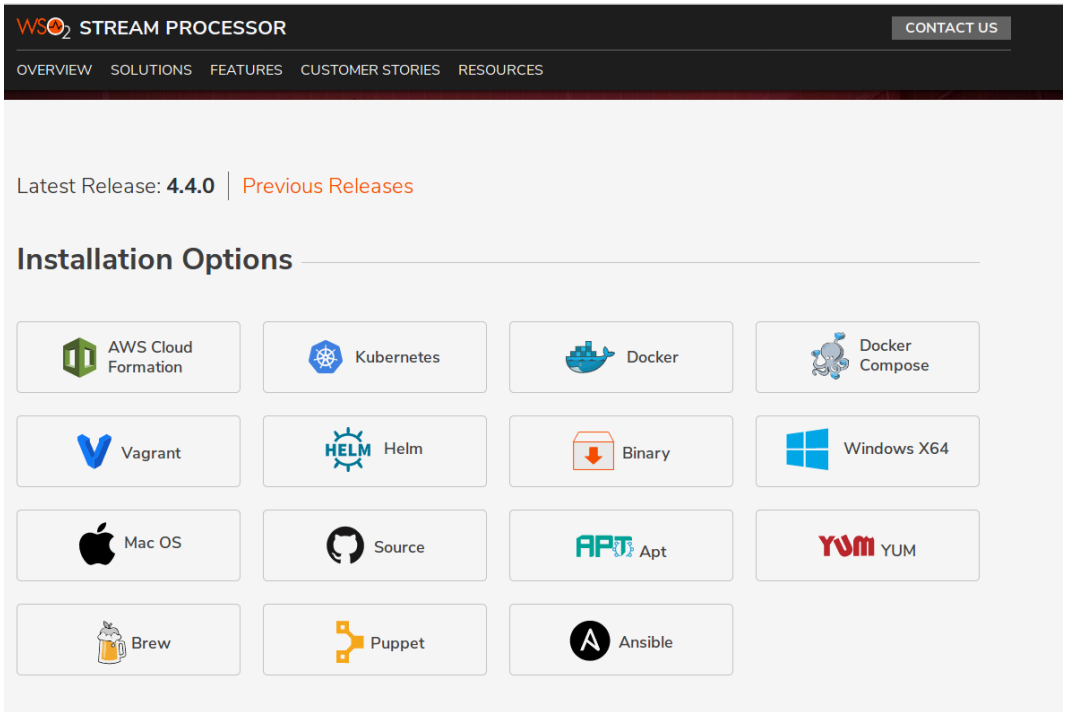


Ilustración 74: Otras opciones de descarga

Para este trabajo se ha optado por instalarlo desde un paquete deb:

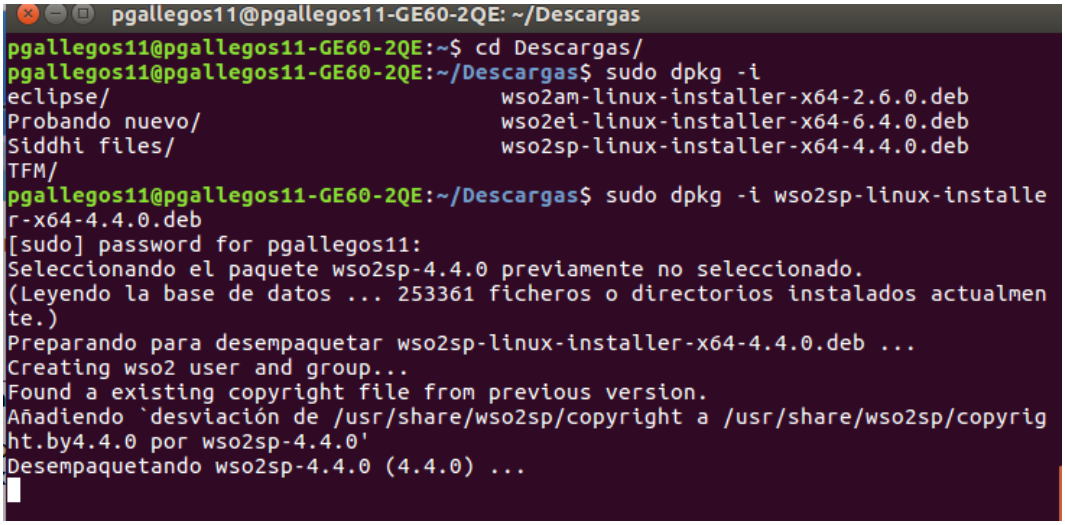


Ilustración 75: Instalación paquete *.deb

Al finalizar la instalación, te muestra los diferentes modos de funcionamiento que se pueden usar del Stream Processor:

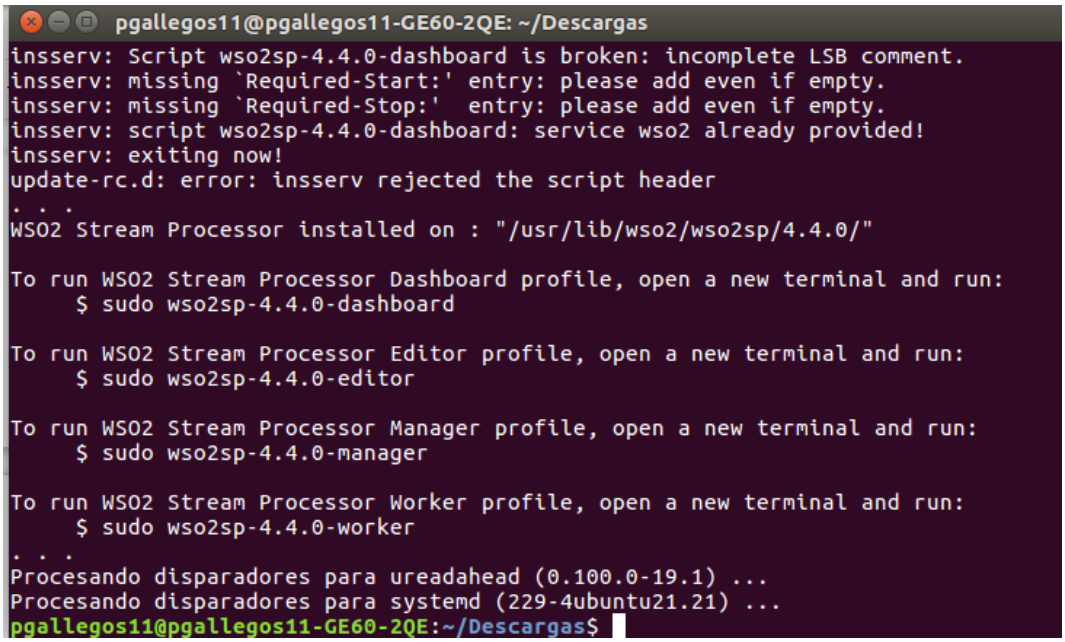


Ilustración 76: Comando WSO2 Stream Processor