

8 | Patrón ETL

Una arquitectura de flujo de datos (*data-flow architecture*) consiste en una estructura en donde una serie de componentes “mueven” datos provenientes de una o varias fuentes a un destino; generalmente sometiéndolos a un proceso de transformación. Existen varios patrones que siguen una arquitectura de flujo de datos. En este capítulo se describe uno muy popular actualmente: el *Patrón ETL*.

8.1. Descripción

El poder disponer de datos, provenientes de fuentes públicas o privadas, respresenta una ventaja para muchas organizaciones. Sin embargo, el tenerlos en condiciones adecuadas para un uso específico, en una aplicación en particular, puede requerir varios esfuerzos. Por ejemplo, es posible que sea necesario integrar múltiples fuentes de datos. Igualmente, es posible que sea necesario homogeneizar datos que están en formatos distintos. El *Patrón ETL* es un patrón arquitectónico que puede ser utilizado para resolver muchas de estas necesidades.

ETL es un acrónimo que significa extraer, transformar, cargar (*extract, transform and load* en inglés). Estas tareas se refieren a las consideradas en un proceso mediante el cual los datos se extraen desde una o varias fuentes, se transforman considerando necesidades específicas del negocio, y luego se cargan a su destino final para su uso en el negocio para una aplicación en particular. En el *Patrón ETL* las tareas de extracción, transformación y carga de datos son generalmente realizadas por distintos componentes organizados como una arquitectura de flujo de datos. Esto es, los componentes están organizados en una secuencia en donde cada componente ejecuta su tarea desde su inicio hasta su finalización; luego se ejecuta la siguiente tarea por el siguiente componente en la secuencia, y así sucesivamente, hasta que se completan todas las tareas. Durante cada tarea, un lote de datos es procesado y al terminar, los datos procesados son la entrada del siguiente componente y así hasta terminar las tareas.

Implementaciones básicas del *Patrón ETL* incluyen un componente que funciona como fuente de datos, un componente que realiza el procesamiento de datos y un componente que realiza la carga de datos en su destino final. La siguiente tabla describe con más detalle éstos elementos así como otros aspectos de interés sobre este patrón.

Elementos	<p><i>Fuente de datos:</i> componente que contiene el, o permite el acceso al, lote de datos a procesar.</p> <p><i>Extractor de datos:</i> componente que obtiene los datos a procesar desde el componente Fuente de datos.</p> <p><i>Transformador de datos:</i> componente que obtiene los datos del Extractor de datos, realiza algún proceso de transformación en estos y los pasa al componente Cargador de Datos.</p> <p><i>Cargador de datos:</i> componente que toma los datos del componente Transformador de datos y los carga al componente Destino de datos.</p> <p><i>Destino de datos:</i> componente que contiene el, o permite el acceso al, lote de datos procesados.</p>
Relaciones	<p><i>Lee:</i> permite a un componente procesador de datos leer un lote de datos desde la fuente de datos.</p> <p><i>Flujo de datos:</i> regula la entrada y salida de datos entre componentes procesadores de datos.</p> <p><i>Escribe:</i> permite a un procesador de datos escribir un lote de datos en el destino de datos.</p>
Modelo de computación	Un lote de datos entra al sistema desde una fuente y se mueve a través de los procesadores de datos hasta que llega a un destino. La disponibilidad de los datos controla el procesamiento.
Restricciones	La ejecución de un procesador de datos empieza una vez que el procesador inmediato anterior en la secuencia termina su ejecución.
Principios de diseño y atributos de calidad	<p><i>Bajo acoplamiento:</i> hay bajo acoplamiento entre los procesadores de datos. Un procesador de datos desconoce de cuál procesador de datos provienen los datos que recibe y a qué procesador de datos serán enviados una vez que los procese.</p> <p><i>Cohesión:</i> cada procesador de datos realiza un procesamiento específico y relacionado en el desarrollo de una única función.</p> <p><i>Facilidad de mantenimiento:</i> es fácil realizar cambios en los procesadores de datos ya que están desacoplados. Igualmente, los cambios en los procesador de datos son fáciles porque están localizados ya que su implementación es cohesiva.</p>

Cuadro 8.1: Descripción de elementos y otros aspectos de interés del *Patrón ETL*.

Considerando la información anteriormente descrita, la Figura 8.1 muestra una vista estática de los elementos del *Patrón ETL*; la Figura 8.2 muestra una vista dinámica.

8.2. Ejemplo práctico

En esta sección se presenta una instancia de uso del *Patrón ETL* utilizando un ejemplo práctico. Para ello, inicialmente se expone una descripción general del sistema en este ejemplo. Posteriormente, se describen los requerimientos de arquitectura. Considerando estos requerimientos, después se presenta un diseño arquitectónico del sistema utilizando

Figura 8.1: Vista estática del *Patrón ETL*.

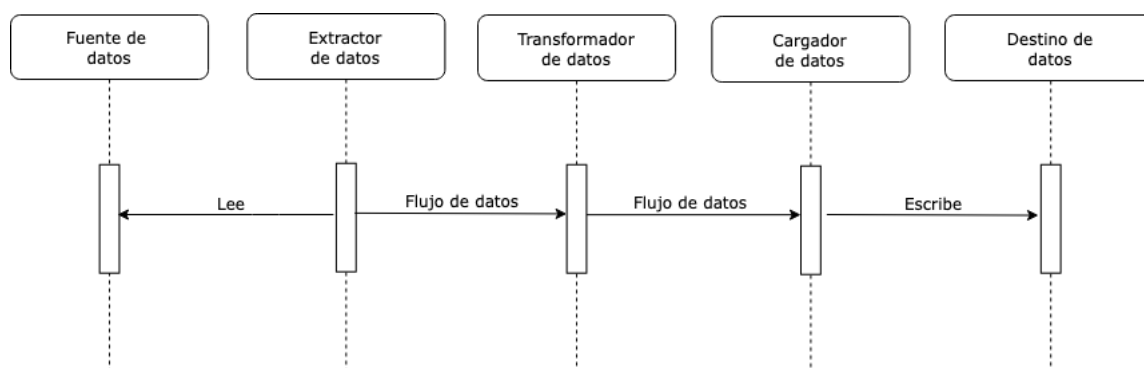


Figura 8.2: Vista dinámica del *Patrón ETL*.

el *Patrón ETL*. Finalmente, se describe una posible implementación del diseño utilizando Luigi [The20].

8.2.1. Descripción General

Para el ejemplo práctico vamos a suponer la existencia de una empresa británica llamada ASDA. Esta empresa se encarga de comercializar distintos productos en varios países de Europa. Este año la empresa ha decidido revisar su estrategia de ventas de forma semanal con el propósito de mejorar la toma de decisiones relacionadas su expansión. Para esto el Chief Sales Officer (CSO) de ASDA solicitó a la Dirección de Comercialización reunir la información de ventas de la compañía en cada una de las sucursales para su procesamiento y análisis. Muchas de estas actividades fueron realizadas en el Sistema Janus.

8.2.2. Requerimientos de Arquitectura

El objetivo de negocio del Sistema Janus es:

Reducir un 50 % el tiempo del proceso de generación de información para apoyar la toma de decisiones relacionadas a la estrategia de expansión de ASDA.

Considerando este objetivo de negocio, el Sistema Janus implementa los siguientes requerimientos de arquitectura:

(a) Requerimientos funcionales:

- Como gerente de ventas necesito visualizar un reporte de ventas para presentar el reporte financiero a los accionistas de la empresa.
- Como gerente de almacén necesito visualizar un reporte de los productos más vendidos para mantener un stock suficiente.
- Como operador de logística necesito visualizar un reporte de la cantidad de productos vendidos para ayudar a mantener un periodo de entrega de productos a los clientes no mayor a 40 días.
- Como CSO de la empresa necesito visualizar un reporte de los indicadores de ventas de todas las sucursales.

(b) Requerimientos de atributos de calidad:

- Flexibilidad: el sistema deberá procesar datos de ventas en distintos formatos y permitir la inclusión de nuevos.
- Escalabilidad: el sistema deberá soportar el procesamiento de datos de ventas mensuales de hasta 500 sucursales.
- Confiabilidad: el sistema deberá tener una precisión en el procesamiento de información de al menos 99.99 %.
- Desempeño: el sistema deberá procesar los datos de ventas mensuales de hasta 500 sucursales en no mas de 60 segundos.

(c) Restricciones técnicas:

- Para el desarrollo del sistema se deberán utilizar Python como lenguaje de programación.
- Para el desarrollo del sistema se deberán utilizar herramientas de código abierto o gratuitas.

8.2.3. Diseño del sistema

En está sección se presenta el diseño del Sistema Janus a través de un conjunto de vistas, usando la notación C4 propuesta por Simon Brown [Bro15]. La vista estática en la figura 8.3 presenta un diagrama de contexto del Sistema Janus. Como puede observarse, y considerando los requerimientos de usuario, el sistema Janus es usado por los gerentes de ventas y almacén, así como por los operadores de logística y el CSO.

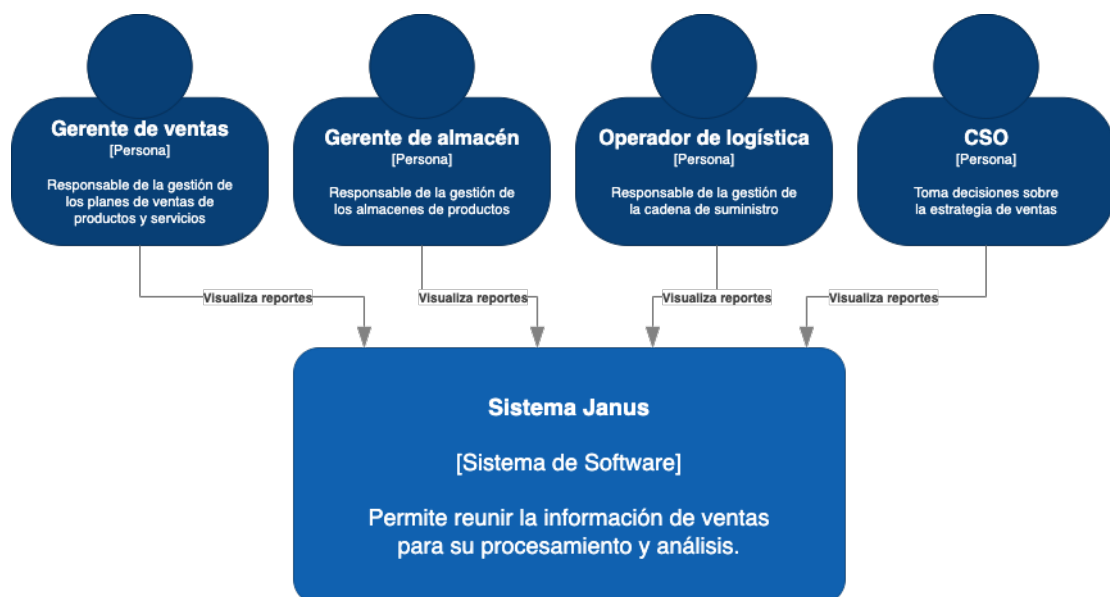


Figura 8.3: Diagrama de contexto del Sistema Janus.

La vista estática en la figura 8.4 muestra la arquitectura principal del Sistema Janus. Existe un componente Cliente que funciona como una interfaz de usuario. Existe un componente *Gestor de datos*, que como explicaremos implementa un *Patrón ETL* para realizar

el proceso de extracción, transformación, carga y visualización de información sobre las ventas en las sucursales de ASDA.

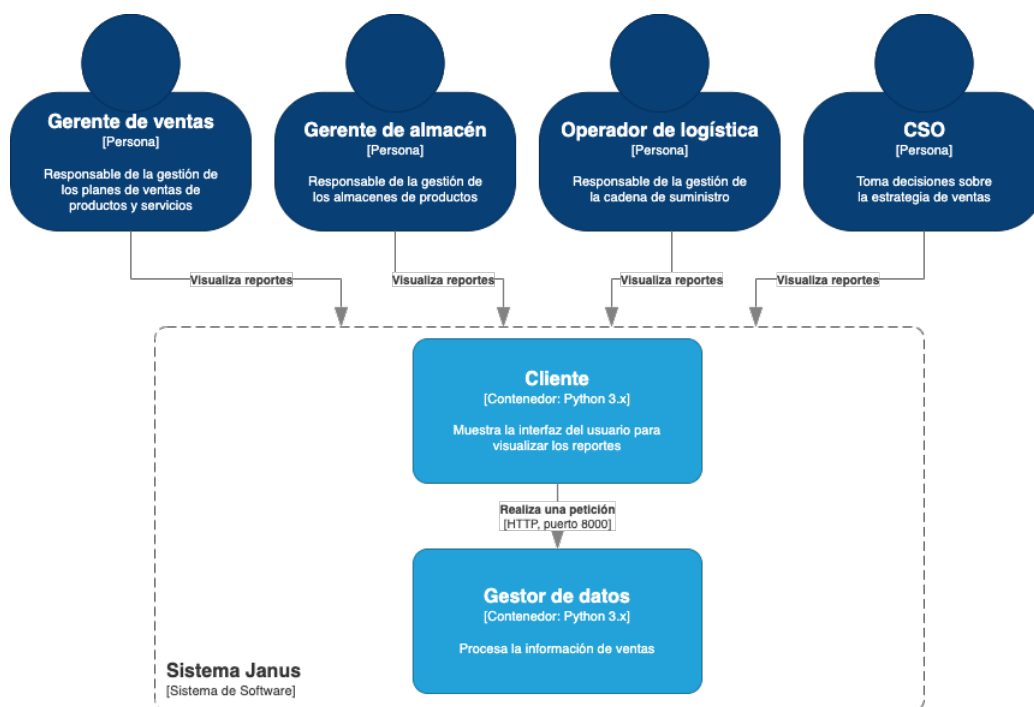


Figura 8.4: Vista de módulos del Sistema Janus.

La figura 8.5 es otra vista estática del sistema que muestra cómo está conformado el Gestor de datos mostrado en la figura anterior. Existe un componente llamado Lector ZIP, el cual se encarga de leer y extraer el contenido de los archivos con la información de ventas que se encuentran en formato *zip*. El Gestor de Datos incluye el *Patrón ETL*; los componentes de este patrón están encerrados en un recuadro rojo. Hay un conjunto de tres componentes Extractores, los cuales se encargan de leer la información extraída del archivo *zip*, y que se encuentra en formatos distintos (*.xml*, *.html* y *.csv*). Hay un conjunto de tres componentes Transformadores los cuales transforman los datos, provistos por los componentes Extractores, en un formato homogéneo para que pueda ser consolidada y después almacenada en una Base de Datos por el componente Cargador.

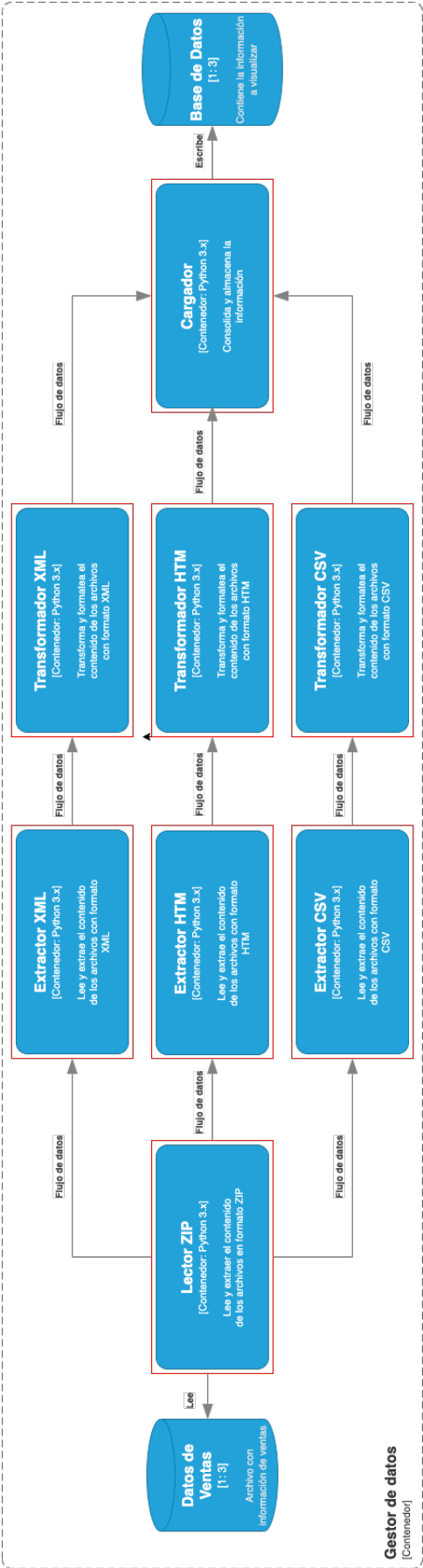


Figura 8.5: Elementos del componente *Gestor de Datos*.

La figura 8.6 muestra la vista dinámica del componente *Gestor de Datos*. En esta vista es posible observar cómo los elementos de este componente interactúan entre sí.

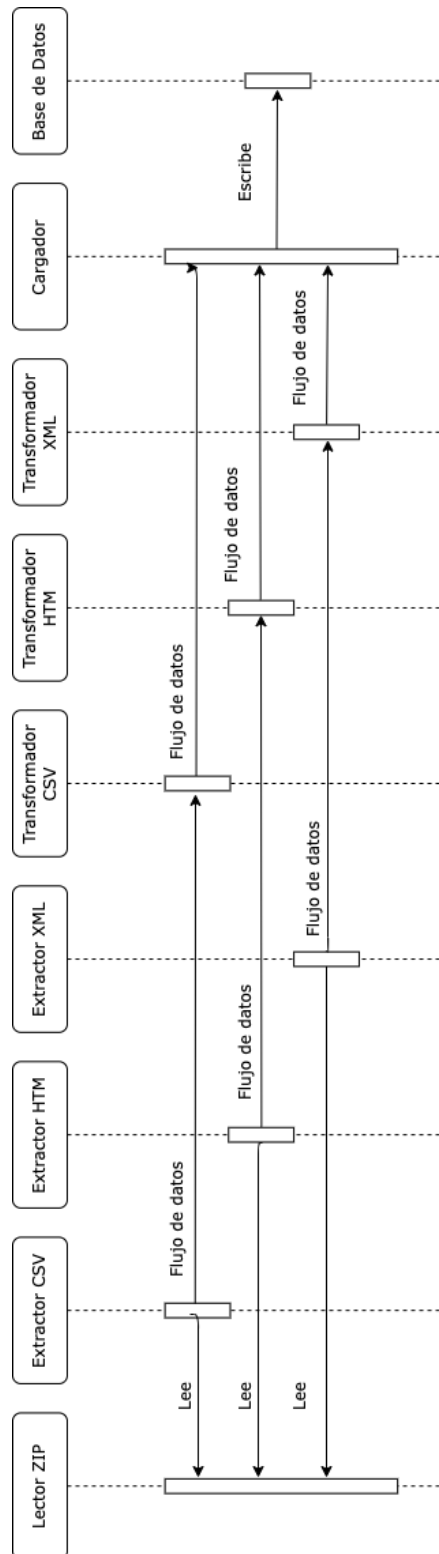


Figura 8.6: Elementos del componente *Gestor de Datos*.

Como puede observarse, en el flujo del componente *Gestor de Datos* mostrado en la figura 8.6 solo existe un orden de ejecución, es decir, cuando un elemento termina su ejecución no volverá a invocarse. También es necesario mencionar que la ejecución de un elemento no se realiza sino hasta que todos los elementos del cuál depende no se han terminado de ejecutar. En este sentido, tomando como referencia la figura 8.6, el elemento Cargador no se ejecutará sino hasta que terminen de ejecutarse todos los elementos Transformador, éstos a su vez, no se ejecutarán hasta que terminen de ejecutarse todos los elementos Extractor, y éstos, no se ejecutarán hasta que se haya ejecutado el elemento Lector ZIP.

8.3. Implementación con Luigi

Luigi [The20] es un paquete de Python que lo ayuda a implementar arquitecturas de flujo de datos mediante la conexión de tareas. Actualmente el framework Luigi es utilizado por Spotify, Foursquare, Mortar Data, Stripe, Red Hat, GetNinjas, Asana, Buffer, entre otros.

Comenzando con un archivo .ZIP como fuente de datos y terminando con un formato .JSON como formato estandarizado que después es enviado a la base de datos. Es importante recordar que dentro del proceso ETL, la salida de un procesador es la entrada del siguiente procesador, en ese sentido, la salida del Lector ZIP es la entrada de los Extractores, su salida es la entrada de los Transformadores y su salida es la entrada del Cargador, siendo éste el último procesador y responsable de enviar los datos procesados a la base de datos.

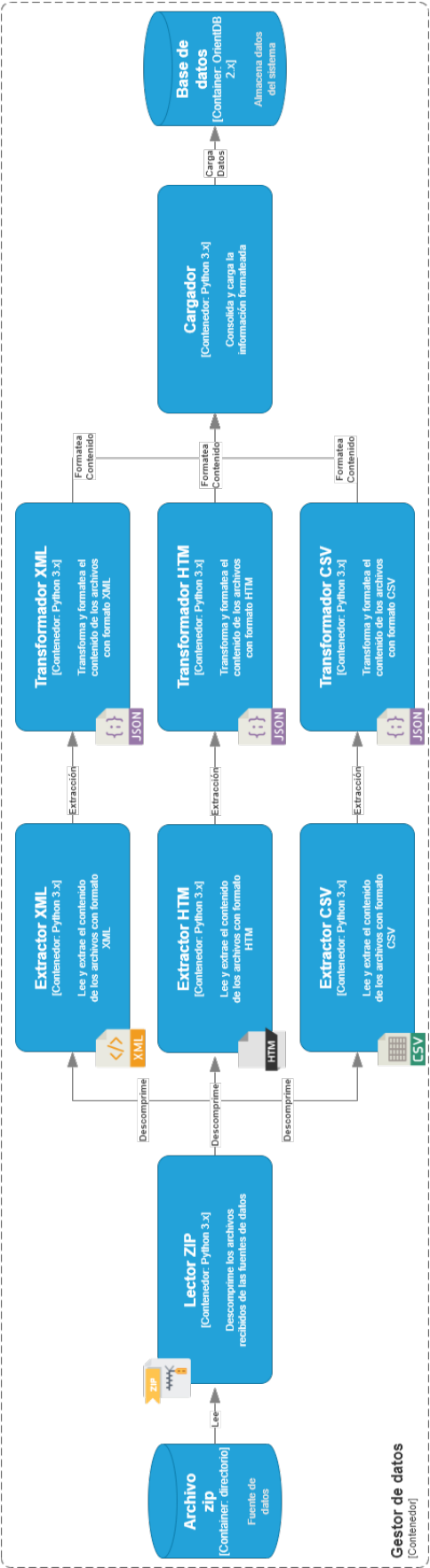


Figura 8.7: Proceso detallado del sistema Janus

8.3.1. Implementación

El código del Sistema Janus se puede descargar desde el siguiente enlace:

<https://gitlab.com/tareas-arquitectura-de-software-curso/flujo-de-datos/>

En este enlace se encontrarán dos repositorios, correspondientes a los componentes Cliente y Gestor de datos respectivamente, tal como se muestra en la figura 8.8.

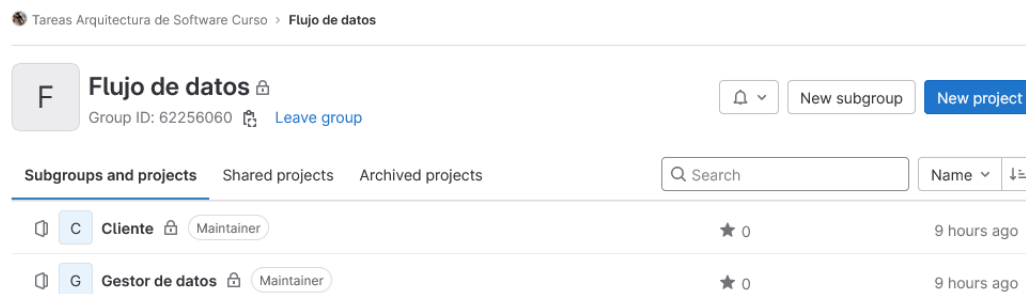


Figura 8.8: Grupo de repositorios del *Patrón ETL*

En esta ocasión nos enfocaremos en el repositorio *Gestor de datos* con la intención de analizar y describir la implementación del proceso ETL. En la figura 8.9 se muestra la estructura del repositorio del componente *Gestor de datos*.

Name	Last commit	Last update
assets	Added reader	2 weeks ago
result	Added project structure	2 weeks ago
src	Added comments	3 hours ago
.gitignore	Initial commit	2 weeks ago
Dockerfile	Added comments	3 hours ago
README.md	Updated component name	4 minutes ago
loader.py	Added comments	3 hours ago
requirements.txt	Added requirements file	2 weeks ago

Figura 8.9: Estructura del repositorio del componente Gestor de datos

Como se puede observar, la estructura principal del repositorio cuenta con 3 directorios y 5 archivos. Comenzaremos describiendo los archivos.

El archivo *.gitignore* contiene la definición de las omisiones que se tomarán en cuenta al contribuir en el código del componente. El archivo *Dockerfile* define las instrucciones que se llevarán a cabo para construir una imagen *Docker* del componente. El archivo *README.md* contiene las instrucciones de uso y otros detalles del repositorio. El archivo *loader.py* corresponde al punto de ejecución del componente. Por último, el archivo *requirements.txt* define las dependencias de *Python* necesarias para la ejecución del componente.

Por otro lado, las carpetas dentro de la estructura del repositorio del componente *Gestor de datos* se utilizan para lo siguiente. Las carpetas *assets* y *result* se utilizan de manera auxiliar por el proceso ETL, de manera inicial la carpeta *result* no contiene más que un archivo *README.md* que describe el nombre de la carpeta, la carpeta *assets* además de contener un archivo *README.md* que también describe el nombre de la carpeta, contiene la fuente de datos en formato .ZIP que es utilizada como entrada del primer procesador de datos.

La carpeta *src* contiene el código de los procesadores de datos utilizados en el proceso ETL, organizados en subcarpetas de acuerdo al tipo de procesador de datos, que en éste caso son *lectores*, *extractores* y *transformadores*.

En la siguiente sección se describirá con mayor detalle el código de uno de los procesadores de datos utilizados en el proceso ETL del sistema Janus.

8.3.2. Descripción del código

En esta sección se describirán extractos de código que implementa el sistema Janus. Debido a que todos los elementos del componente *Gestor de datos* utilizan el framework *Luigi* en el cuadro 8.1 se muestra la estructura de una tarea de *Luigi*.

Código 8.1: Estructura base de una tarea en *Luigi*

```

1 class LuigiTaskExample(luigi.Task):
2
3     def output(self):
4         pass
5
6     def requires(self):
7         pass
8
9     def run(self):
10        pass

```

Crear una tarea de *Luigi* es realmente sencillo, basta con crear una clase y pasarle como argumento la clase `luigi.Task`. Una vez declarada la clase que definirá una tarea de *Luigi* es necesario sobrescribir tres métodos `output()`, `requires()` y `run()`.

En la tabla 8.2 se describen la responsabilidad o el rol que tendrán dentro de la tarea de *Luigi* cada uno de estos métodos.

Método	Descripción
<code>output()</code>	Indica el archivo de salida una vez se haya completado la tarea.
<code>requires()</code>	Contiene una lista de tareas de las cuales depende la tarea actual, es decir, tareas que deben realizarse antes de la tarea actual
<code>run()</code>	Contiene la lógica de la tarea

Cuadro 8.2: Métodos contenidos en la anatomía de una tarea de *Luigi*

Transformador

El cuadro 8.2 muestra un ejemplo de la implementación de una tarea de *Luigi* para transformar datos.

Código 8.2: Ejemplo de implementación de una tarea de Luigi

```

1 class CSVTransformer(luigi.Task):
2
3     def requires(self):
4         return CSVExtractor()
5
6     def run(self):
7         result = []
8         for file in self.input():
9             with file.open() as csv_file:
10                 csv_reader = csv.reader(csv_file)
11                 header = []
12                 regex = re.compile('[^a-zA-Z]')
13                 header = [regex.sub('', column) for column in next(
csv_reader)]
14                 for row in csv_reader:
15                     entry = dict(zip(header, row))
16
17                     if not entry["productdesc"]:
18                         continue
19
20                     result.append(
21                         {
22                             "description": entry["productdesc"],
23                             "quantity": entry["qty"],
24                             "price": entry["rawprice"],
25                             "total": float(entry["qty"]) * float(entry["
rawprice"]),
26                             "invoice": entry["inv"],
27                             "provider": entry["provider"],
28                             "country": entry["countryname"]
29                         }
30                     )
31                 with self.output().open('w') as out:
32                     out.write(json.dumps(result, indent=4))
33
34     def output(self):
35         project_dir = os.path.dirname(os.path.abspath("loader.py"))
36         result_dir = join(project_dir, "result")
37         return luigi.LocalTarget(join(result_dir, "csv.json"))

```

Como se puede observar en el código anterior el nombre de la tarea es `CSVTransformer` y tal como el nombre lo sugiere, esta tarea se encarga de transformar datos contenidos en un archivo `csv`. El método `requires()` indica que esta tarea requiere de una tarea previa para ser ejecutada y esta tarea de la cual depende se llama `CSVExtractor`. Una vez se haya ejecutado la tarea dependiente se procede a ejecutar el método `run`, el cuál contiene la lógica de negocio utilizada para la transformación de los datos. Por último, en el método `output` se define la salida generada después de la ejecución del método `run`, que en este ejemplo será un archivo local llamado `csv.json`.

8.3.3. Ejecución

Una vez clonado el repositorio del sistema, para ejecutarlo, deberemos realizar en 3 pasos: instanciación de la base de datos, ejecución del proceso ETL e instanciación de la interfaz cliente que nos permitirá interactuar con el sistema. A continuación se detallará cada una de los pasos.

Para instanciar la base de datos, se ejecutará el siguiente comando:

```
docker run -d -p 5080:5080 -p 6080:6080 -p 8080:8080 -p 9080:9080 -p
8000:8000 -name dgraph dgraph/standalone
```

Para validar que la base de datos se ha instanciado correctamente, podremos acceder a la url <http://localhost:8000/> desde cualquier navegador, éste mostrará el portal de acceso a la base de datos, tal como se muestra en la figura 8.10.

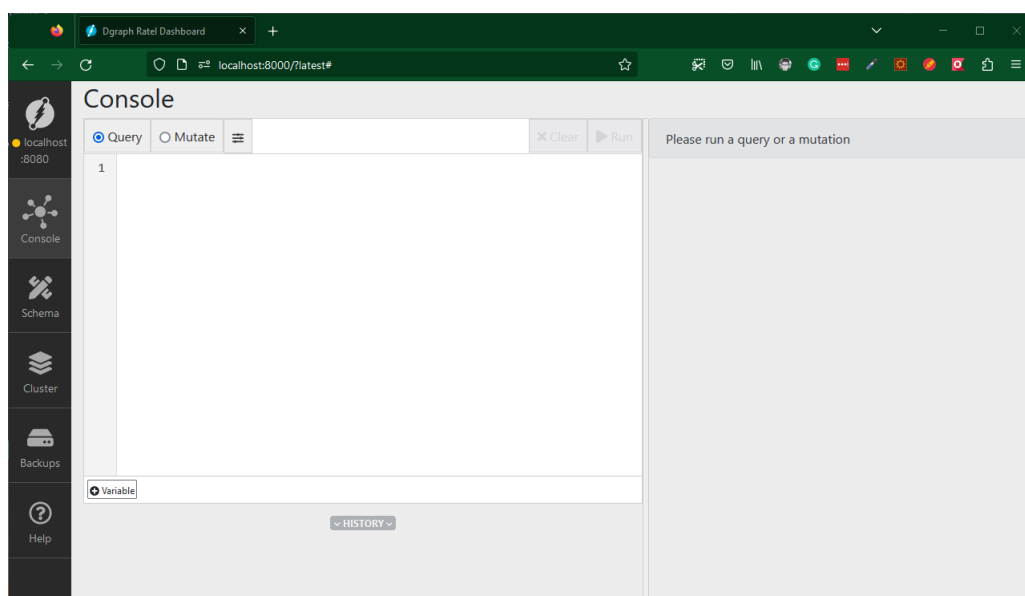


Figura 8.10: Portal de acceso Dgraph

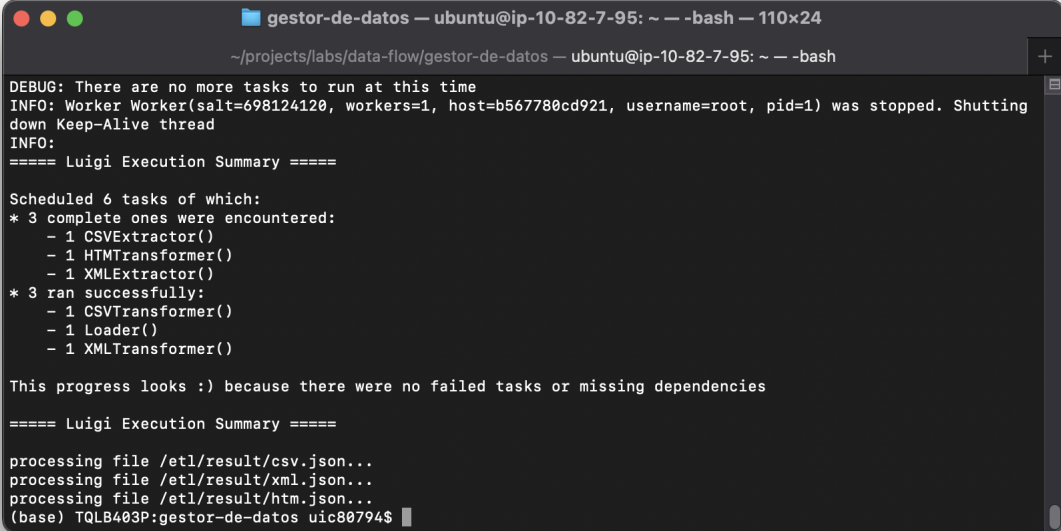
Después de instanciar la base de datos, es momento de ejecutar el proceso ETL, para ello accederemos a la carpeta **gestor-de-datos**, una vez dentro de la carpeta se construirá la imagen de docker que nos permitirá ejecutar el proceso ETL:

```
docker build -t gestor-de-datos .
```

Luego de crear la imagen de docker, se utilizará el siguiente comando para ejecutar el proceso ETL:

```
docker run -rm -name gestor-de-datos -link dgraph:dgraph gestor-de-datos
```

Al terminar la ejecución, se podrá observar el resultado del proceso tal como se muestra en la figura 8.11.



```

gestor-de-datos — ubuntu@ip-10-82-7-95: ~ — -bash — 110x24
~/projects/labs/data-flow/gestor-de-datos — ubuntu@ip-10-82-7-95: ~ — -bash
DEBUG: There are no more tasks to run at this time
INFO: Worker Worker(salt=698124120, workers=1, host=b567780cd921, username=root, pid=1) was stopped. Shutting
down Keep-Alive thread
INFO:
===== Luigi Execution Summary =====

Scheduled 6 tasks of which:
* 3 complete ones were encountered:
  - 1 CSVExtractor()
  - 1 HTMTransformer()
  - 1 XMLExtractor()
* 3 ran successfully:
  - 1 CSVTransformer()
  - 1 Loader()
  - 1 XMLTransformer()

This progress looks :) because there were no failed tasks or missing dependencies

===== Luigi Execution Summary =====

processing file /etl/result/csv.json...
processing file /etl/result/xml.json...
processing file /etl/result/htm.json...
(base) TQLB403P:gestor-de-datos uic80794$

```

Figura 8.11: Resultado ejecución ETL

Finalmente, será momento de instanciar el último componente del sistema, la interfaz cliente que nos permitirá interactuar con el sistema. Para este último componente, será necesario ubicarnos en la carpeta cliente, una vez dentro, construiremos la imagen de docker que nos permitirá instanciar el componente de la interfaz gráfica, utiliza el siguiente comando para construir la imagen:

```
docker build -t cliente .
```

A continuación, ejecuta el siguiente comando para instanciar el componente de la interfaz de usuario:

```
docker run -name cliente -p 0.0.0.0:5000:5000 -link dgraph:dgraph cliente
```

Para acceder al componente de la interfaz de usuario, puede ser a través de la url <http://localhost:5000> desde cualquier navegador, éste mostrará el resumen de las ventas del sistema, tal como se muestra en la figura 8.12.

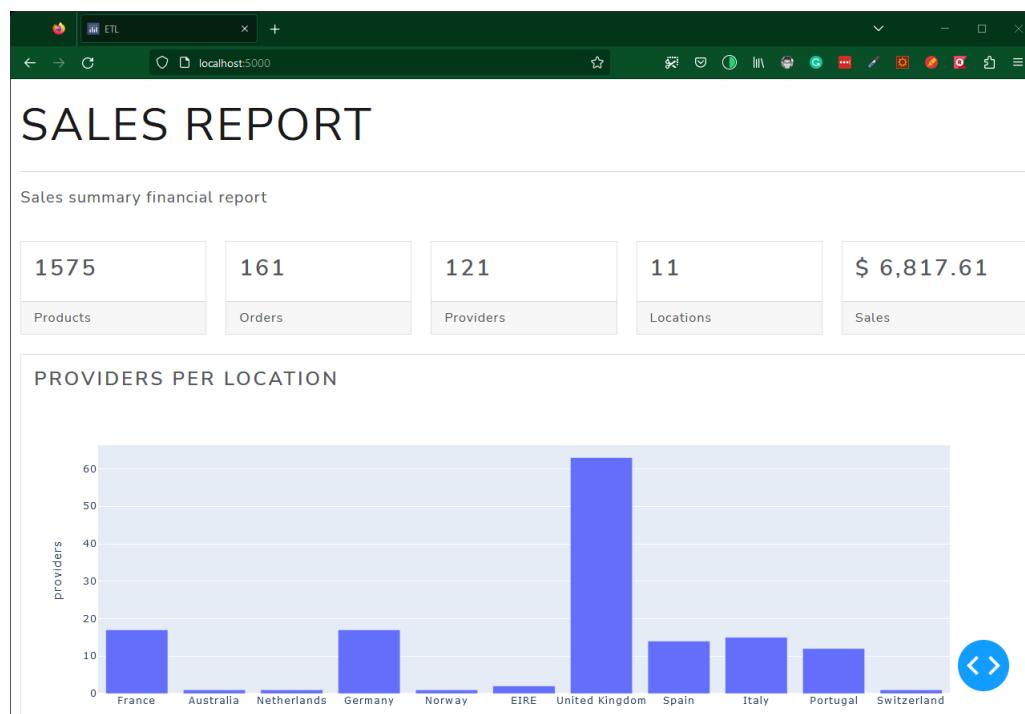


Figura 8.12: Resumen de ventas