Proyecto Integrador Final – Documentación

**Data Engineering - Accenture**

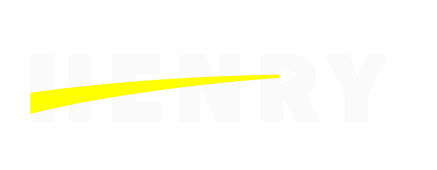


Tabla de contenidos

[Introducción 3](#_Toc200274094)

[Estructura del proyecto 3](#_Toc200274095)

[Estructura de la aplicación 4](#_Toc200274096)

[Solución implementada 4](#_Toc200274097)

[Entendiendo la clase MySQLConnector 5](#_Toc200274098)

[Atributos de instancia 5](#_Toc200274099)

[Métodos 6](#_Toc200274100)

[Justificación 7](#_Toc200274101)

[Entendiendo la clase DataIngestion 7](#_Toc200274102)

[Atributos de instancia 7](#_Toc200274103)

[Métodos 8](#_Toc200274104)

[Justificación 8](#_Toc200274105)

[Factories 9](#_Toc200274106)

[DataLoaderFactory 10](#_Toc200274107)

[DataValidationFactory 11](#_Toc200274108)

[FastApi 12](#_Toc200274109)

[Endpoint principal 12](#_Toc200274110)

[Lógica de Funcionamiento 13](#_Toc200274111)

[Justificación 13](#_Toc200274112)

[Jupyter Notebooks y sentencias de SQL 14](#_Toc200274113)

[CRUDQueryBuilder 14](#_Toc200274114)

[Avance 2 Notebook 15](#_Toc200274115)

[Avance 3 Notebook 15](#_Toc200274116)

# Introducción

El siguiente documento tiene por fin explicar la elaboración del proyecto final integrador del curso de Data Engineering. La idea era elaborar un proyecto que nos permita:

1. Crear las tablas en una base de datos MySQL a partir del modelado de datos
2. Hacer una carga masiva con el comando **LOAD DATA INFILE** de unos archivos .csv con datos de venta de una empresa (productos, categorías, países, ciudades, ventas, etc.)
3. Crear una clase en Python que sea la encargada de gestionar las conexiones a nuestra base de datos creada
4. Utilizar distintas librerías de Python (pandas, sqlalchemy, etc.) para que a partir de la conexión podamos realizar consultas con SQL para acceder a nuestra información.

A partir de este desafio, decidí crear este proyecto de una manera escalable y entendible para cualquier Data Engineering pueda realizar modificaciones o bien entender la lógica del negocio.

## Estructura del proyecto

En el directorio raíz cuenta con 6 directorios y 6 archivos. Detallamos a continuación a que se refiere cada uno de manera genérica:

* + Directorio **data**: Contiene los archivos .csv que se cargan dinámicamente
  + Directorio **src**: Encapsula la lógica de la aplicación
  + Directorio **sql**: Contiene los scripts de SQL utilizados
  + Directorio .**venv**: Contiene todas nuestras dependencias
  + Directorio **documentation**: Contiene la documentación del proyecto
  + **Requirements.txt**: Lista todas nuestras dependencias y sus versiones
  + .**Gitignore**: Archivos que no vamos a subir al repo de GitHub
  + .**Env**: Variables de entorno del sistema (datos sensibles)
  + **README.md**: Breve introducción para clonarnos el repo e iniciar el servidor
  + **Avance\_2.ipynb**: Consultas de SQL del avance 2 en un jupyter notebook
  + **Avance\_3.ipynb**: Consultas de SQL del avance 3 en un jupyter notebook

## Estructura de la aplicación

Como se mencionó anteriormente, el directorio src es quien encapsula la lógica de nuestra aplicación. Acá es donde tenemos:

1. Patrones de diseño implementados (Singleton, Factory)
2. Nuestra clase que nos conecta a la base de datos de MySQL
3. Una api endpoint desarrollado en FastApi para la carga de datos (ya sea de uno de los modelos o de todos los modelos)
4. Los modelos de nuestra base de datos
5. El pipeline de ingesta de datos de manera dinámica

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

# Solución implementada

Como mencionamos al inicio, la idea de este proyecto era capturar una información que se encontraba en archivos .csv, cargarla dinámicamente a la base de datos y ejecutar consultas sobre esta información (grosso modo).

Henry nos sugería hacer una inserción de datos directa. Yo, quise ir un paso más allá, y además de la inserción de datos con un load data infile en la base de datos de MySQL, le agregué validaciones pertinentes a cada modelo antes de cargarlo en la base de datos para así mantener limpia nuestra información desde el primer momento. Todo esta lógica de, capturar los datos, limpiarlos, validar su formato y luego cargarlos dinámicamente (pipeline) esta encapsulado en un api endpoint desarrollado con FastApi para que, el día de mañana, pensarlo como un sistema que se pueda integrar con otros sistemas para ejecutar esta lógica.

## Entendiendo la clase MySQLConnector

Esta clase está ubicada en: ***/src/database/MySQLConnector.py*** y es la responsable de gestionar toda la conexión con la base de datos MySQL. Es una clase fundamental en nuestro sistema, ya que permite ejecutar consultas, cargar datos de forma eficiente y mantener una única instancia de conexión activa mediante el patrón Singleton.

Esta clase se encarga de:

1. Establecer una conexión única a la base de datos utilizando los datos de entorno configurados.
2. Ejecutar consultas SELECT, INSERT, DELETE, UPDATE de manera directa o utilizando SQLAlchemy.
3. Cargar archivos CSV directamente a las tablas de la base de datos utilizando LOAD DATA INFILE.
4. Crear todas las tablas necesarias ejecutando un script SQL completo desde un archivo externo.

Pero ¿cómo hace todo esto? A continuación, detallamos los métodos y atributos que entran en acción para realizar toda esta lógica:

### Atributos de instancia

* *config*: Diccionario que contiene los datos de configuración de la base de datos extraídos del archivo .env (host, usuario, contraseña, puerto, etc.).
* *connection*: Objeto de conexión a MySQL proporcionado por mysql.connector. Se inicializa una única vez.
* *cursor*: Cursor de la conexión para ejecutar comandos SQL directamente.
* engine: Motor de SQLAlchemy creado con los datos de conexión para consultas que requieran alto rendimiento o integración con pandas.
* \_*initialized*: Bandera interna usada por el patrón Singleton para evitar múltiples inicializaciones.

### Métodos

* *\_\_new\_\_*: Implementación del patrón Singleton. Se asegura de que solo exista una única instancia activa de la clase en el sistema.
* *\_\_init\_\_*: Inicializa la configuración de conexión y define los atributos mencionados anteriormente. Solo se ejecuta una vez por instancia gracias al patrón Singleton.
* *connect*: Establece la conexión a la base de datos. Si la conexión ya está establecida, no hace nada. También crea el cursor y el engine de SQLAlchemy. Imprime en consola si la conexión fue exitosa o lanza una excepción en caso de error.
* *query\_select*: Ejecuta una consulta SELECT utilizando el cursor clásico. Devuelve los resultados como una lista de tuplas. Lanza una excepción si la base no está conectada.
* *query\_alchemy*: Ejecuta una consulta SQL utilizando SQLAlchemy y devuelve un Dataframe de pandas. Es útil para integraciones analíticas y trabajos con pandas. Lanza una excepción si el motor no está inicializado.
* *query\_insert\_delete\_update*: Ejecuta una consulta INSERT, DELETE o UPDATE utilizando el cursor clásico. Hace commit si se ejecuta correctamente o rollback en caso de error. Lanza una excepción si no hay conexión.
* *load\_data\_infile*: Carga los datos contenidos en un archivo CSV directamente en la tabla especificada utilizando LOAD DATA INFILE.  
  Pasos:
  1. Construye la sentencia SQL con los nombres de columnas del Dataframe.
  2. Ejecuta la sentencia.
  3. Hace commit si fue exitoso, o rollback e imprime un mensaje de error si algo falla.
* *create\_tables\_from\_sql\_file*: Recibe un path hacia un archivo SQL y ejecuta todas las sentencias contenidas en él para crear las tablas necesarias en la base.  
  Pasos:
  1. Verifica si el archivo existe.
  2. Lee el contenido y separa las sentencias por ;.
  3. Ejecuta cada sentencia.
  4. Hace commit al final si no hay errores.

### Justificación

La clase MySQLConnector centraliza toda la lógica necesaria para conectarse a la base de datos de forma segura, reutilizable y eficiente. Su diseño mediante el patrón Singleton garantiza que solo exista una única conexión activa a la base durante la ejecución del sistema, lo cual previene errores por múltiples accesos simultáneos, reduce el consumo de recursos y mejora el control sobre el estado de la conexión.

Además, encapsula distintos métodos especializados para ejecutar consultas SQL tradicionales, acceder a datos vía SQLAlchemy en Dataframes, realizar cargas masivas optimizadas mediante LOAD DATA INFILE y ejecutar scripts de creación de tablas. Esta abstracción permite desacoplar la lógica de acceso a datos del resto del sistema, facilitando la mantenibilidad, reutilización y escalabilidad del código. También estandariza el manejo de errores y transacciones, lo que contribuye a la robustez general de la aplicación.

## Entendiendo la clase DataIngestion

Esta clase está ubicada en***: /src/pipelines/DataIngestion.py*** y es la encargada de gestionar toda la ingesta de datos del csv, a MySQL. Es una de las clases principales de nuestro sistema, ya que se encarga de:

1. Crear la conexión a la base de datos (a través de la clase [MySQLConnector](#_Entendiendo_la_clase_1))
2. Cargar los datos de csv en memoria
3. Validar los datos que contiene el csv de acuerdo con el modelo que vamos a cargar en la base de datos
4. Finalmente, una vez validado los datos es la encargada de cargar dinámicamente estos datos en la base.

Pero ¿cómo hace todo esto? A continuación, detallamos los métodos y los atributos que entran en acción para realizar toda esta lógica:

### Atributos de instancia

* *model\_class* : Este atributo corresponde al modelo que vamos a cargar en nuestra base de datos
* *loader\_type*: Corresponde a el tipo de carga que vamos a ejecutar para pasarle a nuestra LoaderFactory.
* *database*: Recibe un objeto de la clase [MySQLConnector](#_Entendiendo_la_clase_1)

### Métodos

* *\_\_init\_\_* : Es el método encargado de instanciar nuestro objeto DataIngestion. Recibe por parámetros los datos para instanciar los [atributos](#_Atributos_de_instancia), declara y controla cuales modelos son aceptados para insertar y finalmente crea la conexión a la base de datos.
* *generate\_sql\_path:* Método estáticoquerecibe por parámetros el nombre del script de sql que tiene que buscar y nos retorna la ruta hacia ese script
* *generate\_externaldata\_path*: Método que en base al modelo que se quiere ingresar, retorna el path de la ubicación del archivo .csv para dicho modelo.
* *load\_external\_data*: Método que a partir del atributo loader\_type instanciado, llama a [DataLoaderFactory](#_DataLoaderFactory) para leer y retornar, según el tipo de loader, el dataframe de los datos
* *upload\_data*: Método final encargado de cargar la información a la base de datos donde previamente:

1. Creo las tablas en la base de datos a partir del script de sql
2. Llamo al método load external data para obtener el dataframe con los datos
3. Instancio la clase [DataValidationFactory](#_DataValidationFactory) para validar los datos según el modelo que se quiere cargar
4. Una vez validada la data, la carga dinámicamente a MySQL con un **LOAD DATA INFILE**

### Justificación

La clase [**DataIngestion**](#_Entendiendo_la_clase) encapsula toda la lógica necesaria para cargar datos externos en la base de datos de forma ordenada, reutilizable y validada. Centraliza procesos como la conexión a la base, la lectura de archivos, la validación según el modelo y la carga final, lo que facilita el mantenimiento y la escalabilidad del sistema. Su diseño modular permite integrar nuevos modelos o formatos sin modificar la lógica principal, asegurando integridad y consistencia en los datos cargados.

## Factories

En nuestro sistema decidí implementar dos veces el patrón de diseño **Factory** ya que permite encapsular la lógica de creación de objetos, facilitando la extensión y el mantenimiento del código. Este patrón nos brinda la flexibilidad de instanciar clases específicas en tiempo de ejecución según ciertos parámetros, sin acoplar directamente el código a implementaciones concretas.

La estructura del directorio está diseñada para ser **modular y escalable**, lo que facilita la incorporación de nuevas clases o la extensión de las fábricas existentes en el futuro. El contenido del directorio ***/src/factories/*** se organiza de la siguiente manera:

* Al nivel del directorio raíz (***/src/factories*/**) tenemos las clases que van a ser instanciadas de las factories (En este caso [DataLoaderFactory](#_DataLoaderFactory) y [DataValidationFactory](#_DataValidationFactory))
* Dentro de nuestro directorio raíz, tenemos dos carpetas:
  + **Abstract**: Contiene las clases base con métodos abstractos que definen la interfaz común de las fábricas.
  + **Concrete**: Incluye las implementaciones concretas (clases hijas) que extienden las clases abstractas, y que son retornadas dinámicamente por las fábricas según el contexto.

A screenshot of a computer program

AI-generated content may be incorrect.

### DataLoaderFactory

Implementa el patrón de diseño *Factory* para la creación dinámica de cargadores de datos. Según el tipo de fuente especificada (por ejemplo, "csv" o "json"), la factoría retorna una instancia adecuada para cargar el archivo correspondiente.

#### Estructura

* **DataLoaderFactory**  
  Fábrica principal. Según el parámetro source, devuelve una instancia de CSVLoader o JSONLoader.
* **DataLoader (abstracta)**  
  Define la interfaz load() que deben implementar todas las clases concretas.
* **CSVLoader / JSONLoader**  
  Implementaciones concretas que leen archivos usando pandas.

#### Ejemplo de uso

loader = DataLoaderFactory.create\_loader(source="csv", filepath="datos.csv")

df = loader.load()

#### Ventajas

* Permite agregar fácilmente nuevos tipos de cargadores (por ejemplo, XML, Excel).
* Desacopla la lógica de creación de objetos del código cliente.
* Mejora la mantenibilidad y la escalabilidad del sistema.

### DataValidationFactory

DataValidationFactory aplica el patrón de diseño *Factory* para retornar dinámicamente la clase adecuada para validar los datos según el modelo que se desea importar a la base de datos (como "categories", "employees", "customers", etc.). Cada clase concreta encapsula la lógica de validación específica de su estructura de datos.

#### Estructura

* **DataValidationFactory**  
  Expone un método *get\_validation\_class*(model: str, data: pd.DataFrame) que retorna una instancia de la clase de validación correspondiente al modelo.
* **DataValidation (abstracta)**  
  Clase base que define:
  + *validate*: método abstracto que debe implementar cada clase concreta.
  + Métodos comunes como *\_validate\_columns* y *\_validate\_duplicate\_ids*.
* **CategoryValidation / CityValidation / CustomerValidation / ...**  
  Clases concretas que extienden DataValidation e implementan:
  + Validaciones de columnas específicas.
  + Limpieza de datos según el tipo (strings, IDs, fechas, etc.).
  + Verificación contra registros ya existentes en la base de datos.

#### Ejemplo de uso

validator = DataValidationFactory.get\_validation\_class(model="customers", data=df)

validated\_data = validator.validate(db=mysql\_connector)

#### Ventajas

* **Modularidad**: cada clase maneja un tipo de datos, manteniendo la lógica desacoplada.
* **Escalabilidad**: agregar nuevas validaciones implica solo crear una nueva clase.
* **Reutilización**: métodos como validación de columnas e IDs se comparten desde la clase base.
* **Integridad**: permite filtrar datos duplicados, mal formateados o ya existentes en la base.

## FastApi

Para finalizar, toda esta estructuración del sistema, a partir la clase data ingestión que instancia la base de datos, las factorías, y se encarga de la validación de los datos antes de insertarlos a la base de datos del sistema, todo esto es llamado a través de un api endpoint desarrollado con FastApi, instanciando un servidor en uvicorn para poder en un futuro integrar esta lógica con algún otro sistema o bien, una interfaz gráfica.

### Endpoint principal

**POST /pipeline/upload/{model\_name}**

Carga los datos de un modelo específico (por ejemplo, customers, products, etc.) o de todos los modelos si se especifica all.

#### Parámetros:

* *Valores válidos*: categories, cities, countries, products, employees, customers, o all.
* *model\_name* (str, opcional en la ruta): nombre del modelo a procesar.

#### Entrada:

No requiere body. El archivo CSV correspondiente debe estar ubicado previamente en el sistema o ruta establecida para el modelo.

#### Respuestas:

* **200 OK:**  
  {  
   "message": "Data for customers was uploaded successfully to the database."  
  }  
  o  
  {  
   "message": "Data upload completed. Successfully processed models: products, customers.",  
   "warning": "The following models failed to upload data: employees."  
  }
* **404 Not Found:**  
  {  
   "detail": "Model 'invalid\_model' not found. Available models are categories, cities, countries, products, employees, customers"  
  }
* **500 Internal Server Error:**  
  {  
   "detail": "Failed to upload data for customers: <error\_details>"  
  }

### Lógica de Funcionamiento

**1. src/main.py**

* Crea la instancia de FastAPI.
* Registra el router del módulo api.py.

**2. src/app/api.py**

* Define la ruta /pipeline/upload/{model\_name}.
* Llama al método upload\_data() del controlador DataUploadController.

**3. src/app/controller.py ->** DataUploadController

* Mapea el nombre del modelo a su clase correspondiente mediante MODEL\_MAP.
* Usa la clase DataIngestion para:
  + Cargar datos desde archivos CSV.
  + Validar la estructura y contenido.
  + Insertar en base de datos si pasa todas las validaciones.
* Maneja errores por modelo individualmente.
* Soporta carga múltiple (model\_name = "all").

### Justificación

* **Separación de responsabilidades**
  + controller.py: maneja la lógica de negocio.
  + api.py: define los endpoints y enrutamiento.
  + main.py: configura y lanza la aplicación.
* **Extensibilidad**: para agregar nuevos modelos, basta con:
  + Crear su clase de validación.
  + Agregarla al MODEL\_MAP.
* Los errores son registrados con logger, y el sistema continúa procesando los modelos restantes si alguno falla.

## Jupyter Notebooks y sentencias de SQL

### CRUDQueryBuilder

Esta clase está ubicada en: **/src/database/CRUDQueryBuilder.py** y es la encargada de facilitar la construcción de sentencias SQL para las operaciones básicas de bases de datos: Crear (CREATE), Leer (READ), Actualizar (UPDATE) y Eliminar (DELETE). Es una clase fundamental para interactuar con la base de datos de manera programática, permitiendo generar consultas SQL dinámicamente sin necesidad de escribirlas manualmente.

Pero ¿cómo hace todo esto? A continuación, detallamos los métodos y los atributos que entran en acción para realizar toda esta lógica:

#### Atributos de instancia

* ***table***: El nombre de la tabla sobre la cual se realizará la operación SQL.
* ***query\_type***: El tipo de operación SQL a construir (puede ser "READ", "CREATE", "UPDATE" o "DELETE").
* ***filters***: Una lista de tuplas que representan las condiciones para la cláusula WHERE. Cada tupla contiene el nombre de la columna, el operador y el valor (ej: [("price", "BETWEEN", "1000 AND 1500")]).
* ***values***: Un diccionario que contiene los pares columna: valor para las operaciones de INSERT y UPDATE.
* ***aggregates***: Una lista de funciones de agregación (ej: "COUNT(\*)", "SUM(price)") para las consultas SELECT.
* ***query***: Almacena la sentencia SQL construida.

#### Métodos

* ***\_\_init\_\_***: Es el método encargado de instanciar nuestro objeto CRUDQueryBuilder. Recibe por parámetros el nombre de la tabla, el tipo de consulta, los filtros opcionales, los valores para insertar o actualizar y las agregaciones opcionales. Internamente, llama al método de construcción de consulta correspondiente según el query\_type.
* ***build\_where\_clause()***: Un método auxiliar que construye la parte WHERE de la sentencia SQL utilizando los filtros proporcionados. Si no hay filtros, devuelve una cadena vacía.
* ***build\_select()***: Construye la sentencia SQL SELECT. Si se especifican agregaciones, las incluye en la cláusula SELECT; de lo contrario, selecciona todas las columnas (\*). También incorpora la cláusula WHERE si hay filtros.
* ***build\_insert()***: Construye la sentencia SQL INSERT INTO. Requiere que el atributo values no esté vacío, ya que define las columnas y los valores a insertar.
* ***build\_update()***: Construye la sentencia SQL UPDATE. Requiere que el atributo values no esté vacío para definir las columnas y los nuevos valores. También incorpora la cláusula WHERE si hay filtros.
* ***build\_delete()***: Construye la sentencia SQL DELETE FROM. Incluye la cláusula WHERE si hay filtros para especificar qué filas eliminar.
* ***get\_query()***: Devuelve la cadena de texto con la sentencia SQL que ha sido construida por el objeto.

#### Justificacion

La clase CRUDQueryBuilder encapsula la lógica para generar consultas SQL de forma dinámica y segura, reduciendo la probabilidad de errores sintácticos y mejorando la legibilidad del código. Al centralizar la construcción de consultas, promueve la reutilización y facilita el mantenimiento. Su diseño permite adaptarse a diferentes escenarios de consulta (filtrado, agregación, inserción, actualización, eliminación) sin acoplar directamente el código a sentencias SQL hardcodeadas, lo que hace que el sistema sea más flexible y robusto.

### Avance 2 Notebook

Este notebook se conecta a una base de datos llamada henryapp-base utilizando la clase MySQLConnector. Luego, realiza varias operaciones de consulta de datos.

En primer lugar, consulta las primeras 5 filas de la tabla products y muestra el resultado.

Posteriormente, utiliza la clase CRUDQueryBuilder para construir y ejecutar una consulta que selecciona todos los clientes cuyos apellidos comienzan con la letra "C".

Finalmente, el notebook intenta usar CRUDQueryBuilder para contar los productos cuyo precio está entre 1000 y 1500, pero ejecuta la consulta de clientes nuevamente en lugar de la consulta de productos construida.

### Avance 3 Notebook

El notebook crea varios índices en la base de datos para mejorar el rendimiento de las consultas:

* **idx\_sales\_product\_id** en la tabla sales en la columna product\_id para optimizar las uniones con la tabla products.
* **idx\_sales\_sales\_person\_id** en la tabla sales en la columna sales\_person\_id para mejorar las uniones con la tabla employees.
* **idx\_products\_price\_name** en la tabla products en las columnas price y name para acelerar las búsquedas por estas columnas.
* **idx\_employees\_id** en la tabla employees en la columna employee\_id para acelerar las uniones.

Después de crear los índices, el notebook ejecuta tres consultas complejas utilizando Common Table Expressions (CTEs):

* La primera CTE identifica los 3 productos más vendidos por cantidad total.
* La segunda CTE encuentra los 3 mejores vendedores para el producto más caro.
* La tercera CTE determina la última venta realizada por cada vendedor.

Por último, el notebook crea una vista llamada view\_customer\_locations que combina información de las tablas customers, cities y countries para mostrar la ubicación completa de cada cliente. Luego, verifica que la vista se haya creado correctamente y la utiliza para consultar todos los datos de ubicación de los clientes.