Desarrollo dirigido por modelos para aplicaciones de datos en streaming

Mario Gónzalez Jiménez

Máster en Investigación e Innovación en Tecnologías de la Información y las Comunicaciones

MÁSTERES DE LA UAM 2018 - 2019

Escuela Politécnica Superior

UAM Universidad Autónoma de Madrid

Universidad Autónoma de Madrid

Escuela Politécnica Superior

TRABAJO FIN DE MASTER

DESARROLLO DIRIGIDO POR MODELOS PARA APLICACIONES DE DATOS EN STREAMING

Máster Universitario en Investigación e Innovación en TIC

Mario Gónzalez Jiménez
Tutor: Juan de Lara Jaramillo
Departamento de Ingeniería Informática

5 de Septiembre de 2019

DESARROLLO DIRIGIDO POR MODELOS PARA APLICACIONES DE DATOS EN STREAMING

Autor: Mario Gónzalez Jiménez
Tutor: Juan de Lara Jaramillo

Grupo de Modelado e Ingeniería del software(MISO)
Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

5 de Septiembre de 2019

Abstract

Abstract — Real-time data processing is a Big Data paradigm which processes small batches of data with a very low response time, typically milliseconds. The data processing and analysis is applied over a continuous data flow called streaming, in which the data travels through the components of the system that is implemented. This approach is known as fast data. Streaming data applications are used in multidisciplinary areas such as fraud detection, recommendation systems or remote patient monitoring, among others. However, the development of this type of applications is a technological challenge since it requires extensive knowledge about programming languages, algorithms, Big Data architecture and infrastructure. In addition, they have strict requirements for scalability, performance and stability.

This Master's Thesis presents a methodology for the development of streaming data applications based on the Model-driven engineering paradigm. The applications are created using a high-level design with a domain specific visual language, so that it is not a requirement that users have prior technical knowledge in the field. To support this methodology, we have developed an open-source tool called Datalyzer.

Datalyzer is a web platform where each user has a personal space to create, store and run applications in a self-managed way. The domain specific language this thesis presents is integrated into the platform in a graphical interface that contains components which can be interconnected to represent data flows. Datalyzer incorporates a preconfigured collection of data sources and processors that can be used in the applications. In addition, an extension point is offered to allow users adding new data sources and custom processors with the creation of models that contain their configuration. Once the design process is completed, applications are generated automatically using a code generator. The tool has a dashboard where users can monitor and manage the applications that are running in Datalyzer. In addition, the dashboard is also a data visualization tool and it can receive the data that the applications process.

Finally, the implemented system is evaluated with performance and scalability tests. A case study is also presented, based on the open data catalog of the Community of Madrid that contains traffic and pollution information in real time with the goal of generating a monitoring application.

Key words — Streaming data applications, model-driven engineering, domain specific visual languages, cloud computing.

Resumen

Resumen — El procesamiento de datos en tiempo real es un paradigma de los sistemas Big Data que consiste en procesar pequeños lotes de datos con un tiempo de respuesta muy bajo, típicamente milisegundos. El procesamiento y análisis de datos se realiza bajo un flujo continuo llamado streaming, en el cual los datos circulan a través de componentes del sistema que esté implementado. A este enfoque de diseño se le conoce como fast data. Las aplicaciones de procesamiento de datos en streaming tienen usos en áreas multidisciplinares como la detección de fraude, sistemas de recomendación o monitorización de pacientes en el ámbito de la salud, entre otros. Sin embargo, desarrollar este tipo de aplicaciones es un reto tecnológico ya que requiere de amplios conocimientos sobre lenguajes de programación, algoritmos, arquitectura Big Data e infraestructura. Además, tienen unos estrictos requisitos de escalabilidad, rendimiento y estabilidad.

En este Trabajo de Fin de Máster (TFM) se presenta una metodología para el desarrollo de aplicaciones de datos en streaming basada en el paradigma del Desarrollo de Software Dirigido por Modelos. Las aplicaciones se diseñan a alto nivel mediante un lenguaje visual de dominio específico, de forma que no es un requisito que los usuarios tengan conocimientos técnicos previos en la materia. Para dar soporte a esta metodología, se ha desarrollado la herramienta open-source Datalyzer.

Datalyzer es una plataforma web donde cada usuario tiene un espacio personal para crear, almacenar y ejecutar aplicaciones de forma auto-administrada. El lenguaje de dominio específico que este TFM presenta está integrado en la plataforma en un editor gráfico que contiene una serie de componentes que se conectan entre sí representando flujos de datos. Datalyzer incorpora una librería de fuentes de datos pre-configuradas y procesadores que se pueden añadir a las aplicaciones. Además, se ofrece un punto de extensión para que los usuarios puedan añadir nuevas fuentes de datos y procesadores personalizados con la creación de modelos que contienen la configuración de los mismos. Una vez completado el proceso de diseño, las aplicaciones se generan de forma autónoma usando un generador de código. La herramienta tiene un panel de control donde se puede monitorizar y gestionar las aplicaciones que se estén ejecutando en Datalyzer. Además, en el panel de control también se pueden visualizar los datos que las aplicaciones procesan mediante diversos gráficos, tablas y otros visualizadores.

Finalmente, se evalúa el sistema implementado con pruebas de rendimiento y escalabilidad. También se expone un caso de estudio basado en el catálogo de datos abiertos de la Comunidad de Madrid que contiene información de tráfico y contaminación en tiempo real con el objetivo de generar una aplicación de monitorización y alertas.

Palabras clave — Aplicaciones de datos en streaming, cloud computing, desarrollo dirigido por modelos, lenguajes visuales de dominio específico, generación automática de código.

Índice general

Índice de tablas

Índice de figuras

1 Introducción

En este capítulo se describe la motivación para realizar este trabajo (apartado 1.1), las aportaciones que han resultado de la realización del mismo (apartado 1.2), así como la organización del resto del documento (apartado 1.3).

1.1. Motivación

Vivimos en un mundo hiperconectado, en el cual se producen 2.5 quintillones de bytes de datos cada día[1] a través de millones de dispositivos que pueden ser físicos o digitales. Entre otros, dispositivos físicos como móviles, la internet de las cosas (IoT) y sensores o dispositivos digitales como redes sociales, aplicaciones web y micro servicios. Todos ellos tienen la capacidad de generar datos de forma constante o en *streaming* de los cuales se puede extraer información valiosa para las empresas [1].

Debido a su potencial, el análisis de datos en tiempo real es uno de los principales paradigmas actuales a seguir en el mundo de las TIC [2]. Por ejemplo, monitorizar sistemas permite detectar fallos y anomalías de forma más ágil que un enfoque basado en el análisis de historiales *logs*, lo que permite tiempos de respuesta menores y facilita el mantenimiento predictivo [3]. También podemos definir reglas de acción según evoluciona la actividad, o incluso, prever situaciones futuras en base los datos actuales [4]. Además, los avances en aprendizaje automático e inteligencia artificial nos proporcionan conocimiento y ayudan a construir modelos a partir de los datos que de otra forma no sería posible obtener [5]. Por todo ello, el análisis de datos en tiempo real es de gran interés para las

[1] https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/

empresas, especialmente para empresas digitales (e.j.: Amazon, Ebay) o empresas que analizan patrones para predecir comportamiento usando las redes sociales (e.j.: Facebook, Google).

Sin embargo, desarrollar este tipo de aplicaciones tiene asociado algunos problemas y limitaciones, como el hardware necesario para procesar flujos de datos [6], si bien los avances en el hardware y nuevas tecnologías en los últimos años han minimizado este problema. También existen otros problemas comunes como la gestión de los datos, consolidación y compatibilidad entre diferentes sistemas operativos que se pueden solventar usando nuevos frameworks de desarrollo que han salido al mercado como NATS Streaming[2] o herramientas comerciales [7]. Además, las aplicaciones de streaming de datos tienen que reunir unos estrictos requisitos de eficiencia y estabilidad para su correcto funcionamiento. En [8] los autores establecen las características no funcionales principales que debe cumplir una aplicación para el procesamiento de datos en streaming: escalabilidad, tolerancia a fallos y eficiencia para evitar cuellos de botella.

No obstante, la curva de aprendizaje de las tecnologías asociadas a este ámbito es elevada. En una reciente encuesta [9], los autores concluyen la necesidad de tecnologías de alto nivel que ayuden a los desarrolladores en el despligue y la definición arquitectónica de este tipo de aplicaciones. Por este motivo el desarrollo de las aplicaciones de procesamiento de datos usualmente está reservado a usuarios avanzados –como ingenieros de datos– con perfil multidisciplinar, habilidades técnicas, conocimiento de bases de datos y diversos lenguajes de programación. Este hecho es una clara limitación en ámbitos que no son cercanos a la ingeniería del software, porque que el procesamiento de datos en streaming también es de gran utilidad en otros ámbitos. Por ejemplo, capturar transacciones económicas permite la monitorización para detectar errores y su posterior análisis [10]. En el sector del turismo se emplean dispositivos móviles para obtener información sobre la localización del flujo de turistas y analizar el comportamiento de los mismos en las áreas de interés [11]. Otro ejemplo lo encontramos en los sistemas inteligentes de gestión del tráfico a partir del procesamiento de sensores fijos y móviles [12].

En los últimos años, la computación en la nube o *cloud computing* ha ganado protagonismo como alternativa al desarrollo en infraestructura tradicional [13]. Se estima que el mercado llegará a facturar 411 billones de dolares en 2020, mientras que en 2016 fue de 219 billones de dolares[3]. Actualmente, la nube ya no se limita a alojar servidores si no que podemos encontrar productos innovadores para el análisis de datos, aprendizaje automático, la internet de las cosas y muchas más. Las empresas que apuestan por los servicios cloud crecen un 19,6 % más frente sus competidoras que no lo hacen[4]. La clave de este crecimiento se basa en las ventajas que aporta a las empresas, como la disponibilidad inmediata, los sistemas modulares y el software bajo demanda que repercute en una mayor productividad y ahorro de costes.

En este documento se describe una solución cloud para el desarrollo de aplicaciones

[2] https://nats-io.github.io/docs/

[3] https://www.forbes.com/sites/louiscolumbus/2017/10/18/cloud-computing-market-projected-to-reach-411b-by-2020/

[4] https://www.muycomputerpro.com/2018/08/22/valor-cloud-computing-l2020

de streaming de datos basada en el paradigma del Desarrollo de Software Dirigido por Modelos (MDE) [14]. La herramienta tiene un enfoque de alto nivel, en la que no es necesario escribir código en un lenguaje de programación si no que las aplicaciones se crean usando un lenguaje de dominio especifico (DSL) visual. Esta característica aporta algunas ventajas. Por ejemplo, los conocimientos técnicos sobre la materia no son una barrera, al abstraerse de los diversos lenguajes de programación y frameworks existentes. Por lo tanto, la curva de aprendizaje es mucho menor. Además, se proporciona la infraestructura necesaria para ejecutar las aplicaciones. La herramienta está orientada a un público amplio, no necesariamente usuarios avanzados de la ingeniería del software, si no cualquier usuario que necesite procesar flujos de datos como webmasters, matemáticos o economistas.

Este trabajo ha sido financiado por la Comunidad de Madrid con su plan de empleo juvenil PEJD-2016/TIC-3088.

1.2. Contribuciones y aportaciones

Este trabajo ha participado en el proyecto HPI Future SOC Lab organizado por la Universidad de Postdam con la colaboración del Instituto Hasso Plattner. Dicho proyecto tiene como objetivo ofrecer recursos hardware sin ningún coste a investigadores para aplicaciones orientadas al cloud computing. Datalyzer ha sido aceptado e incluido en las convocatorias de otoño de 2017 y primavera de 2018[5].

Conferencias:

1. Mario González-Jiménez, Juan de Lara. **DATALYZER: Streaming data applications made easy**. Proceedings of the Web Engineering: 18th International Conference, ICWE 2018, pp.: 420-429 (Conferencia Core B) (porcentaje de aceptación del 22 %). Junio 2018, Cáceres (España).

2. Sara Pérez-Soler, Mario González-Jiménez, Esther Guerra, Juan De Lara. **Towards Conversational Syntax for Domain-Specific Languages using Chatbots**. En: *ECMFA'19*, 2019. (porcentaje de aceptación del 50 %). Journal of Object Technology.

Póster:

1. Mario González-Jiménez y Juan De Lara. **Streaming applications made easy with Datalyzer**. En: HPI Future Soc Lab Day, Potsdam (Alemania), Abril. 2018. url: *https://hpi.de/en/research/future-soc-lab-service-oriented-computing/publications/posters-2018-1.html*.

[5]https://hpi.de/en/research/future-soc-lab-service-oriented-computing/projects/archive.html

1.3. Estructura del documento

El resto del documento se estructura de la siguiente forma. El capítulo 2 detalla el estado de arte relacionado con el trabajo desarrollado, se introducen conceptos teóricos sobre el Desarrollo de Software Dirigido por Modelos, se explican conceptos sobre aplicaciones de datos en streaming y finalmente se hace un análisis y comparativa con herramientas similares a la desarrollada. El capítulo 3 plantea el enfoque del trabajo, los meta-modelos que se han diseñado siguiendo la metodología MDE y el proceso para la creación de nuevos modelos. El capítulo 4 describe la arquitectura de la herramienta, los módulos que la componen y tecnologías empleadas, así como el front-end y cómo los usuarios interactúan con la herramienta. En el capítulo 5 se evalúa la herramienta mediante diversos experimentos y se describe un caso de estudio usando el catálogo de datos abiertos de la Comunidad de Madrid. Por último, en el capítulo 6 se exponen las conclusiones del trabajo realizado y el trabajo futuro.

2

Estado del arte

En este capítulo se introducen conceptos sobre la metodología de Desarrollo de Software Dirigido por Modelos (apartado 2.1). Además, en el apartado 2.2 presentamos una visión general sobre las aplicaciones de datos en streaming. Finalmente, en el apartado 2.3 presentamos una panorámica de herramientas para crear aplicaciones de este tipo y una breve comparativa.

2.1. Desarrollo de Software dirigido Modelos

El desarrollo de software dirigido por modelos (MDE, del inglés *Model-Driven Engineering*) es un paradigma para el desarrollo de software que propone el uso de modelos en todo el ciclo de desarrollo de software frente a las tradicionales propuestas basadas en lenguajes de programación [14]. El uso de modelos permite mejorar la productividad, la automatización de procesos y algunos otros aspectos de la calidad del software, como el mantenimiento y la interoperabilidad entre sistemas [15]. Los pilares básicos del MDE son, por tanto, los modelos, las transformaciones de modelos y los lenguajes de dominio específico. El uso de la metodología MDE aporta diversos beneficios como:

- **Arquitectura del software.** Elaboración de un diseño independiente de la tecnología o herramientas mediante el uso de modelos, centradas únicamente en los elementos del sistema.

- **Familia de aplicaciones.** Diseño de una arquitectura con un nivel alto de abstracción que es válido para una familia de aplicaciones en un dominio particular.

- **Productividad.** Incremento de la productividad al ofrecer lenguajes de más alto nivel, orientados al dominio del problema.

En el MDE los modelos se usan para representar sistemas existentes o que se pretende implementar en el futuro. Un modelo trata un aspecto concreto del sistema, definido a un nivel alto de abstracción independiente a la tecnología empleada. Por tanto, un modelo permite el análisis del sistema incluso previo a la implementación. Pero un modelo no solo representa sistemas de software, si no cualquier elemento de la realidad. Se puede entender como una formalización de conceptos más cercano al entendimiento humano que el código de un lenguaje de programación. Un modelo se expresa mediante un *lenguaje de modelado*. El más utilizado actualmente es el estándar UML (Unified Modeling Languague)[1]. UML es un lenguaje de modelado de propósito general que se usa para construir software en múltiples dominios. En contraposición, existen otros lenguajes de modelado para dominios específicos que se llaman lenguajes de dominio específico (DSL) [15].

Un modelo tiene asociado una semántica, es decir, un significado que sirve de traducción entre los símbolos o elementos del modelo y lo que representa en la realidad. Supongamos el modelo de la Figura 2.1. Sin una semántica este modelo podría representar personas y la cantidad de amigos que tienen en común en una red social entre ellos. Pero gracias a su semantica sabemos que su interpretación correcta son ciudades y la distancia que hay entre ellas en kilómetros.

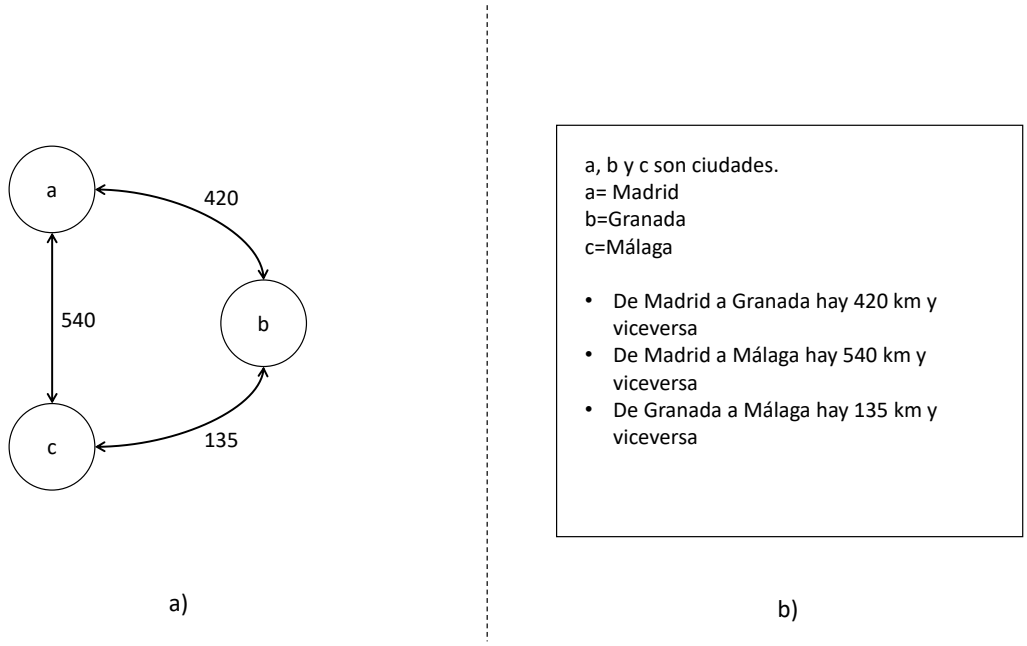

Figura 2.1: a) Modelo b) Interpretación del modelo

Por otro lado, un modelo también tiene una sintaxis concreta y abstracta. La sintaxis concreta, también llamada notación, es el conjunto de símbolos (e.g. gráficos, textuales)

[1] https://www.omg.org/spec/UML/About-UML/

utilizado en un lenguaje de modelado junto con la definición de sus propiedades. En este caso en concreto podemos observar como el lenguaje está formado por círculos, que representan ciudades y arcos que representan las conexiones. Sin embargo, otra notación para el mismo modelo podría utilizar cuadrados en vez de círculos y la traducción del modelo sería exactamente la misma. El modelo subyacente es el mismo independientemente de la notación. Lo que sí es igual es la estructura del modelo, es decir, la combinación lógica de los elementos. De esto se encarga la sintaxis abstracta, que se define con un meta-modelo. Este concepto se explica en la siguiente subsección.

2.1.1. Meta-modelado

Un meta-modelo define la estructura de datos de todos los elementos de una notación y las reglas para las conexiones entre ellas. Sin embargo, también puede expresarse de forma gráfica o textual. Así pues, el meta-modelo también es un modelo en si mismo. Un meta-modelo se define como un modelo de la sintaxis abstracta de un lenguaje de modelado. Mediante este lenguaje podremos crear nuevos modelos válidos conforme a la sintaxis correspondiente. Como un meta-modelo es un modelo en sí mismo, se puede plantear la cuestión de qué lenguaje define el lenguaje de un meta-modelo y así sucesivamente. Para resolver esta cuestión, la OMG propuso la arquitectura de cuatro niveles (Figura 2.2) que establece la relación entre los modelos y los meta-modelos [16]. Los niveles se definen como:

- **M0. Nivel de datos.** Es una instancia de un modelo. Representa un estado, propiedades o procesos del mundo real.
- **M1. Nivel de modelos.** Es una instancia de un meta-modelo. Representa las propiedades del mundo real que se pueden instanciar en el nivel M0.
- **M2. Nivel de meta-modelos.** Es una instancia de un meta-meta-modelo. Define un lenguaje para crear modelos. Uno de los lenguajes mas extendidos es el UML.
- **M3. Nivel de meta-meta-modelos.** Define un lenguaje para crear meta-modelos. La OMG ha propuesto el MOF [16] que es ampliamente usado en la práctica gracias a su implementación dentro de Eclipse (EMF) [17].

La técnica del modelado consiste en la creación de modelos a partir de un lenguaje de modelado definido en un meta-modelo. Por ejemplo, en el anterior apartado hemos diseño un modelo para describir ciudades y la distancia que hay entre si. En la Figura 2.3 se muestra un meta-modelo con el formato UML que define el lenguaje para la creación de este tipo de modelos. El meta-modelo contiene dos clases `Ciudad` y `Camino`. La clase `Ciudad` que representa una ciudad tiene una propiedad `nombre` mientras que la clase `Camino` tiene una propiedad `distancia` que es de tipo entero. Además, la clase `Camino` necesita la referencia de dos instancias de la clase `Ciudad` para definir qué distancia y entre qué ciudades se produce.

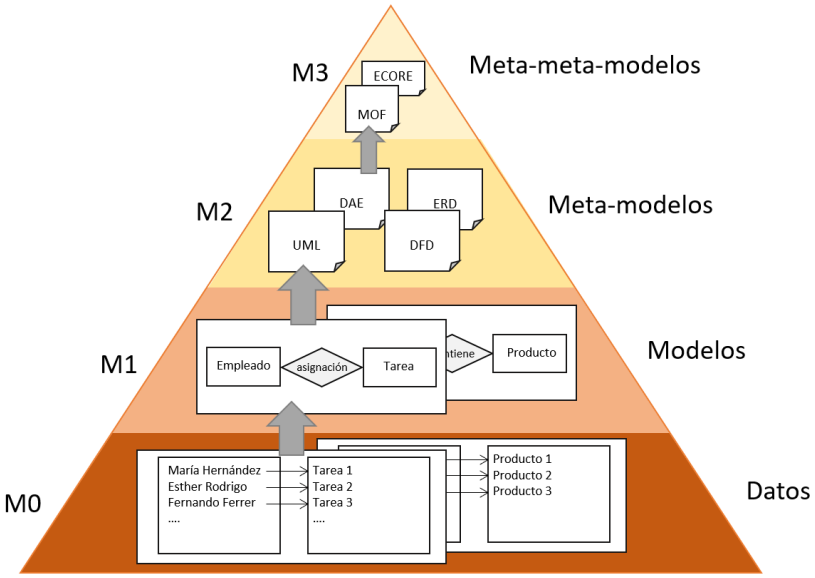

Figura 2.2: Arquitectura de cuatro niveles de la OMG [16].

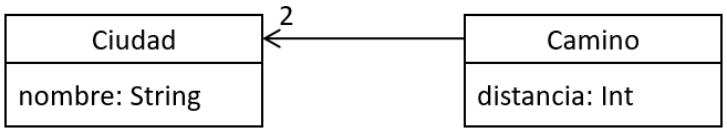

Figura 2.3: Meta-modelo que define el lenguaje para el modelo de la Figura 2.1.

Nótese que un meta-modelo define un lenguaje de modelado: los conceptos del lenguaje, su relación entre ellos y las reglas para que un modelo esté bien formado. Pero el meta-modelo no define su notación (aspecto visual) como se ha explicado anteriormente. Por otro lado, un DSL incluye tanto la abstracta como la definición de la sintaxis concreta.

2.1.2. Lenguaje de Dominio específico (DSL)

Un lenguaje de dominio específico (DSL del inglés *domain specific language*) es un lenguaje de modelado, que puede ser tanto gráfico como textual, orientado a resolver y representar un problema concreto en lugar de usar un lenguaje de meta-modelado de propósito general como UML. Son lenguajes orientados a describir un ámbito concreto de la realidad que se llama dominio. Por lo tanto, los DSLs son especialmente útiles para ser usado por *expertos del dominio* en áreas y disciplinas diversas como la economía, la biología o la educación.

El uso de los DSLs permite mejorar la productividad: son más precisos, expresivos y se adaptan al problema que queremos resolver. Pero de forma intrínseca de sus ventajas también obtenemos sus inconvenientes: desarrollo costoso, mantenibilidad y limitación a un ámbito concreto. Desde un punto de vista de la construcción del lenguaje, los DSLs se pueden categorizar en:

- **Externo.** Es un DSL construido desde cero, con sintaxis propia y compilador o *parser*. Se puede catalogar como un lenguaje en si mismo. Este tipo de DSLs son los más comunes.

- **Interno.** Es un DSL construido a partir de un lenguaje de programación de propósito general, cuya sintaxís se modifica para facilitar la usabilidad aplicado a un dominio específico. Son usados para facilitar la construcción de estructuras de datos, tareas, procesos, etc. Muchos DSLs de este tipo son usados para facilitar el uso de APIs existentes.

Desde un punto de vista del formato del lenguaje, se puede categorizar entre:

- **Textual.** Es un lenguaje formado por un conjunto de sentencias, como cualquier lenguaje de programación. La mayoría de los lenguajes en la informática son de este tipo. Para crear un DSL textual existen herramientas como Xtext [18] o EMFText [19]. Un DSL textual muy conocido es `SQL` que sirve para realizar consultas a bases de datos.

- **Gráfico.** Es un lenguaje basado en elementos gráficos, como el lenguaje `UML` (Figura 2.3). El funcionamiento es similar a un lenguaje textual, con la diferencia de utilizar figuras y conectores para definir conceptos. Para crear un DSL gráfico existen herramientas como Sirius [20] o Graphiti [21].

2.1.3. Generación de código

Las transformaciones de modelos son uno de los pilares básicos del MDE [22]. Una transformación produce, partiendo de uno o más modelos de entrada, uno o más modelos de salida. Este proceso se conoce como una transformación modelo a modelo, pero también existen transformaciones de modelo a texto, como los generadores de código. Un generador de código es un tipo especial de transformación de modelos que tiene como objetivo producir código fuente en un lenguaje de programación común (e.g. Java) a partir de un modelo que describe típicamente una aplicación o un sistema. Funciona de una forma similar a un compilador que produce código binario a partir de código fuente, pero a un nivel más alto de abstracción. Para desarrollar un generador de código hay dos estrategias. La primera consiste en desarrollar un `parser` que recorra el modelo para generar el código fuente de salida. La segunda estrategia consiste en emplear un `parser específico` para la transformación de modelos que también usa el meta-modelo para simplificar este proceso.

El método más común para generar el código fuente es mediante el uso de un motor de plantillas. El proceso es muy parecido a la programación de webs dinámicas como *Java Server Pages* (JSP). Una plantilla contiene un conjunto de sentencias de código fuente que se combina con la información recuperada del modelo de entrada. Existen herramientas para la creación de generadores de código a partir de plantillas como Acceleo [23] o Xtend [24].

2.2. Procesamiento de datos en streaming

El procesamiento de datos en tiempo real es uno de los campos de mayor interés para las empresas y de gran dinamismo debido al continuo lanzamiento de nuevos productos al mercado. Desde el punto de vista de la investigación, es un campo desafiante por los retos tecnológicos que requiere desarrollar este tipo de aplicaciones. Existen multitud de tecnologías válidas para el procesamiento de datos en tiempo real, con distintos enfoques a nivel de diseño. Entender las posibles arquitecturas y elegir qué tecnología es la más adecuada en cada caso es todo un reto.

Sin embargo, todas las arquitecturas, enfoques y tecnologías tienen las mismas premisas en común. Como se detalla en [8], una aplicación para el procesamiento de datos en tiempo real es adecuada cuando se cumplen los siguientes requisitos:

- **Flujo de datos.** Con el objetivo de garantizar una baja latencia, es indispensable que se produzca el procesamiento de los mensajes sin la necesidad de almacenarlos en disco para realizar operaciones sobre ellos.
- **Tolerante a fallos.** La tecnología empleada tiene que incorporar mecanismos para soportar fallos en la transmisión de mensajes como retrasos, mensajes fuera de orden o perdidos.
- **Salida.** La salida de la aplicación tiene que ser consistente en el tiempo.
- **Escalabilidad.** Un sistema para el procesamiento de datos debe ofrecer la posibilidad de distribuir el procesamiento entre distintas maquinas para garantizar su escalabilidad. Idealmente, este proceso deberá ser automático y transparente para el desarrollador.
- **Optimización.** La optimización es clave para un rendimiento adecuado en un sistema que debe garantizar alta fiabilidad y respuesta inmediata incluso con altos volúmenes de tráfico.

A nivel de diseño debemos diferenciar entre **procesamiento por lotes** y **procesamiento en streaming**. El procesamiento por lotes también conocido como *batch processing* opera los mensajes por bloques que se han ido almacenando de forma temporal en un período de tiempo determinado. Por ejemplo, procesar todos los pedidos que se han producido en un día, o todas las transacciones recibidas en un minuto. Como vemos, a nivel de diseño no hay restricción en el período de tiempo (llamado ventana temporal) que podemos emplear. Sin embargo, una ventana temporal amplia nos aleja del procesamiento de datos en tiempo real que es nuestro objetivo. En cualquier caso, el procesamiento por lotes añade un *delay* a la salida puesto que el procesamiento no comienza justo al recibir un dato. El procesamiento por lotes es adecuado para procesar grandes volumenes de datos que conviene analizar en su conjunto en vez de forma individual sin importar en exceso la latencia. Una tecnología muy extendida para este tipo de procesamiento es Apache Spark[2].

[2] https://spark.apache.org/

El procesamiento en streaming, en contraposición al anterior enfoque, está diseñado para trabajar a tiempo real: se procesa cada dato individualmente o en microlotes de pocos registros según es recibido. Por lo tanto, la latencia de un sistema así es mínima (milisegundos). Está orientado a aplicar operaciones poco costosas a los datos como agregaciones o métricas. Si fuera de otro modo, cabría el riesgo de tener un cuello de botella si nuestro sistema tarda más tiempo en procesar un dato que en recibir el siguiente. El procesamiento de datos en streaming es adecuado para tareas como la detección de fraude en transacciones online o detección de anomalías en el tráfico de una red [25]. Una tecnología muy extendida para este procesamiento es Apache Kafka [26]. La Figura 2.4 muestra de forma gráfica la diferencia de procesamiento de datos entre ambos enfoques.

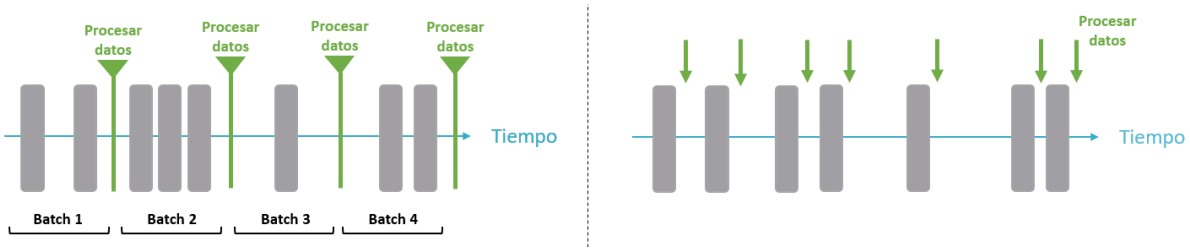

Figura 2.4: a) Procesamiento por lotes b) Procesamiento en streaming

Independientemente del tipo de procesamiento de datos con el que trabajemos, una arquitectura para el procesamiento de datos en tiempo real requiere de, al menos, los siguientes componentes:

- **Ingesta de datos.** Es un módulo que se encarga de capturar y almacenar (usualmente en memoria) los mensajes en tiempo real procedente de una o más fuentes de datos. Este módulo deberá ser compatible con protocolos de transmisión de mensajes como API Rest o Socket TCP.
- **Procesamiento de datos.** Una vez capturados los mensajes, se aplica el procesamiento de interés para su análisis como filtros o agregaciones.
- **Salida.** Una vez tengamos seleccionados y procesados los datos de interés, la aplicación produce una salida. Típicamente, los datos se almacenan en una base de datos o en logs, o bien se redireccionan en un stream a otra aplicación. Posteriormente sobre estos datos podremos generar informes y reportes.

2.3. Tecnologías cloud para el procesamiento de datos en streaming

En este apartado se describen tecnologías y herramientas para la creación de aplicaciones de datos en streaming basadas en la idea de un diseño de las aplicaciones a alto nivel. Es decir, no se explicarán frameworks de desarrollo si no soluciones completas. En el apartado 2.3.1 se reúnen varios productos comerciales y en el apartado 2.3.2 herramientas open-source orientadas a la investigación.

2.3.1. Herramientas comerciales

Cloud Dataflow

Cloud Dataflow (*https://cloud.google.com/dataflow/*) es un producto de Google Cloud para el procesamiento de datos en la nube. Es compatible tanto con el procesamiento de datos por lotes como el procesamiento en streaming. Esta herramienta permite el desarrollo simplificado de flujos de datos por medio de APIs en SQL, Java y Python para el SDK de Apache Beam[3]. Esto ofrece al usuario una completa librería de operaciones, primitivas de análisis, conectores y más que facilita el desarrollo.

La principal ventaja que encontramos en Cloud Dataflow es la simplificación del proceso de gestión y despliegue de las aplicaciones. Está diseñado para que el usuario se centre en la programación y sea la herramienta la que gestione muchas tareas de administración como el rendimiento, la seguridad y la escabilidad del sistema entre otros.

Como desventaja, el entorno de desarrollo no está unificado con el entorno de ejecución. Se deben desarrollar los scripts en local y ejecutarlos mediante la línea de comandos. El entorno de ejecución incorpora herramientas para monitorizar las aplicaciones que se estén ejecutando en tiempo real pero no incorpora un panel de control para la visualización de los datos que generan las aplicaciones.

Azure Data Factory

Azure Data Factory (*https://azure.microsoft.com/es-es/services/data-factory/*) es un producto de Microsoft Azure para crear procesos ETL sin necesidad de programar código, usando un DSL visual que está integrado en la propia herramienta. Al DSL se accede mediante una aplicación web que tiene una interfaz de usuario con un sistema drag-and-drop donde podemos conectar fuentes de datos con bloques de procesamiento y redireccionar la salida a un sistema de destino. También incorpora formularios para la configuración de cada componente de forma visual. Para usuarios avanzados, también permite añadir código personalizado mediante el lenguaje C#.

Un proceso ETL se puede entender como una aplicación de procesamiento de datos pero limitado a las siguientes características:

- **Ingesta de datos.** Carga de los datos desde los sistemas de origen. Típicamente son conexiones a bases de datos.
- **Transformación de los datos.** Se aplican reglas de negocio o funciones sobre canales de datos.
- **Persistencia de datos.** Se almacenan los datos de salida que produce el sistema. Típicamente se vuelcan a una base de datos.

[3] https://beam.apache.org/

Como podemos apreciar, un proceso ETL es una aplicación sencilla de datos orientada a trabajar con bases de datos. El procesamiento también puede ser a tiempo real si se configura correctamente, ya que actualmente existen bases de datos preparadas para este propósito como Couchbase[4]. Además, Azure Data Factory incorpora más de 80 conectores preinstalados para ingestar datos desde distintas tecnologías como Azure SQL Database[5], Amazon Redshift[6] o Hive[7].

Al igual que ocurre con los productos de Google Cloud, los productos de Azure Microsoft están desarrollados de una forma modular para que se integren sus propios servicios entre sí. Azure Data Factory no incluye herramientas para la visualización de datos o análisis en tiempo real, pero se puede conectar a Power BI[8] para este propósito.

Amazon Kinesis

Amazon Kinesis (*https://aws.amazon.com/es/kinesis/*) es un producto de Amazon Web Services para el procesamiento de datos en streaming en tiempo real. La principal característica que encontramos es su capacidad de ingestar datos de forma sencilla provenientes de fuentes heterogéneas como vídeos, audios, logs, dispositivos IoT o incluso aplicaciones. Para el procesamiento de los datos se utiliza SQL y Java. Por ejemplo, se puede establecer una consulta SQL estándar que se ejecuta de forma continuada en un flujo de datos.

Amazon Kinesis es una herramienta auto-gestionada. Por si misma despliega la infraestructura necesaria para ejecutar las aplicaciones. Además escala cualquier volumen de datos con latencias muy bajas. La salida de las aplicaciones se puede redireccionar como un stream de salida a herramientas de visualización de datos u otras aplicaciones.

2.3.2. Herramientas no propietarias

Apache Nifi

Apache Nifi (*https://nifi.apache.org/*) es una plataforma para crear procesos ETL en tiempo real de forma rápida e intuitiva. La herramienta tiene un enfoque de alto nivel, gracias a incorporar un DSL donde se diseñan los flujos de datos y sus operaciones de forma visual. La interfaz de usuario se basa en una aplicación web con un sistema drag-and-drop para conectar componentes y redireccionar flujos de datos. En la propia interfaz mediante formularios se puede configurar cada componente, por ejemplo, aplicar un filtro a un campo concreto. Incorpora más de 200 procesadores de datos pre-instalados para hacer todo tipo de transformaciones como codificar una cadena de caracteres, partir un

[4]https://www.couchbase.com/
[5]https://azure.microsoft.com/es-es/services/sql-database/
[6]https://aws.amazon.com/es/redshift/
[7]https://hive.apache.org/
[8]https://powerbi.microsoft.com/es-es/

texto o ejecutar una consulta SQL. Además, se pueden crear procesadores personalizados en código Java.

Apache Nifi puede funcionar a nivel local, o desplegarse en una infraestructura cloud. Dentro de sus características encontramos la seguridad, como la compatibilidad con SSL, HTTPS, o encriptación de contenido. La herramienta es altamente configurable (e.j. latencia, tolerancia a fallos) pero no es auto-administrada, es decir, no gestiona los recursos por si misma. Además, como su objetivo son los procesos ETL tampoco incluye opciones de visualización de los datos que están procesamiento. Para este propósito debemos redireccionar el flujo de datos a herramientas externas.

Apache Zeppelin

Apache Zeppelin (https://zeppelin.apache.org/) es una aplicación web en forma de *notebook* que sirve para hacer análisis de datos de forma interactiva, así como crear gráficos o incluso ejecutar código en Python o Scala. También se pueden ejecutar consultas en SQL sobre los datos que devuelven tablas, texto o gráficos. Aunque el uso de los notebooks es intuitivo, se requiere de conocimientos de programación y algunos fundamentos de conceptos previos.

Apache Zeppelin no es una herramienta de procesamiento de datos en si misma, pero puede conectarse a una de forma nativa. En concreto, tiene soporte para la conexión e ingesta de datos a través de Apache Spark. Sin embargo, este proceso tiene que ser configurado por el usuario de forma manual. Además, no incluye características para la administración como rendimiento y seguridad.

2.3.3. Conclusiones

En los anteriores apartados hemos estudiado varias herramientas para la creación de aplicaciones de datos en streaming que comparten similitudes con Datalyzer. Sin embargo, debido a que cada una de ellas tiene un enfoque distinto, detectamos algunas limitaciones y deficiencias que queremos solventar con el desarrollo de nuestra herramienta. Por ejemplo, Azure Data Factory es una herramienta de alto nivel que incorpora un DSL visual para el diseño de aplicaciones, pero su uso se limita principalmente a bases de datos y no a cualquier fuente de datos dinámica. Otro caso similar nos lo encontramos en Apache Nifi, una herramienta más abierta y extensible que la anterior pero no incorpora un entorno de trabajo completo con gestión de usuarios, proyectos simultáneos o un panel de control para la monitorización y visualización de datos en tiempo real. También podemos observar algunas herramientas más potentes y versátiles, como Amazon Kinesis o Cloud Dataflow, pero en ambas nos encontramos con el problema de una amplia curva de aprendizaje. Además, como son productos comerciales, tienen el problema de ser *vendor lock-in*, es decir, para ser usadas dentro del ecosistema de la propia empresa. En este sentido, detectamos que puede ser de utilidad una herramienta open-source que además puede servir de base para futuros proyectos.

Por todas estas limitaciones que detectamos en el estado del arte relacionado, presentamos en este TFM una herramienta open-source que integra el desarrollo de aplicaciones y el despliegue de mismas. Además, se ha diseñado para ofrecer extensibilidad respecto a las fuentes de datos y procesadores. A modo de resumen, se presenta en la Tabla 2.1 una breve comparativa de las principales características de Datalyzer y las herramientas que hemos estudiado anteriormente.

Herramienta	Fuentes heterogéneas	Procesadores preinstalados	Procesadores personalizados	Batch	Streaming	Stream processing	Visualización de datos	DSL	Auto administrada
Datalyzer	Sí	Sí	Sí	No	Sí	Sí	Sí	Sí	Sí
Cloud Dataflow	Sí	Sí	Sí	Sí	Sí	Sí	No	No	Sí
Azure Data Factory	No	Sí	Sí	Sí	No	No	No	Sí	Sí
Amazon Kinesis	Sí	No	Sí	Sí	Sí	Sí	Sí	No	Sí
Apache Nifi	Sí	Sí	Sí	No	Sí	Sí	No	Sí	No
Apache Zeppelin	No	No	Sí	No	No	No	Sí	No	No

Tabla 2.1: Comparación de las principales características de las herramientas estudiadas y Datalyzer

3 Enfoque

En este capítulo se explicará el enfoque y los conceptos que propone este trabajo. En el apartado 3.1 se describe el planteamiento general de Datalyzer. En el apartado 3.2 se explican los meta-modelos propuestos para describir aplicaciones de datos, y en el apartado 3.3 cómo se crean modelos conforme a los meta-modelos propuestos usando nuestra herramienta.

3.1. Planteamiento general

En este TFM se presenta, por las motivaciones expuestas en el apartado 1.1, una herramienta para el desarrollo de aplicaciones de datos en streaming siguiendo un enfoque de la metodología de Desarollo de Software Dirigido por Modelos [27]. Usando técnicas MDE que se han explicado en el apartado 2.1, el desarrollo de las aplicaciones se realiza a alto nivel gracias al modelado. En concreto, se diseñan mediante un lenguaje de dominio específico (DSL) visual, y se generan de forma automática mediante un generador de código. La Figura 3.1 muestra el diseño conceptual de las aplicaciones a nivel de componentes inspirada en un proceso ETL. Mediante un módulo de ingesta de datos, las aplicaciones pueden estar conectadas a cualquier fuente de datos dinámica para recibir información que actúa como entrada en el sistema. Por ejemplo, pueden conectarse a un API Rest, recibir datos a través de un TCP Socket o incluso servicios web como Twitter[1]. Los datos que se ingestan pasan por canales de datos sobre los que se pueden añadir procesadores personalizados: agregadores, filtros y más. Por último, los nuevos datos que se generan de salida sirven para su posterior análisis, visualización o almacenamiento.

[1] https://twitter.com/

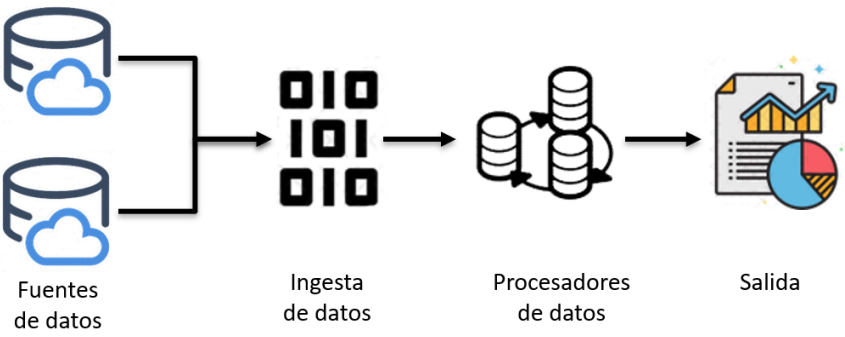

Figura 3.1: Diseño de las aplicaciones de datos en streaming.

La herramienta, llamada Datalyzer, se ha desarrollado en forma de plataforma web. Funciona, por tanto, como un servicio en la nube centralizado donde el único requisito software para el usuario es un navegador web. Es posible ejecutarlo tanto de forma local creando un servidor en la propia máquina del usuario, como en una infraestructura cloud real. En la plataforma web, los usuarios pueden realizar distintas acciones que de forma ilustrativa se muestra en la Figura 3.2:

- **Registro.** Todos los usuarios tienen que estar registrados para acceder a la plataforma. El registro se realiza mediante un formulario.
- **Espacio personal.** Cada usuario registrado en la plataforma dispone de un espacio personal para proyectos (aplicaciones) que pueden guardar, editar o ejecutar en cualquier momento.
- **Añadir fuentes de datos.** Cada fuente de datos que podemos usar como entrada en las aplicaciones tiene asociado un modelo que describe su configuración (e.g. url, autentificación) almacenado en una base de datos. Para garantizar extensibilidad, los usuarios pueden añadir nuevas fuentes de datos creando un nuevo modelo que se almacenará posteriormente en la base de datos.
- **Diseño de las aplicaciones.** Las aplicaciones se diseñan a través de un editor gráfico que genera un modelo como se explica en el apartado 3.2.2. Los modelos se validan para detectar errores y, posteriormente, se genera la aplicación en el servidor usando un generador de código.
- **Panel de control.** Datalyzer incorpora un panel de control personalizado para cada proyecto conectado de forma remota con la aplicación que se ejecuta en el servidor. En este panel, los usuarios pueden realizar distintas acciones como iniciar, pausar o parar la aplicación y otros pequeños elementos de monitorización. Además, también puede recibir los datos de salida que la aplicación genera. Gracias a esta funcionalidad, los usuarios tienen disponibles gráficos para la visualización de datos en tiempo real.

Como hemos visto, Datalyzer no es únicamente una herramienta que facilita el desarrollo de aplicaciones de datos en streaming. Es un entorno de trabajo completo

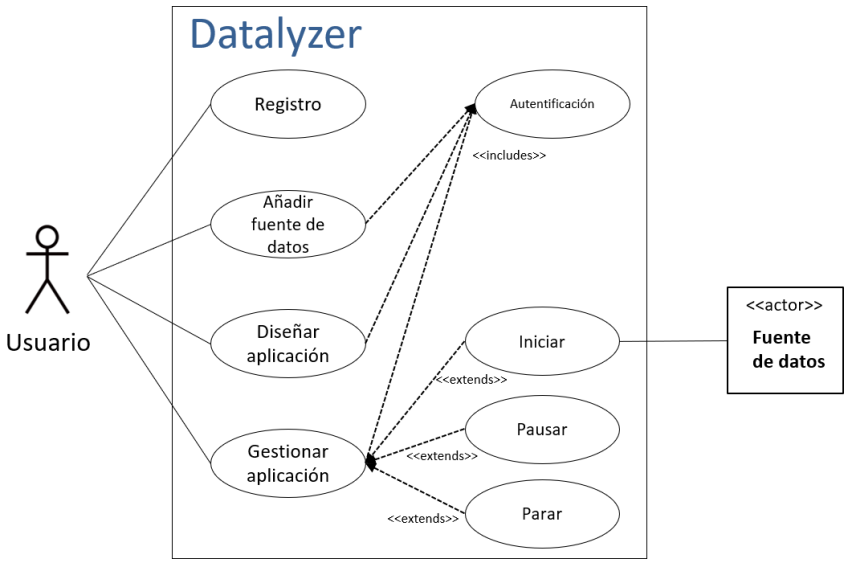

Figura 3.2: Diagrama de casos de uso de Datalyzer

para desarrollar y ejecutar aplicaciones en la nube. Desde procesos ETL, a monitorización mediante gráficos en el panel de control, entre otras. Además, otra posibilidad disponible es el redireccionamiento de los datos de salida a aplicaciones externas. En resumen, Datalyzer se puede usar como un servicio web para la ingesta y transformación de datos heterogéneos, pero también usar las posibilidades que ofrece el panel de control para desarrollar aplicaciones completas.

3.2. Diseño de las aplicaciones de datos en streaming

En este apartado se explican los meta-modelos que se han diseñado para describir aplicaciones de datos en streaming, así como también fuentes de datos y procesadores personalizados.

3.2.1. Fuentes de datos

Las fuentes de datos dinámicas son heterogéneas respecto a protocolos, tecnologías o el tipo de acceso a los datos, que típicamente pueden ser basadas en polling (bajo demanda) o pushing (streaming) [28]. Nuestro objetivo es obtener streams de datos de cualquier fuente de datos dinámica, abstrayéndonos de las características técnicas de cada una, ya sea de tipo *polling* o *pushing*. De esta forma, podemos usar cualquier fuente de datos de forma uniforme para las aplicaciones. Para este propósito, hemos diseñado un meta-modelo declarativo para describir fuentes de datos, lo que facilita la incorporación de nuevas fuentes a la herramienta. Se muestra un esquema del meta-modelo en la Figura 3.3.

El meta-modelo distingue entre fuentes de datos, y tipos de fuente de datos. Este

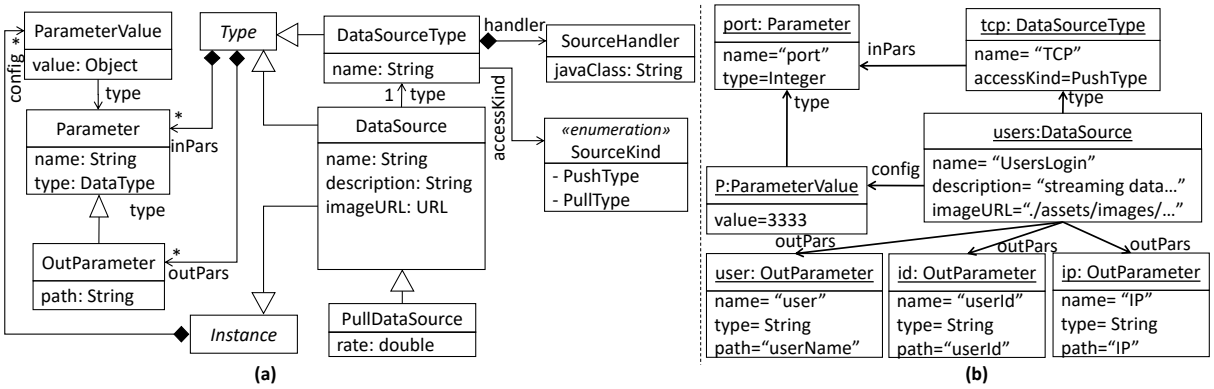

Figura 3.3: (a) Meta-modelo conceptual para definir fuentes de datos dinámicas. (b) Ejemplo de un modelo para una fuente de tipo TCP.

último se refiere a los tipos de tecnologías de transmisión de datos, como REST, Socket o tecnologías no estándar como, por ejemplo, servicios web para la transmisión de mensajes como PubNub[2] o redes sociales como Twitter. Representando tipos de fuentes de datos de forma explícita en el modelo, se garantiza extensibilidad. En este momento, las aplicaciones soportan los tipos: API Rest, TCP Socket, XML, Twitter y RSS. Para añadir nuevos tipos, tenemos que crear un modelo con un objeto de tipo `DataSourceType`. Éste contiene un nombre y una tecnología de acceso, además de un `SourceHandler` que es un fichero java asociado al generador de código. Las fuentes de datos con tecnología de acceso de tipo pushing son adecuadas para las aplicaciones de datos en streaming de forma intrínseca porque generan un stream de datos de forma continua. Por otro lado, el tipo de acceso polling solo genera datos bajo petición. No obstante, podemos simular un stream de datos creando un bucle indefinido que manda peticiones cada cierto tiempo (ratio). Así, podemos trabajar de forma uniforme con cualquier fuente de datos independientemente de la tecnología subyacente. `DataSourceType` puede contener parámetros de entrada y de salida. Por ejemplo, una conexión Socket TCP necesita un **puerto** como parámetro de entrada, mientras que una conexión vía API REST necesita una **URL** o Twitter necesita un **token** privado para la autentificación.

Una vez que se han definido los `DataSourceType` de interés (e.j.: REST), se pueden definir fuentes de datos concretas para cada tipo (e.j.: OpenWeatherMap[3]). `DataSource` contiene un nombre, una descripción y una imagen que se usará en el editor gráfico para construir los modelos de las aplicaciones. La clase `DataSource` hereda de `Type` y de `Instance`. Esto es así, ya que es necesario dar valores a los parametros de entrada previamente definidos (e.j.: la URL para una API REST), así como definir parámetros propios (e.j.: el nombre de una ciudad para OpenWeatherMap).

Los parámetros de salida de la clase `OutParameter` son los datos que reciben las aplicaciones. Tienen un atributo llamado `path` que describe la forma en la que se capturan y parsean los mensajes entrantes. En el caso de que los mensajes se reciban en formato

[2]https://www.pubnub.com/
[3]https://openweathermap.org/api

XML, existe una expresión XPath[4], mientras que los mensajes en JSON tienen una expresión equivalente JSONPath[5].

3.2.2. Aplicaciones de datos en streaming

Como hemos visto anteriormente en el esquema de la Figura 3.1, las aplicaciones se componen de canales de datos que reciben información sobre los que podemos aplicar procesadores (e.j.: transformaciones, filtros) y proporcionan una salida. Éste diseño está formalizado en el meta-modelo que hemos diseñado para describir aplicaciones de datos en streaming en la que se muestra en Figura 3.4.

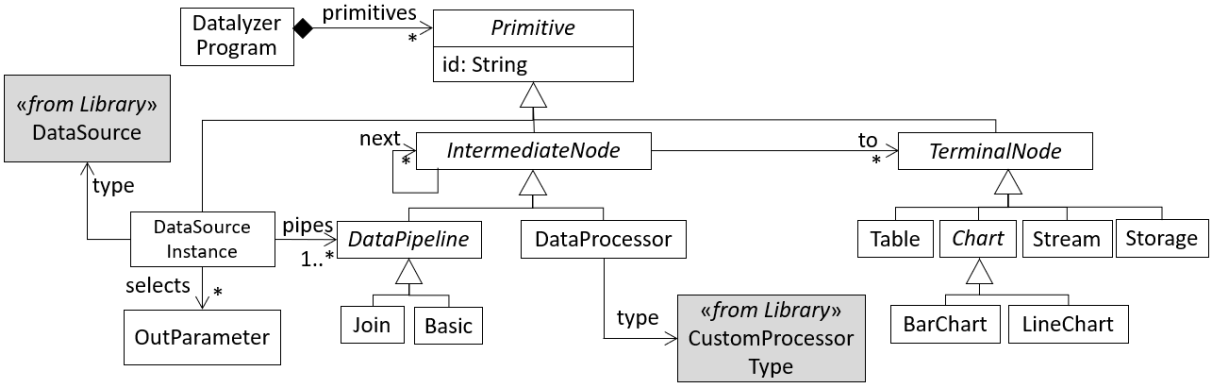

Figura 3.4: Meta-modelo para describir aplicaciones de datos en streaming.

En Datalyzer, las aplicaciones se crean instanciando y conectando diferente tipos de primitivas. La clase `DataSourceInstance` representa una instancia de una fuente de datos concreta previamente definida. Los modelos que describen las fuentes de datos (apartado 3.3.1) se almacenan en una base de datos, y este meta-modelo recibe esa información en forma de librería de clases. Cuando una instancia se añade al modelo, el usuario necesita configurar los parámetros que tenga asociado. Por ejemplo, una instancia del tipo Twitter requiere especificar una palabra clave, localización geográfica (opcional) y la autentificación. El usuario también necesita seleccionar los datos de salida que desea capturar, así como el nombre de usuario, la fecha y el texto del Tweet.

Un objeto de tipo `DataSourceInstance` se puede conectar a uno o varios `DataPipeline`, que representan canales de datos. Nuestro enfoque soporta dos tipos distintos que se han implementado siguiendo los principios del framework Apache Beam[6]. `Basic` es un canal de datos simple que trata cada dato entrante de forma individual, bajo un modelo de clave-valor siendo la clave la procedencia del dato (fuente) y valor, el dato en si mismo. `Join` es un canal de datos que sirve para crear un modelo en base a los datos entrantes, gracias a su capacidad para combinar datos provenientes de distintas fuentes. Su funcionamiento está basado en una tupla de datos con una única ventana temporal

[4]https://www.w3.org/TR/1999/REC-xpath-19991116/
[5]https://www.baeldung.com/guide-to-jayway-jsonpath
[6]https://beam.apache.org/documentation/pipelines/design-your-pipeline/

[29]. Por ejemplo, si tenemos una instancia de Twitter conectada y otra de OpenWeatherMap podemos crear un modelo que contenga un tweet generado bajo el hastag #madrid y la tempetura de Madrid. La Figura 3.5 muestra de forma gráfica un esquema del funcionamiento de ambos tipos de canales de datos. A la izquierda se muestra un canal de tipo **Basic** que recibe 4 campos proveniente de 2 fuentes de datos distintas. Dichos campos se insertan dentro del canal de datos y se extraen, uno a uno, en orden. A la derecha se muestra un canal de tipo **Join** que recibe los mismos datos, pero que genera de salida una estructura que contiene los 4 datos recibidos.

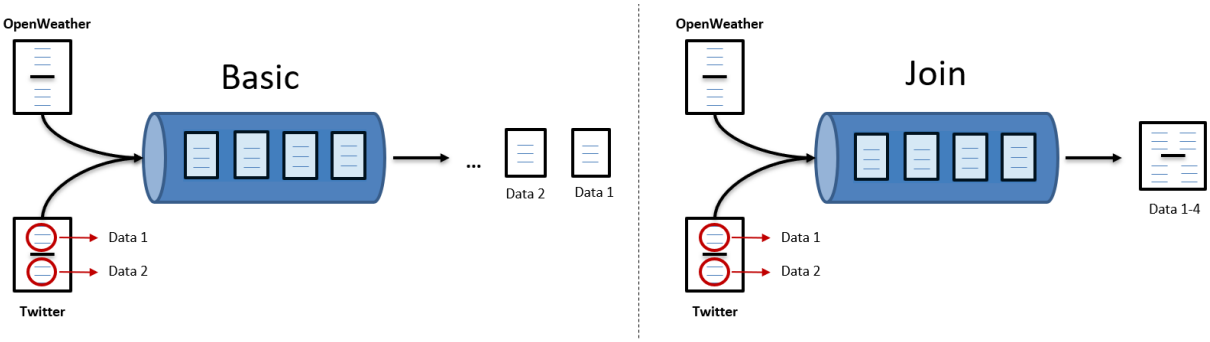

Figura 3.5: Esquema de los tipos de canales `Basic` y `Join`.

Los canales de datos se pueden conectar a uno o varios `DataProcessor` que representan procesadores de datos, como decodificadores, transformadores o filtros (apartado 3.2.3). De la misma forma que ocurre con la clase `DataSourceInstance`, los modelos de los procesadores están almacenados en una base de datos y actúan como una librería externa. Gracias a la librería, los procesadores no están explícitamente integrados en nuestro meta-modelo a través de una serie de sub-clases lo que supondría diseñar un lenguaje cerrado. En cambio, la librería se puede extender con nuevos procesadores sin modificar el meta-modelo de las aplicaciones.

Las clases `DataPipeline` y `DataProcessor` son `IntermediateNode` y por tanto, deben estar conectados con uno o varios `TerminalNode` que representan nodos terminales de las aplicaciones con funcionalidad concreta (e.j.: guardar datos en logs, stream de salida). Con estas clases se configura también el panel de control de las aplicaciones. Cada subclase que hereda de `TerminalNode` representa un widget distinto que permite, por ejemplo, visualizar los datos en una tabla con `Table` o visualizarlos con diferentes tipos de gráficos (`BarChart` y `LineChart`).

3.2.3. Procesadores de datos personalizados

Los procesadores de datos personalizados permiten a los usuarios aplicar transformaciones sobre los datos según los requisitos de cada aplicación. Siguiendo un enfoque similar a las fuentes de datos, los distintos tipos de procesadores no están contenidos de forma explícita en el meta-modelo de las aplicaciones (Figura 3.4). En su lugar, se ha diseñado un punto de extensión que permite a los usuarios desarrollar nuevos procesadores y que éstos actúen como una librería. La semántica de los procesadores se define mediante una

clase en código Java, cuya estructura se ajusta al meta-modelo de la Figura 3.6. Los procesadores se desarrollan de forma genérica, y por tanto se pueden instanciar en cuantas aplicaciones se desee.

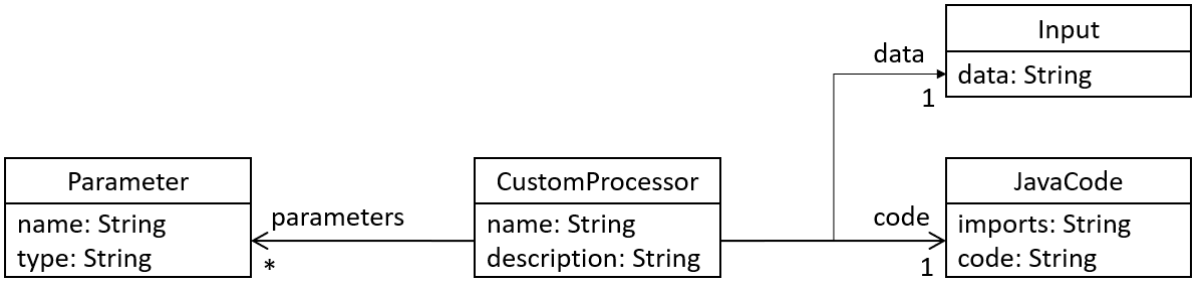

Figura 3.6: Meta-modelo para los procesadores de datos personalizados.

La clase `CustomProcessor` contiene un nombre y una descripción sobre su funcionalidad. Tiene asociado un `JavaCode` que contiene el código Java que el usuario ha desarrollado incluyendo los *imports* de las librerías utilizadas. Además, con esta propuesta los procesadores pueden contener parámetros mediante la clase `Parameter`. Los parámetros tienen un tipo y un nombre, y el valor lo establece el usuario en el DSL mientras diseña la aplicación, de la misma forma que hace con los parámetros de las fuentes de datos. Finalmente, el procesador recibe como dato entrante un `Input`. El dato se recibe en un String proveniente de un canal de datos al que está conectado. En este momento no hay soporte para procesadores múltiples, es decir, que reciban múltiples datos de entrada, por lo que solo recibe un único dato de forma simultanea.

Como se explica en el apartado 3.3.3, este meta-modelo se instancia haciendo ingeniería inversa: el usuario desarrolla una clase Java, y nuestra herramienta extrae de la misma toda la información necesaria para crear un modelo conforme al meta-modelo.

3.3. Creación de modelos

En el anterior apartado hemos estudiado los meta-modelos que hemos diseñado para describir fuentes de datos, aplicaciones y procesadores. En este apartado se detalla el proceso que sigue un usuario para crear modelos usando nuestra herramienta.

3.3.1. Fuentes de datos

Como ya hemos visto anteriormente, las fuentes de datos de interés se deben definir antes de diseñar las aplicaciones. Los modelos se guardan en una base de datos, y están disponibles para instanciarse en cuantas aplicaciones se requiera. Actualmente, la base de datos actúa como una librería, que es de dominio público. Es decir, todos los usuarios tienen acceso de lectura y escritura. Como trabajo futuro, se podría añadir también librerías privadas para cada usuario.

El usuario puede crear nuevos modelos de forma sencilla usando una plantilla. Dicha plantilla contiene toda la información que debe contener el modelo, y el usuario tendrá que rellenar los campos con la información precisa. Se muestra una captura de un modelo a modo de ejemplo en la Figura 3.7. Una vez finalizado el modelo, se envía y se almacena en la base de datos.

Figura 3.7: Creación de un modelo para una API Rest.

Ejemplo. Creamos un modelo para la API de OpenWheaterMap, que se muestra en la Figura 3.7. La información requerida se obtendrá de su documentación[7]. El nombre y la descripción son a elección del usuario, mientras la `url` de la API la obtendremos en la documentación. Como autentificación requiere un APPID personal, por tanto el usuario tendrá que registrarse para obtener dicho APPID. `Type` debe ser *APIRest* mientras que `protocol` es *http://*. Como parámetros de la API configuramos dos: el nombre de la ciudad de la que queremos obtener la temperatura (parámetro requerido), y la unidad métrica en la que devolverá los valores (parámetro opcional). Como parámetros de salida (los datos que devuelve la API) solo configuramos para obtener la temperatura, dato que lo obtendremos con el path *main.temp*. También podríamos configurar más valores, como la descripción o la intensidad del tiempo, siguiendo el mismo proceso. Por último, debido al tipo de fuente de datos `acessKind` debe valer *PushType* y `rate` *0* (el usuario dará valor a este campo en el DSL, como si fuera un parámetro).

3.3.2. Aplicaciones de datos en streaming

Los modelos de las aplicaciones se crean con un editor gráfico que se ha desarrollado e integrado en la plataforma web. Se muestra una captura del editor en la Figura 3.8. A la izquierda se encuentra la paleta de objetos que podemos usar en las aplicaciones, cada una representada con una imagen. Los objetos corresponden con las clases del meta-modelo de la Figura 3.4 incluyendo las librerías de las fuentes de datos y procesadores. En el centro se encuentra el area de trabajo, donde se diseña la aplicación a través de objetos conectados entre sí. El editor funciona con un sistema de *drag and drop*: haciendo click en

[7] https://openweathermap.org/current

las imágenes, se seleccionan y arrastran al área de trabajo. También se pueden seleccionar los objetos añadidos al área de trabajo, donde se despliega un panel de configuración a la derecha. Aquí el usuario podrá dar valores a los parámetros, seleccionar los datos de interés, añadir un nombre a los objetos y cualquier otra configuración que tenga asociada cada objeto. Finalmente, los objetos se pueden conectar entre sí mediante enlaces, que representan flujos de datos. Para facilitar la tarea al usuario, existe un control de tipos, por lo que no se puede conectar una entrada con otra entrada, o una fuente de datos con un nodo terminal. También se pueden borrar objetos añadidos al proyecto.

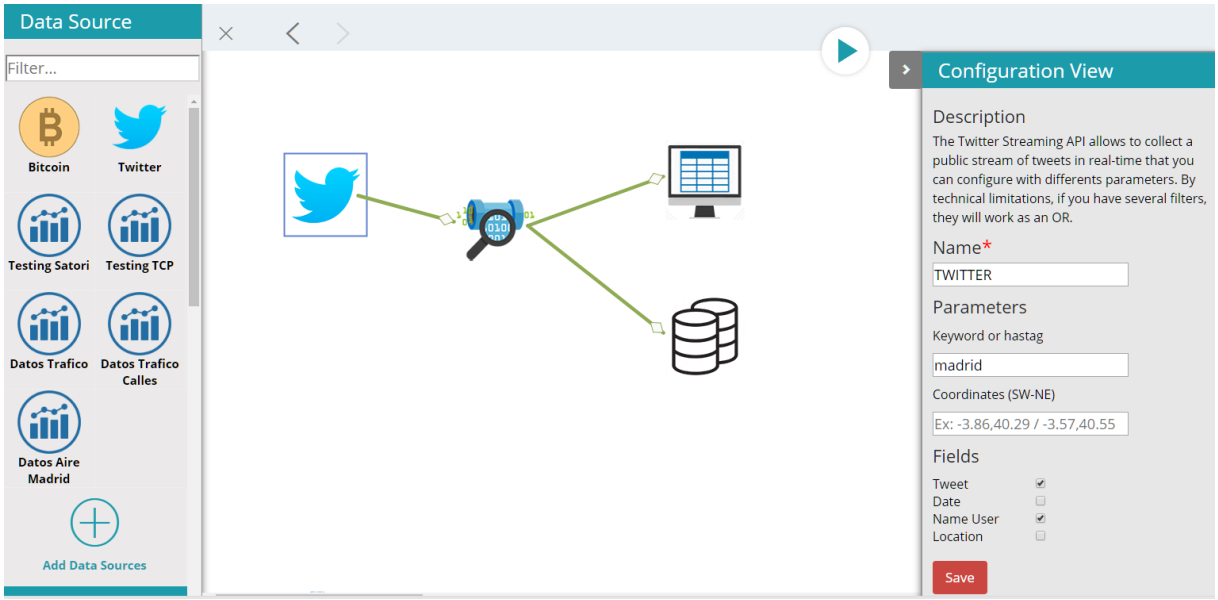

Figura 3.8: Creación de un modelo para una API Rest.

Cuando el usuario ha terminado de diseñar la aplicación, debe pulsar en el botón *play* (arriba a la derecha). Automáticamente se lanza un validador del modelo, que checkea que sea válido conforme al meta-modelo de las aplicaciones, y no tenga campos vacíos que sean requeridos. En un nuevo panel que se despliega en la parte inferior, se informa al usuario si se ha encontrado algún error. En el caso de que sea un modelo correcto, también se informa al usuario y se guarda en la base de datos. Gracias a esta funcionalidad, el proyecto se podrá abrir y editar de nuevo en el futuro si se desea.

Ejemplo. La Figura 3.8 muestra una captura del diseño de una sencilla aplicación con una fuente y un canal de datos. En concreto, se usa Twitter como fuente de datos para recopilar los tweets que se generan bajo el keyword *madrid*. Se seleccionan como datos de interés el texto del tweet y el nombre del usuario que lo ha escrito. El flujo de datos se conecta con un canal de datos, que a su vez está conectado a dos nodos terminales: un nodo para mostrar los datos generados en una tabla en el panel de control, y otro para almacenarlos en un fichero.

3.3.3. Procesadores de datos personalizados

Los procesadores de datos personalizados son un punto de extensión dónde los usuarios pueden añadir sus propios transformadores de datos en código Java. Para ello, se creará un modelo del procesador haciendo ingeniería inversa: el usuario desarrolla una clase JAVA, y el modelo se infiere de forma automática a partir de éste. Posteriormente, el modelo se envía a la base de datos. De igual forma que ocurre con los modelos de las fuentes de datos, los procesadores actúan como una librería pública. Es decir, una vez desarrollados se pueden instanciar en cuantas aplicaciones se requiera. Se muestra un esquema de este proceso en la Figura 3.9.

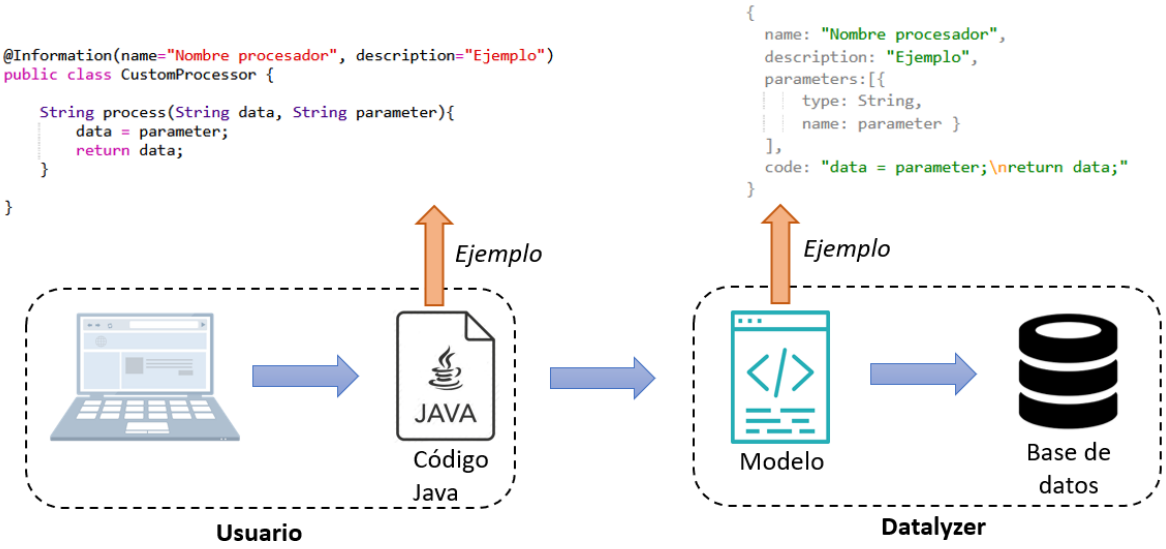

Figura 3.9: Creación de un modelo para una API Rest.

El usuario tiene que desarrollar una clase que se llama `CustomProcessor`. A través de notaciones, debe indicar un nombre y una descripción sobre la funcionalidad que realiza el procesador. Posteriormente, hay que implementar un método llamado `process` que puede tener de forma opcional parámetros de entrada. Con el propósito de hacer los procesadores genéricos, a los parámetros se les asigna un valor específico en el DSL (apartado 3.3.2), de forma similar que ocuure con los parámetros de entrada de las fuentes de datos u otras primitivas. De esta forma, podemos tener un procesador genérico (e.j. un filtro) que se puede configurar posteriormente dependiendo de los requisitos.

3.3.4. Conclusiones

En este capítulo hemos explicado el enfoque y las características principales que tiene la herramienta que hemos desarrollado. Se puede concluir que Datalyzer es una herramienta web que unifica el proceso de diseño y creación de las aplicaciones, y el proceso de despliegue e infraestructura. Posteriormente, se ha descrito en profundidad los meta-modelos que hemos diseñado para describir aplicaciones de datos en streaming, fuentes de datos y procesadores. Finalmente, se muestra el proceso de creación de modelos

conforme a los meta-modelos propuestos, así como un ejemplo de cada uno de ellos. En el siguiente capítulo se explicará con más detalle el diseño técnico y la arquitectura del sistema desarrollado, así como la tecnología empleada.

4

Arquitectura y herramienta

En este capítulo se explicará la arquitectura de Datalyzer (apartado 4.1), profundizando en cada uno de sus componentes y cómo ha sido desarrollado. En el apartado 4.2 se describe la herramienta y el front-end de la plataforma web.

4.1. Arquitectura

Datalyzer se ha diseñado y desarrollado con una arquitectura modular a través de componentes que se comunican entre sí. La Figura 4.1 muestra la arquitectura del sistema desarrollado. Podemos diferenciar entre dos bloques: los componentes back-end que se ejecutan en el servidor y los componentes front-end que se ejecutan en el lado del cliente. Gracias a este diseño podemos actualizar, por ejemplo, el generador de código sin alterar ningún otro componente. Además, la tecnología y el lenguaje de programación que se ha usado en cada componente es independiente del resto. A continuación se explicará la funcionalidad de cada componente y cómo ha sido desarrollado.

4.1.1. Plataforma web y base de datos

Datalyzer utiliza una base de datos con distintos propósitos. En concreto, tenemos dos componentes (1 y 2 en la Figura 4.1) que usan una base de datos MongoDB para almacenar información. MongoDB[1] es una base de datos no relacional orientada a guardar documentos en formato JSON. Los documentos se guardan en colecciones, que

[1] https://www.mongodb.com/

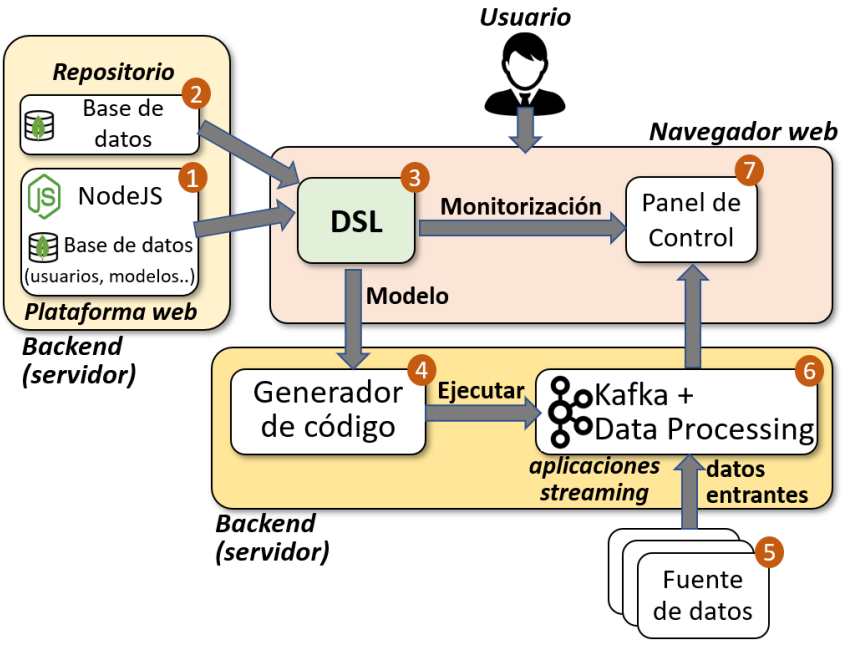

Figura 4.1: Arquitectura de Datalyzer y cada uno de sus componentes

son similares a una tabla en una base de datos relacional, pero no es necesario que los documentos que se almacenan tengan la misma estructura.

El back-end de la plataforma web está desarrollado con el framework NodeJS[2] que es un motor de ejecución Javascript para el lado del servidor y HandlebarsJS[3] como motor de plantillas web que funciona con un Modelo-Vista-Controlador (MVC). La plataforma cuenta con un sistema de registro de usuarios, que incluye: nombre completo, mail y contraseña. Esta información se guarda en la base de datos en una colección específica para guardar usuarios. La base de datos también tiene otra colección donde se guardan los proyectos que crea cada usuario. Por cada proyecto se almacena su propietario, propiedades como el nombre y descripción, y el modelo de la aplicación que genera el DSL. Debido a esta persistencia del modelo, la aplicación se puede guardar y modificar cuando se desee.

Los modelos de las fuentes de datos y los procesadores también se almacenan en sus propias colecciones, con la diferencia de que no tienen propietario y actúan como un repositorio para las aplicaciones. Esto es, que cualquier usuario puede usarlos. Dichos modelos se cargan cada vez que un usuario abre el DSL en el navegador: el servidor consulta la base de datos y envía los modelos al cliente. Con este proceso se garantiza que cada vez que un usuario abre el DSL esté actualizado, tanto con el propio proyecto como con las librerías.

[2]https://nodejs.org/es/
[3]https://handlebarsjs.com/

4.1.2. DSL

El DSL (componente 3 en la Figura 4.1) es una aplicación web desarrollada en Javascript a partir de un fork del proyecto open-source BrainBox[4]. La aplicación tiene un diseño de programación orientado a objetos gracias a las posibilidades que ofrece el último estándar de Javascript ECMAScript 6 (ES6) [30]. Cada clase del meta-modelo de la Figura 3.4 tiene su propio objeto con la particularidad de `DataSourceInstance` y `DataProcessor`. Cuando se carga el DSL en el navegador, recibe todos los modelos de las fuentes de datos y procesadores. A partir de los modelos, se crean objetos para cada uno de ellos de forma dinámica. Como resultado, obtenemos un conjunto de objetos que representan toda la funcionalidad disponible para las aplicaciones y que se pueden visualizar de forma gráfica con una imagen (apartado 3.3.2).

El modelo de la aplicación se genera en tiempo real conforme el usuario realiza acciones sobre la aplicación, como añadir un nuevo objeto o una conexión. El modelo de la aplicación tiene un formato de tipo JSON, de igual forma que ocurre con el resto de modelos, porque este formato es adecuado para manipularlo en Javascript y posteriormente almacenarlo en la base de datos. Finalmente, la propia aplicación valida el modelo mediante comprobaciones sobre el JSON generado tales como campos no nulos o parámetros requeridos. Una vez que el usuario finaliza el proceso de creación y validación, el modelo se envía al servidor. Este proceso está ejemplificado en la Figura 4.2.

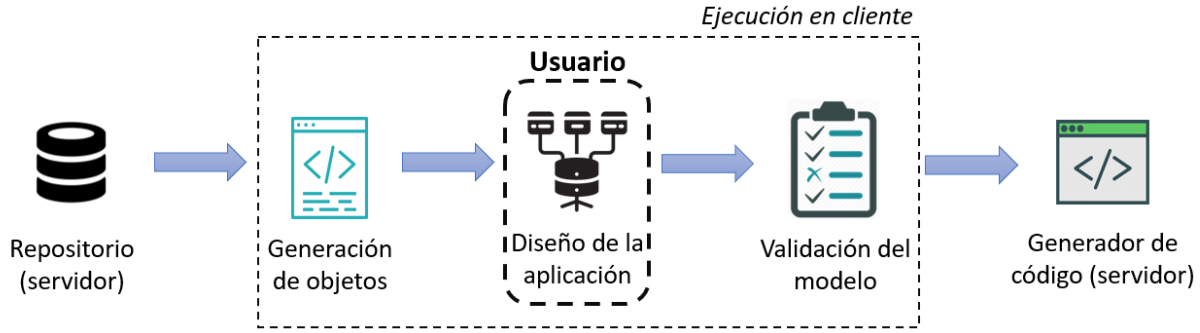

Figura 4.2: Proceso para la generación de una aplicación.

4.1.3. Generador de código

El generador de código (componente 4 en la Figura 4.1) se encarga de transformar los modelos de las aplicaciones en código fuente que posteriormente se puede compilar y ejecutar. Por lo tanto, el generador de código es un componente que recibe un modelo como entrada y genera varios archivos de salida. El modelo de entrada tiene que tener un formato JSON y, además, se asume que está correctamente formado para evitar errores de ejecución. Los archivos de salida son ficheros Java que en su conjunto forma una aplicación que se explica más en detalle en el apartado 4.1.6. La aplicación se almacena en una carpeta de forma interna dentro de Datalyzer.

[4]https://freegroup.github.io/brainbox/

El generador está desarrollado en lenguaje Java sin usar ningún framework específico para la generación de código. Actúa como un *parser*: recorre las clases internas del modelo una a una para generar el código fuente a través de plantillas. Una plantilla es una clase Java con una estructura definida. En concreto, todas las plantillas tiene un método llamado *generateCode* que devuelve en un string el código de una clase Java. Posteriormente, se guarda el string como un fichero con un nombre y ruta específica.

Cada clase del meta-modelo de la Figura 3.6 tiene su propia plantilla, por ejemplo, una plantilla para crear un canal de datos o generar un archivo log. Además, como podemos ver en el meta-modelo de la Figura 3.4, cada tipo de fuente de datos también tiene asociado una plantilla. Como todas las fuentes de datos del mismo tipo tienen un funcionamiento similar, solo es necesario tener una plantilla para cada tipo.

Como se ha mencionado, una plantilla es una clase que de forma interna genera código Java y lo guarda en un fichero. Por ejemplo, una plantilla asociada al tipo de datos Socket TCP genera una clase Java que crea un Socket TCP en un puerto específico. Una plantilla asociada un canal de datos, genera una clase Java que crea un *topic* en Kafka y su correspondiente configuración. Para que la plantilla tenga un óptimo funcionamiento, tendrá que estar preparada para poder ajustarse según la configuración que recibida del modelo (e.j. autentificación en una API Rest) como parámetros.

La Figura 4.3 muestra de forma gráfica el funcionamiento interno del generador de código. Todo el proceso está automatizado a través de scripts, que tiene versiones tanto en Linux como en Windows para que Datalyzer se pueda desplegar en ambos sistemas.

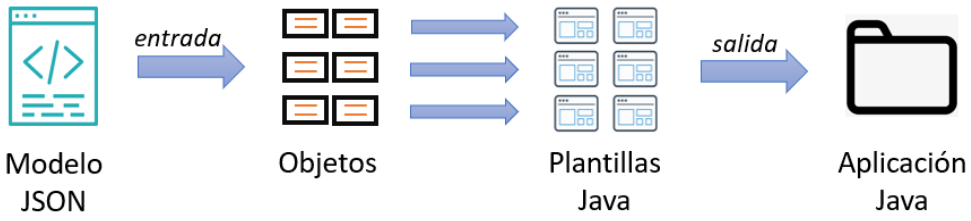

Figura 4.3: Proceso para la generación de una aplicación.

4.1.4. Aplicaciones de datos en streaming

Las aplicaciones de datos en streaming que se crean en Datalyzer (componente 5 y 6 en la Figura 4.1) son aplicaciones Java basadas en el proyecto open-source Kafka [26]. Kafka es un sistema de alto rendimiento que proporciona manipulación de datos en tiempo real, con una baja latencia y en un entorno distribuido. Además, Kafka es tolerante a fallos gracias a usar a nivel interno particionamiento y replicación. Con todas estas características, y según los requisitos que hemos estudiado en el apartado 2.3, podemos afirmar que Kafka es una tecnología adecuada para el desarrollo de aplicaciones de datos en streaming.

El diseño de las aplicaciones desarrolladas se basa en la arquitectura del propio sistema de Kafka que funciona con un proceso de publicación/subscriptor. La Figura 4.4 muestra

el diseño de la aplicación y el flujo que siguen los datos desde su entrada al sistema hasta su salida.

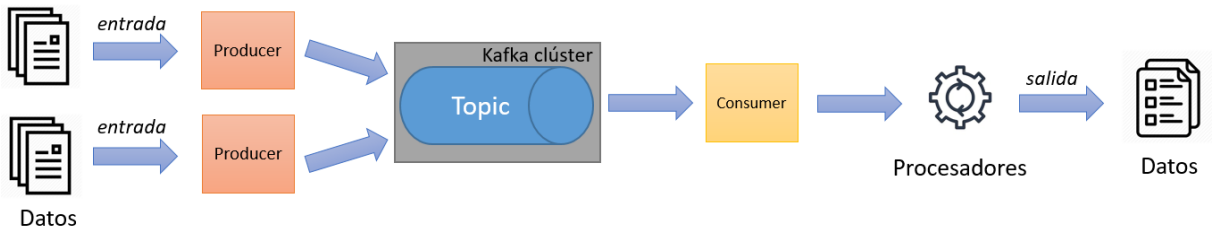

Figura 4.4: Proceso para la generación de una aplicación.

Las aplicaciones tienen tres tipos de clases Java principales: `producer`, `topic` y `consumer`. La clase `producer` es el módulo de ingesta de datos. Cada clase de este tipo está conectada a una única fuente de datos de la que recibe información. Por tanto, la aplicación contendrá un `producer` por cada fuente de datos que esté conectada. Por ejemplo, una clase `producer` para un socket TCP tendría incorporado las siguientes tareas: crear el socket con un puerto en específico y quedarse a la espera de conexiones entrantes. Cuando se reciben datos, se aplica una normalización de los mismos. Este proceso sirve para hacer una limpieza de los datos de interés y los que desechamos, y posteriormente crear un modelo de datos en un formato estándar. De esta forma, es indiferente que una clase `producer` ingeste datos en formato JSON, XML o cualquier otro porque al final del proceso obtendremos de todos ellos un modelo de datos clave-valor en formato JSON. Una vez tenemos el modelo listo, lo insertamos en un canal de datos a través de un `topic`. Una clase `producer` puede insertar datos en uno o más `topic`.

La clase `topic` es una clase interna de Kafka que sirve para crear un canal de datos. Una aplicación tendrá tantos `topic` como canales de datos se haya diseñado. Todo el manejo de los datos a través del canal, como la recepción, almacenamiento, replicación y distribución de los mensajes es transparente para el programador. De todas estas tareas se encarga el clúster de Kafka. El canal de datos funciona como una cola donde un `producer` inserta un dato, y el dato se almacena en orden de recepción hasta que lo extrae una clase `consumer`. Se ha añadido una capa superior de abstracción para implementar dos tipos de `topic` distintos para hacer agregaciones de datos como se explica en el apartado 3.2.2.

Finalmente, la clase `consumer` es la clase que extrae los mensajes de un canal de datos y los redirecciona a una salida, típicamente el panel de control. Una aplicación tiene `consumer` tantos `topic` contenga. Esta clase también redirecciona los datos a un procesador que esté asignado al canal de datos. A diferencia de las clases `producer`, solo puede haber un `consumer` asignado a un `topic` específico.

4.1.5. Panel de control

Como las aplicaciones en Datalyzer se ejecutan de forma interna, ya esté desplegado a nivel local o en un cloud, se ha desarrollado un panel de control (componente 7 en la Figura 4.1) donde los usuarios pueden realizar distintas acciones sobre las mismas

de forma remota como, por ejemplo, iniciar o parar la ejecución. Cada aplicación tiene su propio panel de control personalizado, que se genera de forma dinámica a partir de la configuración de la misma. El panel de control pertenece a la parte front-end de la plataforma web, y por tanto, se ejecuta en el navegador del usuario. A nivel técnico, está íntegramente desarrollado en Javascript.

El panel de control está compuesto de pequeños elementos llamados *widgets*. Cada uno de ellos ha sido desarrollado de forma individual, y realizan alguna acción concreta sobre los datos. Por ejemplo, podemos encontrar *widgets* para visualizar datos en una tabla, en un gráfico, un botón para descargar un log asociado y más. Todos estos *widgets* representan la funcionalidad que podemos añadir a las aplicaciones mientras la diseñamos en el DSL.

La comunicación entre las aplicaciones y el panel de control se realiza mediante una conexión socket. Cuando el usuario pulsa en un botón, se envía por el socket un comando que la aplicación recibe y parsea para, posteriormente, realizar dicha acción. Los datos que se generan también se envían al panel de control por el socket.

4.1.6. Despliegue

Todos los componentes que se han explicado anteriormente están encapsulados en un contenedor de Docker [31]. Docker es una tecnología de virtualización que permite empaquetar software en unidades llamadas contenedores que incluyen todos los elementos necesarios para ejecutar dicho software, incluido código, librerías o herramientas. Los contenedores de Docker son portables y se pueden ejecutar en cualquier sistema operativo y maquina en la que Docker esté instalado.

Como resultado, se puede desplegar Datalyzer fácilmente en cualquier ordenador que tenga instalado Docker ejecutando el contenedor que se ha preparado. Cuando ejecutamos el contenedor, de forma automática se crea un servidor web por el puerto 8080. Si accedemos con el navegador a dicho puerto, entraremos en la plataforma de Datalyzer. No se requiere configuración, y este proceso de despliegue es válido para ejecutarlo tanto de forma local como en una infraestructura cloud.

4.2. Herramienta

En este apartado se explicará el front-end de la plataforma web y cómo los usuarios interactúan con Datalyzer.

4.2.1. Plataforma web

Como ya se ha explicado anteriormente, Datalyzer es una plataforma web y, por tanto, la interacción con el usuario se realiza totalmente a través del navegador. Cuando se accede

a la web, en primer lugar el usuario verá un formulario para registrarse en la plataforma o para hacer iniciar sesión. Se muestra una captura en las imágenes 1 y 2 de la Figura 4.5.

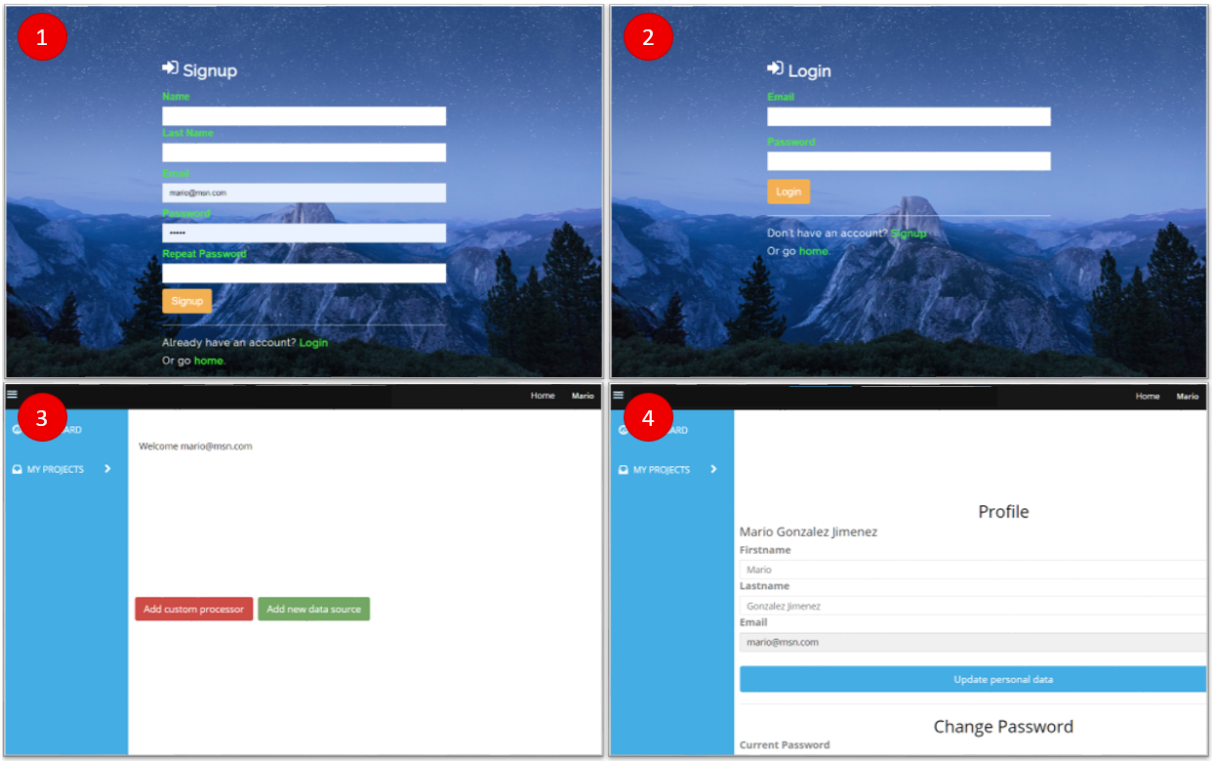

Figura 4.5: Capturas de la plataforma web de Datalyzer.

Una vez el usuario accede a la plataforma con su cuenta, se carga en panel principal que se muestra en la imagen 3 de la Figura 4.5. En esta pantalla el usuario puede acceder a su espacio personal a través del menú, a su perfil o el acceso para añadir nuevos modelos de fuentes de datos y procesadores. Finalmente, en la imagen 4 se muestra la pantalla con el perfil del usuario, donde se visualiza y se pueden modificar algunos datos como la contraseña y el nombre.

4.2.2. Espacio personal

Cada usuario tiene un espacio personal para crear, listar y editar sus proyectos. En la imagen 1 de la Figura 4.6 se muestra el panel donde se puede ver todos los proyectos que tienen asignado. Cada proyecto tiene un nombre, descripción, fecha de creación y fecha de modificación. También hay un botón que redirecciona a la pantalla para crear un nuevo proyecto.

La imagen 2 de la Figura 4.6 muestra el panel que tiene asociado cada proyecto. En este panel se puede visualizar información como el nombre, la descripción y el propietario. También tenemos acceso al editor gráfico (DSL), borrar el proyecto o abrir el panel de control.

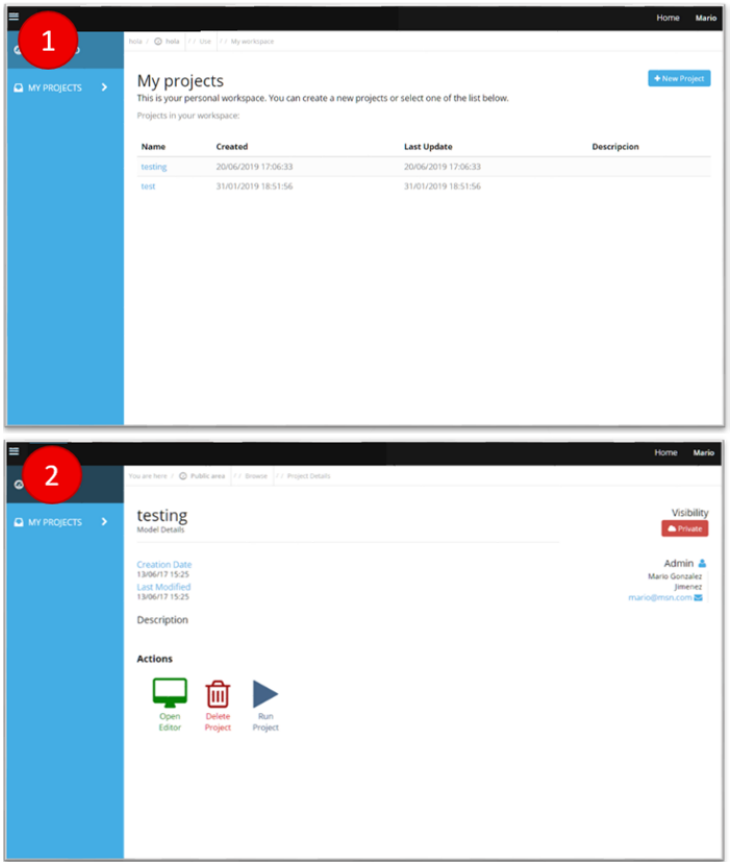

Figura 4.6: Capturas del espacio personal de un usuario.

4.2.3. Añadir nuevas fuentes de datos

Un usuario puede añadir un modelo de una fuente de datos al repositorio de Datalyzer siguiendo el proceso que se explica a continuación. El modelo tiene un formato JSON que contiene toda la información del meta-modelo de la Figura 3.3. Para facilitar la creación del modelo al usuario, se usa la herramienta ObjGen[5] que permite construir objetos de tipo JSON con una sintaxis más intuitiva y amigable. Se proporciona al usuario una plantilla que tiene que rellenar con la información correspondiente a la fuente de datos. El modelo se transforma a formato JSON de forma automática. En la imagen 1 de la Figura 4.7 podemos observar una captura de la plantilla y el modelo JSON. Se puede encontrar un ejemplo de creación de un modelo en el apartado 3.3.1.

Una vez se han rellenado todos los campos de la plantilla, se tiene que copiar el modelo e insertarlo en el formulario de Datalyzer (imagen 2 de la Figura 4.7). Pulsando el botón *enviar*, el servidor recibe el modelo, valida que tiene toda la información necesaria y lo añade al repositorio.

[5]http://www.objgen.com/json/models/J6j

Figura 4.7: Creación de un modelo de una fuente de datos usando ObjGen.

4.2.4. Añadir nuevos procesadores personalizados

Un usuario puede añadir un nuevo procesador al repositorio de Datalyzer desarrollando una clase Java dentro de la propia plataforma. Para implementar esta funcionalidad en Datalyzer se utiliza la herramienta Repl.it[6] que es un IDE de Java online gratuito. La clase que el usuario tiene que desarrollar se llama `CustomProcessor` que cuenta con dos notaciones: nombre y descripción. Además, la clase tiene un método llamado `process` donde el usuario puede añadir su código. Este método tiene de forma obligatoria un argumento `data` de tipo String donde se recibe el valor serializado que pasa por el canal de datos al que está conectado el procesador. De forma opcional, se pueden añadir mas argumentos del tipo que se deseen que se convertirán posteriormente en parámetros al hacer la conversión código-modelo. El método devuelve una variable de tipo string que sustituirá en el canal de datos al valor que entra al procesador. También se puede retornar un valor *null*, en cuyo caso áctua como un filtro. Es decir, no se vuelve a introducir ningún valor en el canal de datos para sustituir el valor de entrada. Se muestra una captura de la herramienta Relp.it y la clase `CustomProcessor` en la Figura 4.8.

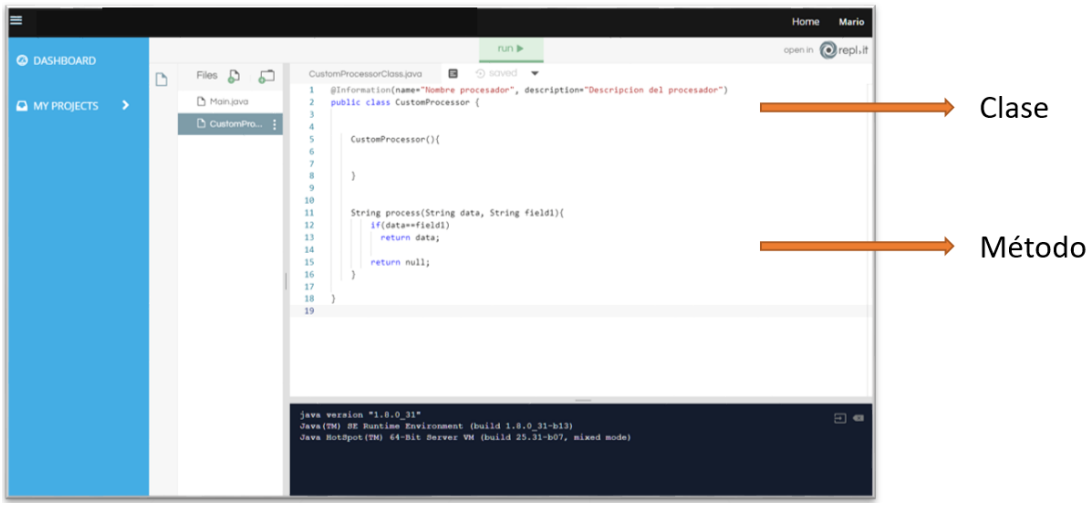

Figura 4.8: Creación de un modelo de un procesador usando Repl.it.

[6]https://repl.it/

Finalmente, el usuario tiene que pulsar en el botón *run* para ejecutar el proyecto Java en Relp.it. Se incluye una clase *main* que de forma automática extrae el modelo del procesador a partir de la clase desarrollada y, posteriormente, envía el modelo al repositorio de Datalyzer para su almacenamiento.

Una vez que el modelo está almacenado en el repositorio, el procesador se puede instanciar en el DSL. La Figura 4.9 muestra una captura de una instancia de un procesador aplicado sobre un canal de datos. Para instanciar un procesador sobre un canal de datos hay que añadir al area de trabajo un objeto de tipo `CustomProcessor`. Este objeto es genérico y permite seleccionar, mediante un despegable, el procesador que queremos usar. Además, se muestra la descripción, así como los parámetros y el campo de entrada del mismo.

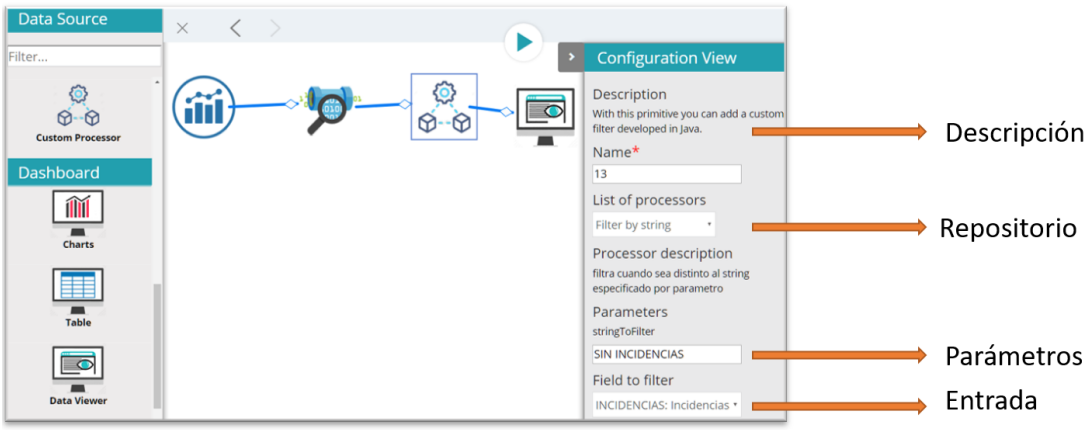

Figura 4.9: Ejemplo de un procesador en el DSL.

4.2.5. DSL

Como ya se ha mencionado, el DSL es un editor gráfico que funciona con un simple mecanismo de *drag-and-drop*. A la izquierda del editor, se encuentra el panel de objetos que contiene todas las primitvas que podemos añadir y usar en la aplicación que estamos diseñando como fuentes de datos, canales de datos o procesadores. En el centro tenemos el área de trabajo, donde colocaremos los objetos para configurarlos y posteriormente conectarlos entre sí. Cuando clicamos en un objeto que está añadido al área de trabajo, se despliega a la derecha el panel de configuración donde podemos ajustar algunos parámetros y variables. Por un último, cuando se clica en el botón de *play* se despliega un panel inferior que contiene el validador del modelo. En este panel se informa al usuario si el modelo está correctamente formado o ha encontrado algún error. En el caso de que sea correcto, se ofrece la posibilidad de enviar el modelo al servidor para generar la aplicación. Se muestra una captura en la Figura 4.8. En el apartado 3.3.2 explicamos de forma más detallada el proceso para la generación de un modelo con ejemplo.

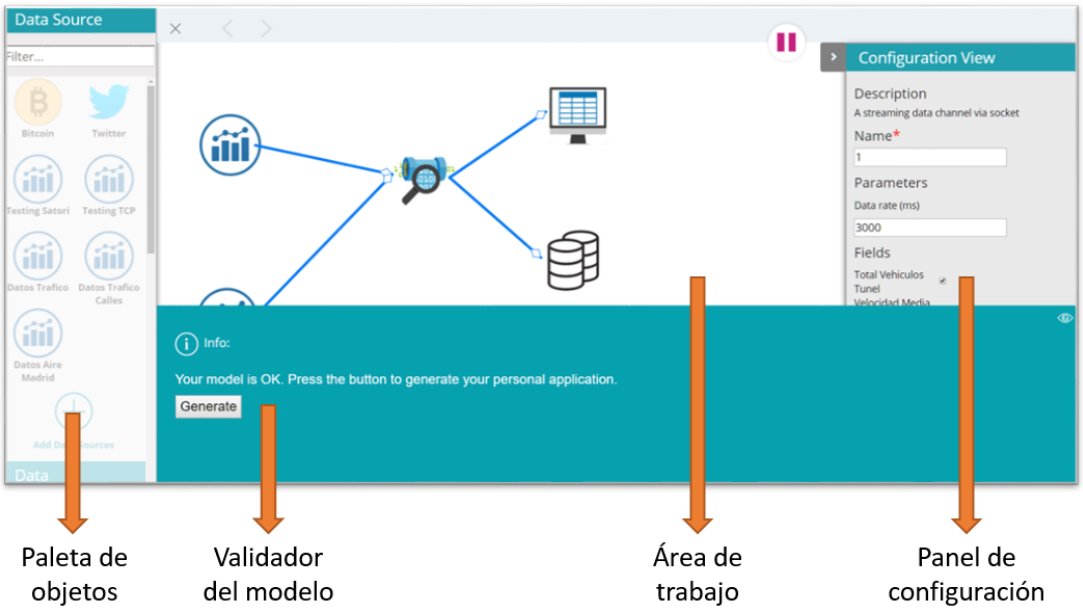

Figura 4.10: Captura del DSL.

4.2.6. Panel de control

El panel de control se genera de forma dinámica para cada proyecto según el usuario lo haya configurado, aunque todos los paneles de control comparten la misma estructura. En la parte superior se encuentran los elementos de monitorización, estos son, botones para ejecutar, pausar y parar la aplicación, así como un indicador del tráfico de red en tiempo real en bytes/segundo. En la parte inferior podemos encontrar los *widgets* que están asociados al proyecto, como tablas, gráficos u otros visualizadores de datos. Cada *widget* está identificado por un título que se le ha asignado previamente. Los *widgets* funcionan de forma asíncrona e independiente, y se refrescan de forma automática cuando se reciben datos. Se muestra una captura del panel de control de una aplicación a modo de ejemplo en la Figura 4.11.

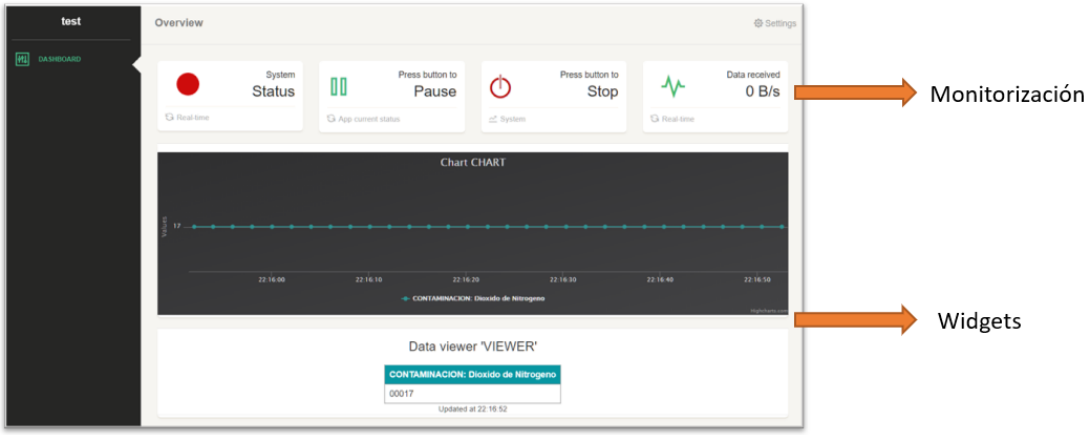

Figura 4.11: Captura del panel de control.

4.2.7. Conclusiones

En este capítulo hemos explicado todos los detalles técnicos de la aplicación que hemos desarrollado. Datalyzer está diseñado con una arquitectura modular que contiene componentes independientes como el generador de código, la plataforma web o la base de datos. Gracias a este diseño, la herramienta es más mantenible y flexible si en el futuro se quiere añadir nuevas características. Por ejemplo, podemos actualizar el generador de código o incluso desarrollarlo de nuevo en otro lenguaje de programación sin necesidad de alterar ningún otro componente, ya que el modelo que recibe como entrada no varía. También podríamos cambiar la tecnología de la base de datos modificando unicamente la configuración de la misma en la plataforma web. Posteriormente, hemos explicado el front-end de los aspectos más importantes de la plataforma web incluyendo capturas de la herramienta desarrollada. En el siguiente capítulo se expondrá la evaluación que se ha realizado sobre Datalyzer, los resultados y las conclusiones.

5 Evaluación

En este capítulo se exponen y discuten los resultados de los experimentos que se han realizado para evaluar la aplicación desarrollada. En el apartado 5.1 se detallan distintas pruebas técnicas que evalúan el rendimiento, estabilidad y escalabilidad del sistema. En el apartado 5.1 se expone un caso de estudio que pone en práctica toda la funcionalidad que se ha desarrollado.

5.1. Experimentos

Los siguientes experimentos se han realizado usando la infraestructura cloud de la Universidad de Potsdam a través del proyecto para investigadores HPI Future SOC Lab. El hardware del equipo utilizado tiene las siguientes características:

- Maquina virtual con Ubuntu 16.04
- 4 cores Intel Xeon @2.4 GHz
- 8GB RAM
- 100GB HDD

A continuación se explica cada experimento de manera individual, las condiciones del mismo y las conclusiones que extraemos de los resultados.

5.1.1. Pérdida de datos

Descripción

El objetivo de este experimento consiste en evaluar si existe pérdida de datos en el sistema, así como calcular la tasa de error. También se analiza si existen anomalías, como mensajes corruptos, incompletos y cualquier otro comportamiento extraño en el sistema.

Condiciones

Se crea una aplicación con un solo canal y fuente de datos. Se recopila un log con los datos de entrada al sistema, y otro log con los datos de salida. Posteriormente, se comparan ambos logs para detectar pérdida de mensajes y anomalías. El experimento se realiza durante 14 minutos en los que se reciben 31835 mensajes, lo que da un tasa de 37 mensajes/segundo.

Resultados

Comparando el log de entrada y el log de salida, obtenemos que la tasa de error es de 1,15 %. Aunque es una tasa de error pequeña, también es mejorable puesto que se espera que no haya pérdida de mensajes. No se detectan anomalías en los mensajes. Los resultados se muestran en la Tabla 5.1.

	Resultado
Pérdida de mensajes	367
Anomalías	0
Tasa de error	1,15 %

Tabla 5.1: Resultados para la pérdida de datos.

5.1.2. Delay

Descripción

El objetivo de la siguiente serie de experimentos consiste en evaluar la estabilidad y la eficiencia del sistema, y detectar si existen cuellos de botella. Para ello se recopila el retardo, también conocido como delay, que tardan los mensajes en ser procesados por Datalyzer a lo largo del tiempo en diferentes escenarios.

Condiciones: un canal de datos

Se diseña un escenario sencillo con una aplicación que tiene un canal y una fuente de datos de tipo Socket TCP. Se repite el experimento dos veces para duplicar el ratio de mensajes recibidos y confrontar el delay a lo largo del tiempo. El experimento tiene una duración de 1 hora.

Resultados

El delay promedio que se ha obtenido es de 281 y 310 milisegundos para el delay lento y el duplicado, respectivamente, como se muestra en la Tabla 5.2. Además, como se aprecia en la Figura 5.1 el delay es constante en el tiempo en ambos casos. Se aprecia que llegan algunos mensajes con retraso, si bien observamos que es un hecho aislado que puede deberse a alguna anomalía en el mensaje recibido.

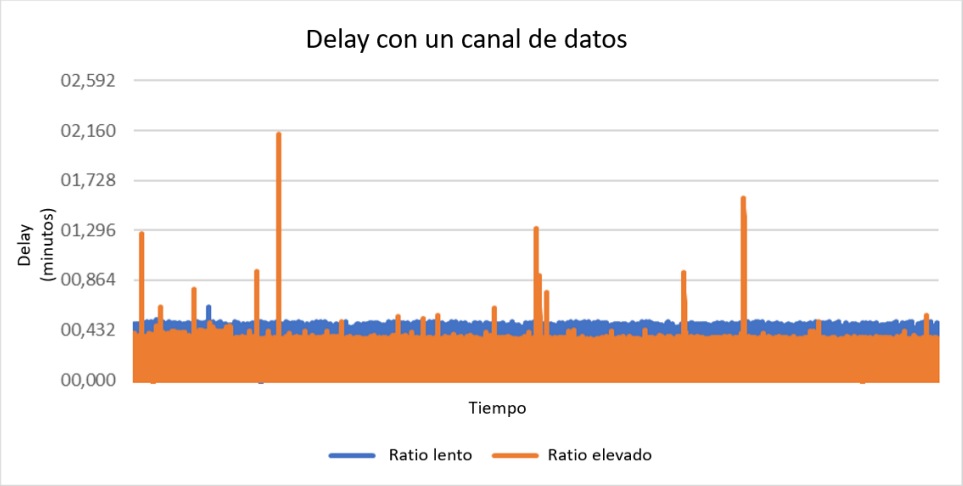

Figura 5.1: Delay con un canal de datos.

	Resultado (segundos)
Promedio delay lento	0.281
Promedio delay rápido	0,310

Tabla 5.2: Delay promedio con un canal de datos.

Condiciones: dos canales de datos

Se mide el delay de una aplicación con dos canales de datos que reciben mensajes con un ratio similar. El objetivo de este experimento es evaluar el comportamiento y rendimiento de ambos canales, así como detectar alguna anomalía. Se recopilan datos durante una hora.

Resultados

Como se aprecia en la Figura 5.2, ambos canales de datos tienen un comportamiento y rendimiento similar, como esperábamos obtener. Además, se ha comprobado que el delay promedio es 271 milisegundos que es similar al experimento anterior. Como se puede ver en la gráfica, se ha recibido una anomalía en un dato procesado por el Canal 1 aunque no consideramos que este error aislado sea relevante.

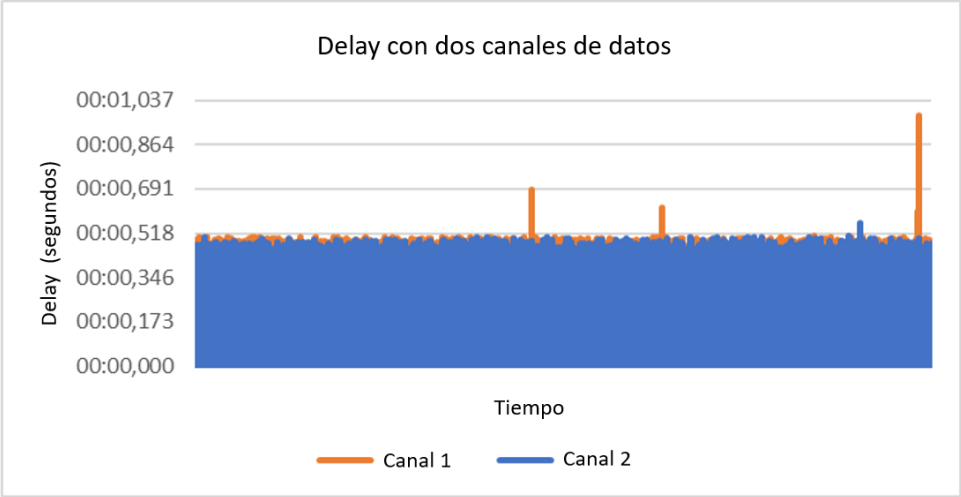

Figura 5.2: Delay con dos canales de datos.

Condiciones: cuatro canales de datos

Se repite el experimento anterior pero con cuatro canales de datos simultáneos. El objetivo de este experimento es comprobar la escalabilidad del sistema, así como detectar anomalías o cuellos de botella.

Resultados

Como se puede observar en la Figura 5.3, el rendimiento de todos los canales de datos es similar. El delay no aumenta con el tiempo, por lo que no se detectan cuellos de botella. El delay promedio es de 233 milisegundos, similar que en experimentos anteriores. Todo ello nos hace concluir que el sistema es estable, que no se detecta cuellos de botella y que el rendimiento es óptimo dentro de los limites del hardware donde esté funcionando.

Condiciones: ocho canales de datos

Se repite el experimento anterior pero doblando el número de canales de datos, es decir, ocho canales de datos simultáneos. El objetivo de este experimento es comprobar la escalabilidad del sistema, así como detectar anomalías o cuellos de botella.

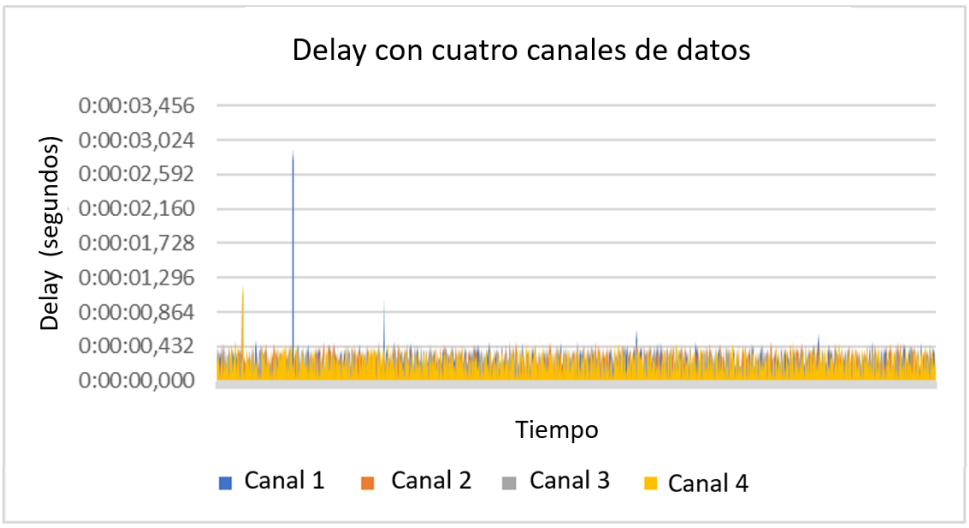

Figura 5.3: Delay con cuatro canales de datos.

Resultados

Como se puede observar en la Figura 5.4, el rendimiento de todos los canales de datos es similar. Obtenemos unos resultados equivalentes que en experimentos anteriores. No se detectan anomalías. El delay promedio es de 240 milisegundos.

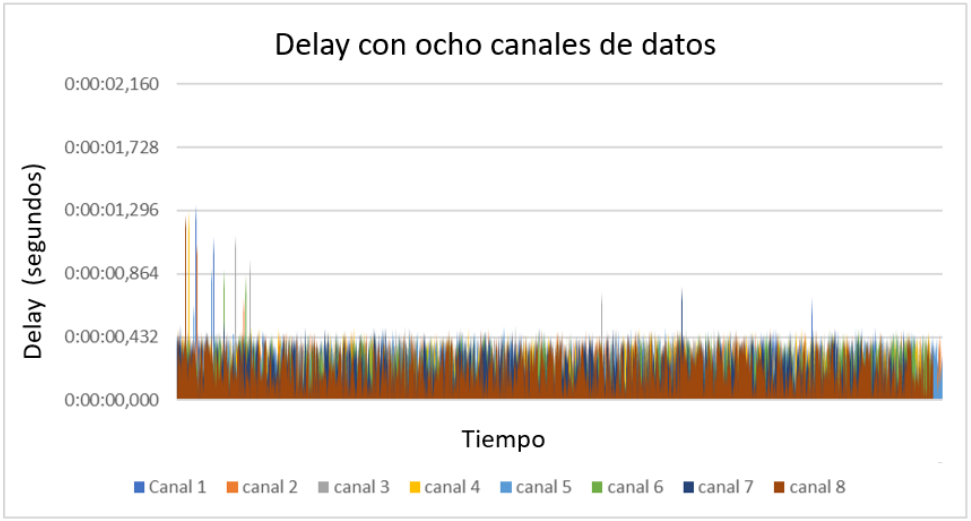

Figura 5.4: Delay con ocho canales de datos.

5.2. Caso de estudio

Con el objetivo de testear toda la funcionalidad desarrollada y mostrar el potencial de Datalyzer, se ha diseñado un caso de estudio a partir de un catálogo de datos abiertos, también conocido como *open data*. Los datos abiertos son datos que pueden

ser usados de forma libre, ya sea en su distribución, reutilización o cualquier otro uso sin restricción de derechos de autor. Típicamente, los datos abiertos son ofrecidos por instituciones, ayuntamientos y otros organismos públicos con el objetivo de ofrecer la máxima transparencia al ciudadano.

El objetivo del caso de estudio es generar una aplicación para la monitorización de algunas variables de interés general sobre la ciudad, proporcionar información útil al ciudadano y crear alertas en base a los datos recibidos. En concreto, los datos de contaminación de la ciudad y el estado del tráfico. Como Madrid es una ciudad grande con muchas estaciones meteorológicas e información sobre el tráfico (calles), se simplifica el proceso ajustando el caso de estudio a los siguientes parámetros:

- **Tráfico.** Se monitoriza el tráfico en la autopista M30, tanto el túnel como la superficie, porque es la vía principal de circulación de Madrid y consideramos que es representativa del estado de la ciudad.
- **Estaciones meteorológicas.** Se monitoriza la estación meteorológica de la calle Arturo soria, en concreto la estación que está ubicada a la entrada de la M30, así como también la estación de Méndez Álvaro.

Para lograr dicho objetivo, se utiliza el catálogo de datos abiertos de la Comunidad de Madrid[1]. El portal ofrece diversos datasets con información sobre la ciudad como accidentes de tráficos, agendas, aviso y más. Algunos datasets son estáticos y otros dinámicos con una tasa de refresco establecida, que puede ser de 1 día, 1 hora o en tiempo real. Para el caso de estudio utilizamos solo datasets que ofrecen información a tiempo real. En concreto, los datasets que usa el caso de estudio son los siguientes:

- **Tráfico M30[2].** Proporciona el número de vehículos que circulan por la M30, tanto en el túnel como la superficie, y la velocidad media en kilómetros/hora de ambos tramos. También informa si hay alguna incidencia.
- **Tráfico calles[3].** Proporciona información sobre el estado del tráfico de un conjunto de calles de Madrid como, por ejemplo, la intensidad (número de coches/hora).
- **Calidad de aire[4].** Proporciona los datos que recopilan las estaciones meteorológicas instaladas por toda la ciudad que miden distintos parámetros del aire.

Como añadido, también se va a usar la red social Twitter para recibir mensajes que los usuarios publican en Madrid sobre la contaminación o el tráfico.

La aplicación que hemos diseñado tiene un total de 7 canales de datos, cada uno con un propósito distinto. La Tabla 5.3 muestra la funcionalidad de cada canal de datos, la fuente a la que está conectada y la salida que proporciona. Se muestra una captura parcial

[1] https://datos.madrid.es/portal/site/egob/
[2] https://datos.madrid.es/egob/catalogo/212117-7899005-trafico-calle30-general.xml
[3] https://datos.madrid.es/egob/catalogo/202087-0-trafico-intensidad.xml
[4] https://datos.madrid.es/egob/catalogo/212117-7899005-trafico-calle30-general.xml

(por espacio) de la aplicación en el DSL en la Figura 5.5. Finalmente, como resultado de la ejecución de la aplicación se genera un panel de control que se muestra parcialmente en el componente 2 de la Figura 5.5.

Canal	**Fuente de datos**	**Descripción**	**Procesador personalizado**	**Salida (Panel de control)**
1	Tráfico M30	Monitorización de la velocidad media en el túnel y la superficie	No	Gráfico
2	Tráfico calles	Monitorización de la intensidad en la calle Arturo Soria	No	Gráfico
3	Calidad del aire	Monitorización del dióxido de nitrógeno	No	Gráfico
4	Tráfico M30	Detección de incidencias (alerta)	Sí	Visualizador
5	Calidad del aire	Detección niveles máximos permitidos de contaminación (alerta)	Sí	Visualizador
6	Tráfico M30	Detección de atascos a partir de la la velocidad media (alerta)	Sí	Visualizador
7	Twitter	Captura de mensajes relacionados	No	Tabla

Tabla 5.3: Canales de datos en la aplicación diseñada para el caso de estudio.

5.2.1. Conclusiones

En este capítulo se ha expuesto la evaluación que se ha realizado sobre Datalyzer. En primer lugar hemos realizado numerosos experimentos para testear el rendimiento y la escalabilidad, así como detectar fallos de diseño o la tasa de error en la transmisión de mensajes. En conjunto, en base a los resultados podemos concluir que Datalyzer está preparado para usarse en un entorno de producción, no se detectan fallos graves de diseño, ni problemas de rendimiento o cuellos de botella. Si bien la tasa de error es mejorable puesto que no debería haber pérdida de mensajes. Por otro lado, el caso de estudio usa todas las características de Datalyzer a fin de mostrar el potencial que tiene en escenarios reales, como es la monitorización de un sistema *smart-city*.

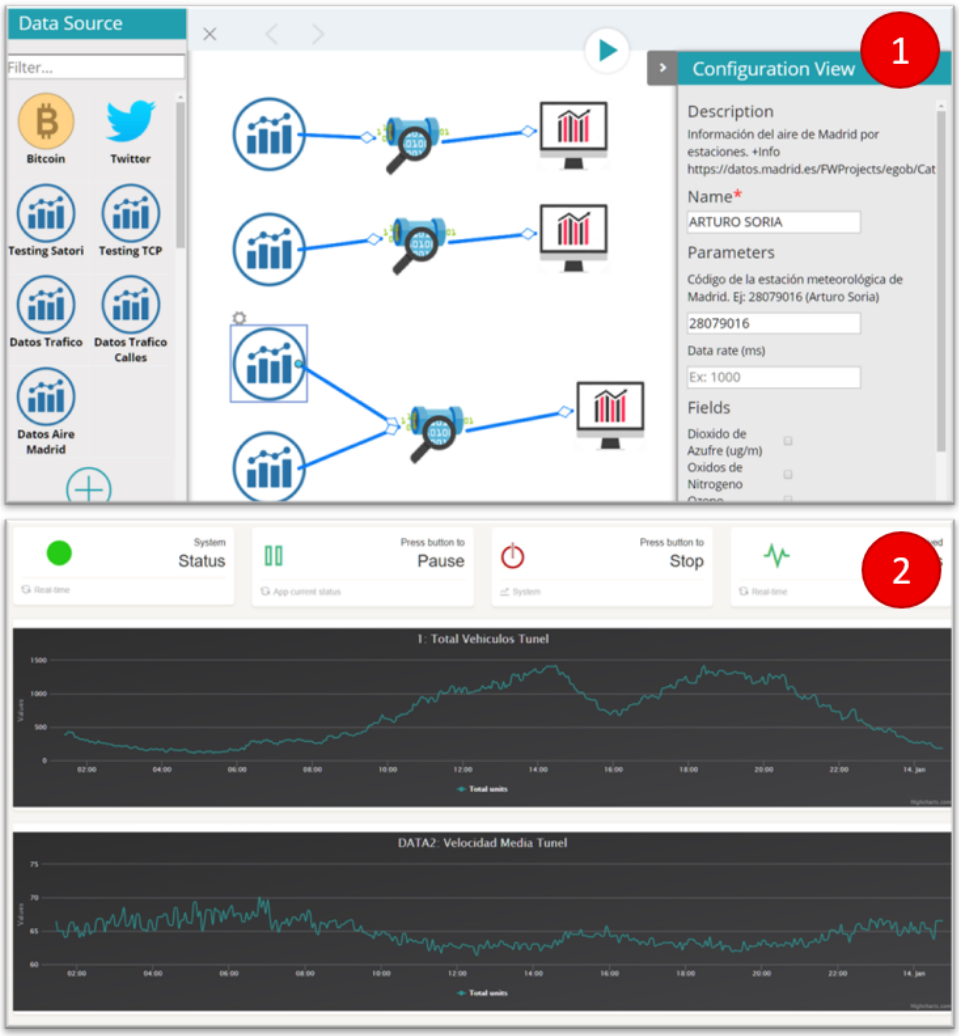

Figura 5.5: Capturas del DSL y panel de control del caso de estudio.

6

Conclusiones y Trabajo Futuro

En este Trabajo de Fin de Máster se ha presentado una metodología para el desarrollo de aplicaciones de datos en streaming que está soportada por Datalyzer, una herramienta open-source que ofrece un entorno de trabajo para crear, almacenar y ejecutar aplicaciones. Además, como se ha explicado en el apartado 4.1.6, Datalyzer se puede desplegar de forma sencilla usando Docker en una infraestructura cloud y también de forma local si un usuario desea trabajar con la herramienta a nivel personal.

Las aplicaciones que se pueden desarrollar en Datalyzer tienen una arquitectura basada en un proceso ETL. En primer lugar, las aplicaciones pueden ingestar datos de cualquier fuente de datos dinámica independiente del protocolo o de su método de acceso, siempre y cuando su modelo se encuentre en la base de datos y su *handler* en el generador de código. Actualmente, Datalyzer tiene una librería con algunas fuentes de datos pre-configuradas como Twitter, OpenWheatherMap o el catálogo de datos abiertos de la Comunidad de Madrid, entre otros. Además, soporta múltiples tecnologías como API Rest, Socket TCP o XML. Los propios usuarios también pueden añadir nuevas fuentes de datos a la librería creando su modelo correspondiente (apartado 3.3).

Los flujos de información que genera la ingesta de datos se transforman en canales de datos sobre los que se puede añadir procesadores. Datalyzer también incorpora una librería de procesadores con algunos pre-instalados como filtros o umbrales. Los usuarios también pueden añadir nuevos procesadores desarrollando una clase en código Java. De esta forma, se garantiza extensibilidad para crear procesadores según los requisitos de cada aplicación, además de aportar algunas características propias de un entorno de trabajo colaborativo puesto que los procesadores los puede usar cualquier usuario.

El proceso de diseño y creación de las aplicaciones es sencillo e intuitivo, gracias al desarrollo de un lenguaje de dominio específico adecuado para esta tarea e integrado en

un editor gráfico (apartado 3.3.2).

Finalmente, cada canal de datos produce un streaming de datos de salida que se puede redireccionar a múltiples destinos según esté configurado. Por ejemplo, podemos hacer una persistencia de los datos en forma de archivos log o podemos redireccionarlos al panel de control para su visualización.

El panel de control es otra de las principales características de Datalyzer. Como las aplicaciones se ejecutan de forma interna, se ofrece un panel para que los usuarios puedan monitorizar y gestionar sus aplicaciones de forma remota y centralizada. Además, el panel de control también puede recibir los datos que procesan las aplicaciones, lo que permite la posibilidad de hacer un análisis de los datos directamente en Datalyzer, así como crear alertas y otras funciones.

En el capítulo 5 se ha evaluado la herramienta a través de experimentos y un caso de estudio. No se ha detectado un fallo de diseño que produzca un cuello de botella en el sistema, si bien tenemos que tener en cuenta los límites hardware del ordenador en el que se esté ejecutando. De los resultados que observamos de los experimentos, podemos concluir que Datalyzer estaría preparado para la fase de puesta en producción.

Como hemos visto anteriormente en el caso de estudio, con las características que actualmente tiene Datalyzer disponible, se presenta un entorno de trabajo para desarrollar aplicaciones de datos en streaming de forma totalmente funcional y que sirve para desarrollar aplicaciones útiles en el mundo real.

Si bien se presenta una herramienta funcional y que se ha evaluado de forma exitosa, existen diversas líneas de investigación como trabajo futuro:

- Se ofrecen algunas características de trabajo colaborativo, como la posibilidad de crear nuevos modelos de fuentes de datos y procesadores que se almacenan en una librería pública. Sin embargo, no podemos afirmar que Datalyzer sea una herramienta colaborativa actualmente. Para ello, los proyectos se deberían poder compartir entre usuarios, ya sea de forma pública -para todos- o privada -con invitación-.

- Algunos usuarios pueden no estar interesados en compartir sus modelos ya sean fuentes de datos o procesadores de forma pública con el resto de usuario. Se propone crear un repositorio alternativo que asocie cada modelo a su autor y que no se comparta con el resto, o solo se comparta bajo invitación.

- El panel de control tiene potencial para convertirse en una avanzada herramienta de data analytics. Para conseguir este objetivo, se debería implementar nuevos y diferentes widgets de visualización y transformación de datos en tiempo real. Por ejemplo, un widget sobre el que se puedan hacer consultas SQL sobre los datos almacenados ya sea en memoria o en ficheros logs.

- Datalyzer se encapsula en un único contenedor Docker para su despliegue y puesta en funcionamiento. Sin embargo, se podría encapsular en una red de contenedores

usando Kubernetes[1] como orquestador, lo que ofrecería la posibilidad de hacer un despliegue de manera distribuida en un clúster o red de ordenadores. Este nuevo diseño de despliegue facilitaría la escalabilidad del sistema y el rendimiento.

- Diseño de nuevos experimentos de prueba de carga con el objetivo de testear el máximo número de usuarios o proyectos de forma simultánea que soporta el sistema sin sobrecarga, teniendo en cuenta los límites hardware sobre el que esté desplegado.
- Implementar en la ventana de home (Figura 4.5) de la plataforma web un pequeño panel de control donde se le muestre al usuario de forma gráfica el número de proyectos que tiene asociado, los proyectos que están actualmente ejecutándose, avisos de error y cualquier otra información de utilidad.
- Diseño de un nuevo caso de estudio basado en el ámbito de los dispositivos IoT.

[1] https://kubernetes.io/es/

Bibliografía

[1] Zhenning Xu, Gary L. Frankwick y Edward Ramirez. «Effects of big data analytics and traditional marketing analytics on new product success: A knowledge fusion perspective». En: *Journal of Business Research* 69.5 (2016). Designing implementable innovative realities, págs. 1562 -1566. ISSN: 0148-2963. DOI: https://doi.org/10.1016/j.jbusres.2015.10.017. URL: http://www.sciencedirect.com/science/article/pii/S0148296315004403.

[2] Yaoxue Zhang y col. «A Survey on Emerging Computing Paradigms for Big Data». En: *Chinese Journal of Electronics* 26.1 (2017), págs. 1-12. DOI: 10.1049/cje.2016.11.016.

[3] Laura Rettig y col. «Online anomaly detection over Big Data streams». En: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015. DOI: 10.1109/bigdata.2015.7363865.

[4] Victor C. Liang y col. «Mercury: Metro density prediction with recurrent neural network on streaming CDR data». En: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016. DOI: 10.1109/icde.2016.7498348.

[5] Lekha R. Nair, Sujala D. Shetty y Siddhanth D. Shetty. «Applying spark based machine learning model on streaming big data for health status prediction». En: *Computers & Electrical Engineering* 65 (2018), págs. 393 -399. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2017.03.009. URL: http://www.sciencedirect.com/science/article/pii/S0045790617305359.

[6] Constantin Pohl, Philipp Götze y Kai-Uwe Sattler. «A Cost Model for Data Stream Processing on Modern Hardware». En: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1, 2017*. 2017. URL: http://www.adms-conf.org/2017/camera-ready/adms2017_final.pdf.

[7] Gabriel Iuhasz y Ioan Dragan. «An Overview of Monitoring Tools for Big Data and Cloud Applications». En: *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2015. DOI: 10.1109/synasc.2015.62.

[8] Michael Stonebraker, U?ur Çetintemel y Stan Zdonik. «The 8 requirements of real-time stream processing». En: *ACM SIGMOD Record* 34.4 (2005), págs. 42-47. DOI: 10.1145/1107499.1107504.

[9] Marcos Dias de Assunção, Alexandre da Silva Veith y Rajkumar Buyya. «Distributed data stream processing and edge computing: A survey on resource elasticity and future directions». En: *Journal of Network and Computer Applications* 103 (2018), págs. 1-17. DOI: 10.1016/j.jnca.2017.12.001.

[10] Hal R. Varian. «Big Data: New Tricks for Econometrics». En: *Journal of Economic Perspectives* 28.2 (2014), págs. 3-28. DOI: 10.1257/jep.28.2.3.

[11] Siyang Qin y col. «Applying Big Data Analytics to Monitor Tourist Flow for the Scenic Area Operation Management». En: *Discrete Dynamics in Nature and Society* 2019 (2019), págs. 1-11. DOI: 10.1155/2019/8239047.

[12] François Schnitzler y col. «Heterogeneous Stream Processing and Crowdsourcing for Traffic Monitoring: Highlights». En: *Machine Learning and Knowledge Discovery in Databases*. Springer Berlin Heidelberg, 2014, págs. 520-523. DOI: 10.1007/978-3-662-44845-8_49.

[13] Thamer Al-Rousan. «Cloud Computing for Global Software Development». En: *International Journal of Cloud Applications and Computing* 5.1 (2015), págs. 58-68. DOI: 10.4018/ijcac.2015010105.

[14] Vicente Pelechano Antonio Valecillos Juan Manuel Vara Cristina Vicente-Chicote Jesús García Molina Félix O. García Rubio. *Desarrollo del software dirigido por modelos: Conceptos, métodos y herramientas*. Ra-Ma Editorial, 2013. ISBN: 978-84-9964-215-4.

[15] Steven Kelly y Juha-Pekka Tolvanen. *Domain-Specific Modeling*. John Wiley & Sons, Inc., 2008. DOI: 10.1002/9780470249260.

[16] Object Management Group (OMG). *Meta-Object Facility (MOF) Specification, Version 2.5.1*. OMG Document Number formal/16-11-01 (https://www.omg.org/spec/MOF/2.5.1/). 2016.

[17] David Steinberg y col. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0321331885.

[18] Moritz Eysholdt y Heiko Behrens. «Xtext: implement your language faster than the quick and dirty way». En: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH 10*. ACM Press, 2010. DOI: 10.1145/1869542.1869625.

[19] Florian Heidenreich y col. «Model-Based Language Engineering with EMFText». En: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, págs. 322-345. DOI: 10.1007/978-3-642-35992-7_9.

[20] Vladimir Viyovic, Mirjam Maksimovic y Branko Perisic. «Sirius: A rapid development of DSM graphical editor». En: *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, 2014. DOI: 10.1109/ines.2014.6909375.

[21] Markus Gerhart y M. Boger. «Concepts for the model-driven generation of graphical editors in eclipse by using the graphiti framework». En: *International Journal of Computer Techniques* (2016).

[22] S. Sendall y W. Kozaczynski. «Model transformation: the heart and soul of model-driven software development». En: *IEEE Software* 20.5 (2003), págs. 42-45. DOI: 10.1109/ms.2003.1231150.

[23] Jonathan Musset y col. «Acceleo user guide». En: *See also http://acceleo.org/doc/obeo/en/acceleo-2.6-user-guide. pdf* 2 (2006), pág. 157.

[24] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[25] Dibyendu Bhattacharya y Manidipa Mitra. «ANALYTICS ON BIG FAST DATA USING REAL TIME STREAM DATA PROCESSING ARCHITECTURE». En: 2013.

[26] Nishant Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[27] Marco Brambilla, Jordi Cabot y Manuel Wimmer. «Model-Driven Software Engineering in Practice: Second Edition». En: *Synthesis Lectures on Software Engineering* 3.1 (2017), págs. 1-207. DOI: 10.2200/s00751ed2v01y201701swe004.

[28] Andreas Harth y col. «On-the-fly Integration of Static and Dynamic Sources». En: *Proceedings of the Fourth International Workshop on Consuming Linked Data (COLD2013)*. 2013.

[29] Nihal Dindar y col. «Modeling the execution semantics of stream processing engines with SECRET». En: *The VLDB Journal* 22.4 (2012), págs. 421-446. DOI: 10.1007/s00778-012-0297-3.

[30] Narayan Prusty. *Learning ECMAScript 6*. Packt Publishing Ltd, 2015.

[31] Charles Anderson. «Docker [Software engineering]». En: *IEEE Software* 32.3 (2015), págs. 102-c3. DOI: 10.1109/ms.2015.62.