



# UNIVERSIDAD NACIONAL DE COLOMBIA

---

## TRABAJO FINAL #4

---

En esta sesión especial transformamos el reto en oportunidad. Debido a ajustes en el cronograma institucional, el tiempo para evaluar se ha reducido, pero eso no nos detendrá.

Este taller ha sido diseñado con un enfoque práctico y guiado, para que durante la clase logres construir de forma progresiva lo que sería gran parte del trabajo final, usando una de las herramientas más potentes y utilizadas hoy en día en el mundo del análisis y la ingeniería de datos: dbt (**Data Build Tool**).

El objetivo es que aprendas haciendo, paso a paso, aplicando conceptos claves mientras desarrollas una solución funcional y bien estructurada. Esto no solo facilitará tu proceso de evaluación, sino que también potenciará tu experiencia y habilidades para el mundo laboral.

## REQUERIIENTOS

---

Para trabajar con dbt, es importante tener nociones básicas de los siguientes lenguajes y tecnologías:

- **Jinja** : Es un lenguaje de plantillas utilizado para generar texto dinámico, comúnmente usado en HTML o archivos de configuración. Permite incluir lógica como condicionales, ciclos y variables dentro de un archivo de texto.

- **YML** (YAML): Es un lenguaje de serialización de datos, utilizado para representar información estructurada de manera sencilla y legible. Es ampliamente usado para archivos de configuración por su claridad y formato anidado.
- **SQL** : Lenguaje de consulta estructurado (Structured Query Language), utilizado para acceder, manipular y transformar datos almacenados en bases de datos. Aunque existen otras alternativas, SQL sigue siendo el lenguaje estándar para operaciones sobre datos relacionales.

Aunque dbt puede incluir funciones en Python en algunas versiones, su esencia está centrada en realizar transformaciones exclusivamente con SQL, de manera modular y controlada.

En los materiales adjuntos encontrarán referencias para que puedan profundizar en estos conceptos por su cuenta más adelante.

## DUCKDB

---

### EXPLORACIÓN

1. DuckDB ofrece la opción de lanzar una interfaz gráfica ligera e integrada, ideal para trabajar con bases de datos embebidas o portables. Esta interfaz web permite ejecutar consultas SQL, explorar tablas, cargar archivos y visualizar resultados desde el navegador, sin necesidad de instalar herramientas externas.

```
duckdb -ui
```

2. Crea una base de datos en la locación que desees, se recomienda una carpeta del proyecto de linux, bien estructurada.

## DBT

---

### 1. ABRIR VISUAL STUDIO CODE (VS CODE)

Abre **VS Code** preferiblemente en la carpeta donde tienes tu proyecto en Linux o en una ubicación de tu preferencia.

💡 *TIP: dbt y DuckDB son herramientas portables, así que puedes trabajar desde cualquier carpeta sin necesidad de instalación global.*

## 2. VERIFICAR QUE DBT ESTÉ INSTALADO

Abre una terminal dentro de VS Code (o tu terminal favorita) y ejecuta:

```
dbt --version
```

## 3. CREAR UN NUEVO PROYECTO DBT

Ejecuta el siguiente comando (reemplaza nombre\_proyecto por el nombre que desees, sin espacios ni caracteres especiales):

```
dbt init nombre_proyecto
```

Este comando realizará dos cosas importantes:

1. Creará un archivo `profiles.yml` (si no existe) en una ubicación global ( `~/ .dbt/` o para windows `c:\Users\<TU_USUARIO>\.dbt\` ), el cual almacenará la configuración por defecto para conectarse a tu motor de base de datos, ya sea DuckDB, Databricks, Snowflake, etc.
2. Generará la estructura base del proyecto dbt, incluyendo carpetas como `models`, `tests`, y archivos como `dbt_project.yml`.

### ⚠ Warning

La configuración global en el archivo `profiles.yml` aplica para **todos los proyectos de dbt** en el equipo.

Si solo vas a trabajar con un proyecto, puedes usar esa configuración sin problema (por defecto se guarda en:

- `~/.dbt/` en Linux/macOS
- `C:\Users\<TU_USUARIO>\.dbt\` en Windows).

Sin embargo, si piensas automatizar con Lambdas, servidores, o manejar múltiples proyectos, la **mejor práctica** es evitar la configuración global y en su lugar incluir el archivo `profiles.yml` dentro del propio proyecto de dbt.

## 4. CAMBIAR RUTA POR DEFECTO DE DBT

Para evitar depender de una configuración global, por buenas prácticas configuraremos el proyecto para que use un perfil ubicado en la misma carpeta de trabajo. Para esto, se debe definir una variable de entorno.

1. Crear la carpeta `.dbt` dentro del proyecto de dbt.
2. Copia la ruta absoluta de esa carpeta.
3. Si tienes Linux o macOS y sabes cómo configurar variables de entorno en tu `.bash_profile`, `.bashrc` o `.zshrc`, simplemente agrega:

```
export DBT_PROFILES_DIR=tu_ruta/.dbt
```

En caso de que no tengas experiencia configurando eso, crea en la raíz del proyecto una carpeta llamada `.vscode`, y dentro de ella un archivo llamado `settings.json` con el siguiente contenido:

```
{
  "terminal.integrated.env.osx": {
    "DBT_PROFILES_DIR": "tu_ruta/.dbt"
  }
}
```

## 💡 Tip

"terminal.integrated.env.osx" → solo funciona en macOS

"terminal.integrated.env.linux" → solo funciona en Linux

"terminal.integrated.env.windows" → solo funciona en Windows

## 5. CONFIGURAR CONEXIÓN

1. El primer paso al trabajar con dbt, después de crear el proyecto, es modificar el archivo `profiles.yml` (<https://docs.getdbt.com/docs/core/connect-data-platform/connection-profiles>) para establecer la conexión con el motor de base de datos que se va a utilizar (por ejemplo, *DuckDB*, *Databricks*, *Snowflake*, etc.). Para ello debemos realizar lo siguiente:

2. crear el archivo `profiles.yml` y agregar el siguiente contenido:

```

name: # nombre del perfil (debe coincidir con el especificado en dbt_project.yml
target: dev # entorno activo por defecto (puede ser 'dev', 'prod', etc.)
outputs: # define las salidas o configuraciones de conexión
  dev: # nombre del entorno de salida (debe coincidir con 'target')
    type: duckdb # tipo de conector: en este caso, DuckDB
    path: 'path/db.duckdb' # ruta al archivo .duckdb (puede ser relativa o a
    extensions: # lista de extensiones que se cargarán al iniciar la conexión
      - httpfs # Permite leer archivos desde URLs remotas (como HTT
      - parquet # Permite leer y escribir archivos Parquet directame

```

3. Cambiar el nombre del perfil por el que desees.

4. Luego, abre el archivo `dbt_project.yml`, busca la propiedad `profile` y asígnale, entre comillas simples, el mismo nombre que definiste en `profiles.yml`.

5. Para validar que la conexión esté correctamente configurada, ejecutar:

```
dbt debug
```

## 💡 TIPS

1. Si tienes varios perfiles o varios targets definidos en tu archivo `profiles.yml` y deseas hacer debug de una combinación específica, puedes usar:

```
dbt debug --profile profile_name --target target_name
```

Esto te permite verificar que la configuración de conexión para > ese perfil y ese entorno específico (target) está correctamente > definida y funcionando.

2. dbt permite usar variables de entorno dentro del archivo `profiles.yml` para que la configuración sea más segura, reutilizable y portátil, especialmente útil para entornos como servidores, CI/CD o cuando manejas secretos.:

```
mi_perfil:
  target: dev
  outputs:
    dev:
      type: duckdb
      path: "{{ env_var('DUCKDB_PATH') }}"
      extensions:
        ...
```

## 6. PROBICIONAR BASE DE DATOS DE DUCKDB

1. En este paso vamos a crear las estructuras necesarias y cargar datos en DuckDB. Puedes utilizar uno de los archivos que ya hayas generado anteriormente (esto también te permitirá verificar si tus datos están bien estructurados), o usar el siguiente archivo de ejemplo. Guárdalo con el nombre `consolidado.log` en la raíz del proyecto:

```
1 | date;mission;device_type;device_status;hash
2 | 120625201441;CLNM;satelite;killed;MTIwNjI1MjAxNDQxCg==
3 | 120625201441;ORBONE;satelite;killed;MTIwNjI1MjAxNDQxCg==
4 | 120625201441;UNKN;nave;faulty;MTIwNjI1MjAxNDQxCg==
5 | 120625201441;UNKN;sonda;good;MTIwNjI1MjAxNDQxCg==
```

2. Crea un archivo llamado `app.py` con el siguiente contenido:



```
1 import sys
2 import duckdb
3 import traceback
4 import pandas as pd
5
6
7
8 # se toman los valores de la configuración y archivos calculados
9 csv_file_path: str = sys.argv[1]
10 csv_delimeter: str = sys.argv[2]
11 database_path: str = sys.argv[3]
12 table_name: str = sys.argv[4]
13
14 connection = None
15
16 try :
17     # configurar conexión simple
18     connection = duckdb.connect(database_path)
19
20     # crea los esquemas del patron llamado medallion architecture
21     # mirar: https://www.databricks.com/glossary/medallion-architecture
22     connection.execute("CREATE SCHEMA IF NOT EXISTS bronze;")
23     connection.execute("CREATE SCHEMA IF NOT EXISTS silver;")
24     connection.execute("CREATE SCHEMA IF NOT EXISTS gold;")
25
26     # se va a realizar un pipeline incremental, los datos generados necesi
27     # unico identificador, por eso la forma mas simple es aplicarle secuen
28     # https://duckdb.org/docs/stable/sql/statements/create_sequence.html
29     connection.execute("CREATE SEQUENCE IF NOT EXISTS bronze.seq_events;")
30
31     # se crea events como una tabla para guardar historicos, se pone date
32     # para no perder los 0 de adelante, mas adelante esto se pasara a fech
33     # event_id sera el valor de la secuencia
34     # DEFAULT nextval('bronze.event_seq') forma sencilla de asignar una se
35     connection.execute(f"""
36         CREATE TABLE IF NOT EXISTS {table_name}(
37             event_id INTEGER NOT NULL PRIMARY KEY DEFAULT nextval('bronze.
38             date STRING NOT NULL,
39             mission STRING NOT NULL,
40             device_type STRING NOT NULL,
41             device_status STRING NOT NULL,
42             hash STRING NOT NULL
43         );
44     """)
45
46
47     df = pd.read_csv(csv_file_path, sep=csv_delimeter)
48     # registrar el DataFrame como una tabla temporal en memoria
49     connection.register('temp_df', df)
50
51     # insertar datos sin tocar el event_id (automaticamente la tabla alime
52     connection.execute(f"""
53         INSERT INTO {table_name} (date, mission, device_type, device_statu
```



```

54         SELECT date, mission, device_type, device_status, hash FROM temp_d
55         """)
56     except Exception as ex:
57         print(ex)
58         traceback.print_exc()
59         sys.exit(-1) # marcar salida como erronea
60
61     finally:
62         if connection is not None:
63             connection.close()

```

3. Crear un pequeño script de bash para verificar rápidamente que todo funcione. Más adelante, este proceso se integrará en tu proyecto final con dbt. Crea un archivo llamado:

Para linux y Mac `cargar.sh`

```

#!/bin/bash

# ruta del archivo consolidado
file="$(pwd)/consolidado.log"

# Ejecutar el script Python que carga los datos
python app.py $file ";" "ruta_db" "bronze.events"

```

Para windows `cargar.bat`

```

@echo off
set FILE=%cd%\consolidado.log
python app.py %FILE% ";" "ruta_db" "bronze.events"

```

Ejecutamos el programa para verificar que todo funcione bien y tener datos en DuckDB

## 7. ANALIZAR Y ENTENDER DATOS

La práctica que les he recomendado insistentemente es simple pero poderosa: primero se debe entender la información, luego se implementa la solución. En el mundo de los datos, este principio es fundamental.

Antes de aplicar transformaciones, modelos o reportes, es imprescindible comprender la naturaleza, estructura y contexto de los datos. Solo así se pueden tomar decisiones efectivas y evitar errores innecesarios.

1 . Analizar datos almacenados:

```
SELECT * FROM <alias>.bronze.events
```

## 2. Analizar como solucionar problemas:

```
SELECT
    STRPTIME(date, '%d%m%y%H%M%S') AS date,
FROM <alias>.bronze.events
```

## 3. Funciones de tiempo

```
extract(YEAR FROM date) AS event_year,      -- año (ej. 2025)
extract(MONTH FROM date) AS event_month,    -- mes (1-12)
extract(DAY FROM date) AS event_day,        -- día del mes (1-31)
extract(HOUR FROM date) AS event_hour,      -- hora (0-23)
extract(MINUTE FROM date) AS event_minute,  -- minuto (0-59)
extract(SECOND FROM date) AS event_second,  -- segundo (0-59)
dayofweek(date) AS week_day,               -- día de la semana (1 = domingo,
dayofyear(date) AS year_day,               -- día del año (1-366)
decade(date) AS decade_event,             -- década (ej. 2020)
era(date) AS era_event,                   -- era (1 = d.C., 0 = a.C.)
quarter(date) AS quarter_event,           -- trimestre (1-4)
weekofyear(date) AS year_week,            -- semana del año (1-53)
UPPER(strftime(date, '%A')) AS day_name,   -- nombre del día en mayúsculas (
UPPER(strftime(date, '%B')) AS month_name, -- nombre del mes en mayúsculas (
UPPER(strftime(date, '%p')) AS time_indicator, -- indicador AM/PM en mayúsculas
strftime(date, '%d/%m/%Y %H:%M:%S') AS format_event_date -- Fecha con formato
```

## 4. Cuando trabajamos con fechas en SQL, como es el caso de nuestra base

`<alias>.bronze.events`, es común que tengamos que transformar o convertir el campo `date` para que pueda ser usado con funciones asociadas al tipo de dato. Si no usamos una estrategia adecuada, terminamos repitiendo esa transformación muchas veces en una misma consulta, lo cual, no solo hace que el código sea largo y difícil de mantener, sino que también puede causar errores.

### ¿Cómo lo solucionamos? ¿Que se les ocurre?

► Haz clic aquí para ver la solución

## 8. CREANDO NUESTROS PRIMEROS MODELOS

Antes de entrar en el modelamiento, es importante hacer un pequeño recordatorio de conceptos muy elementales. Para esto, nos dirigiremos al archivo `intro.dbt.html`, específicamente a la sección **FUNCIONES CLAVES**, donde se resumen funciones fundamentales que nos serán útiles más adelante.

Después de validarlo, continuamos con el proceso de construcción de modelos en dbt.

1. Nos dirigimos a la carpeta `models/` de tu proyecto dbt.
2. Creamos (o editamos) el archivo llamado `sources.yml`.
3. Agregamos el siguiente contenido para registrar la tabla fuente:

```
version: 2

sources:
  - name: dlake
    description: this is the duckdb source related to events
    schema: bronze
    tables:
      - name: events
        description: NASA events related to different devices
```

4. Luego, nos dirigimos a la carpeta `analyses/`, la cual está diseñada para guardar consultas analíticas temporales o exploratorias que no forman parte directa del pipeline de transformación.

5. Dentro de esta carpeta, crea un archivo `.sql` con cualquier nombre, por ejemplo:

```
SELECT *
FROM {{ source('dlake', 'events') }}
LIMIT 10;
```

6. Al utilizar `source()` y `ref()` en dbt, no tenemos que preocuparnos por la ubicación de los archivos `.sql` o de configuración `.yml`. Por esta razón, y para mantener una estructura clara, vamos a crear las siguientes carpetas dentro de `models/`:

```
├─ models
│   ├── marts
│   ├── reports
│   ├── source
│   └── staging
```

## 7. Vamos a crear nuestro primer modelo.

Dirígete a la carpeta `models/source/`, crea un archivo llamado `src_events.sql` y agrega el siguiente contenido:

```
{{ config(
    materialized='ephemeral',
) }}

SELECT
    event_id,
    date,
    mission,
    device_type,
    device_status,
    hash
FROM {{ source('dlake', 'events') }}
```

 Mirar materialización: <https://docs.getdbt.com/docs/build/materializations> (<https://docs.getdbt.com/docs/build/materializations>)

## 8. Luego, vamos a la carpeta `models/staging/` y creamos el archivo `stg_events.sql`, y le agregamos las siguientes cosas:

- Configuración de materialización:

```
{{ config(
    materialized='incremental',
    unique_key='event_id',
    schema='silver'
) }}
```

- Filtro incremental

```
WHERE
    {% if is_incremental() %}
        event_id > (SELECT MAX(event_id) FROM {{ this }})
    {% else %}
        1=1
    {% endif %}
```

- Reemplazamos la tabla de origen por:

```
{{ ref('src_events') }}
```

## 🔔 TIP

Este paso se puede hacer directamente en `source`, pero lo pongo como forma ilustrativo en caso de que tengan que hacer procesamiento extensivo a la fuente antes de ser consumida.

9. Para ejecutar los modelos, simplemente corre:

```
dbt run
```

10. Seguramente a muchos (por no decir a todos) les aparecerá que los datos se están almacenando en un esquema incorrecto. Esto ocurre porque algunas veces dbt puede tener conflictos con el esquema definido o con la forma en que gestiona los esquemas por defecto. Para solucionarlo hay dos formas:

A. Ajustar la configuración del archivo `dbt_project.yml`, dentro del bloque `models:`, agrega o ajusta el nombre del esquema (por ejemplo `bronze`):

```
models:
  tu_nombre_de_proyecto:
    +schema: bronze
```

B. Reescribiendo la macro que viene por defecto en dbt. Para esto dirígete a la carpeta `macros/` > crea un archivo llamado `solve_schema.sql` > agrega el siguiente contenido:

```
{% macro generate_schema_name(custom_schema_name, node) -%}
  {{ custom_schema_name if custom_schema_name is not none else target.schema }}
{% endmacro %}
```



Iniciemos con la forma **B** y nuevamente ejecutamos.

11. Finalmente, vamos a empezar poco a poco a generar documentación del proyecto. Para esto, simplemente ejecuta en la terminal:

```
dbt docs generate
```

Luego, abre o ve a la terminal y dirígete a la sección llamada **LINEAGE** , donde podrás visualizar cómo se conectan tus modelos y fuentes.

## 9. GENERANDO LOS REPORTE

1. Ya tenemos nuestra tabla depurada, y como pueden ver, todo lo que hemos hecho empieza a tomar más sentido.

Ahora contamos con los campos listos para trabajar y generar tablas enriquecidas que nos servirán para reportes y análisis. iniciaremos creando los siguientes archivos:

1. Consultas basadas en los scripts de Linux. Estos archivos corresponden a las **consultas SQL que generaron en el apolo-11**.

Simplemente deben copiarlas y reemplazar la tabla fuente por la referencia al modelo `FROM {{ ref('stg_events') }}` , por ejemplo:

```
fct_percentages.sql    --> APLSTATS-PERCENTAGES
fct_missions.sql       --> APLSTATS-MISSIONS
fct_disconnections.sql --> APLSTATS-DISCONNECTIONS
fct_events.sql         --> APLSTATS-EVENTS
```

2. Estos archivos estarán enfocados en analizar la información que generamos en el modelo de staging con los campos derivados de la fecha; crea los archivos y adiciona las consultas relacionadas a semanas, trimestres, días, etc.

- `fct_day_events.sql`

```
SELECT
  day_name,
  COUNT(*) AS total_events
FROM {{ ref('stg_events') }}
GROUP BY day_name
ORDER BY 2
```

- `fct_event_days.sql`

```
SELECT
  event_year,
  day_name,
  COUNT(*) AS total_events
FROM {{ ref('stg_events') }}
GROUP BY event_year, day_name
ORDER BY 3
```

- fct\_operational\_duration.sql

```
SELECT
    mission,
    MIN(real_event_date) AS first_event,
    MAX(real_event_date) AS last_event,
    DATEDIFF('day', MIN(real_event_date), MAX(real_event_date)) AS active_days,
    COUNT(*) AS total_events,
    COUNT(*) * 1.0 / NULLIF(DATEDIFF('day', MIN(real_event_date), MAX(real_event_date)), 0) AS active_days_ratio
FROM {{ ref('stg_events') }}
GROUP BY mission
ORDER BY active_days DESC
```

- fct\_week\_tendencies.sql

```
SELECT
    year_week,
    COUNT(*) AS total_events
FROM {{ ref('stg_events') }}
GROUP BY year_week
ORDER BY year_week
```

- fct\_simultaneous\_events.sql

```
SELECT
    mission,
    event_hour,
    event_minute,
    event_second,
    COUNT(*) AS simultaneous_events
FROM {{ ref('stg_events') }}
GROUP BY mission, event_hour, event_minute, event_second
HAVING COUNT(*) > 1
ORDER BY simultaneous_events DESC
```

3. Estos archivos deben contener consultas que usen funciones de ventana disponibles en DuckDB (como row\_number, rank, lead, lag, etc.).  
El proceso es el mismo: crea el archivo y agrega el contenido según vayas desarrollando la lógica.

- \* fct\_ranks.sql

```

SELECT *
FROM (
    SELECT
        mission,
        device,
        COUNT(*) AS total,
        ROW_NUMBER() OVER (PARTITION BY mission ORDER BY COUNT(*) DESC) AS 1
    FROM {{ ref('stg_events') }}
    GROUP BY mission, device
) ranked

```

◦ fct\_mission\_analysis.sql

```

SELECT
    format_event_date,
    mission,
    status,
    SUM(CASE WHEN status = 'FAULTY' THEN 1 ELSE 0 END)
        OVER (PARTITION BY event_year, event_month, event_day, mission) AS 1
    SUM(CASE WHEN status = 'KILLED' THEN 1 ELSE 0 END)
        OVER (PARTITION BY event_year, event_month, event_day, mission)
    SUM(CASE WHEN status = 'UNKNOWN' THEN 1 ELSE 0 END)
        OVER (PARTITION BY event_year, event_month, event_day, mission)
    SUM(CASE WHEN status = 'WARNING' THEN 1 ELSE 0 END)
        OVER (PARTITION BY event_year, event_month, event_day, mission)
    SUM(CASE WHEN status = 'GOOD' THEN 1 ELSE 0 END)
        OVER (PARTITION BY event_year, event_month, event_day, mission)
    SUM(CASE WHEN status = 'EXCELLENT' THEN 1 ELSE 0 END)
        OVER (PARTITION BY event_year, event_month, event_day, mission)
    COUNT(*) OVER (PARTITION BY event_year, event_month, event_day, mission)
FROM {{ ref('stg_events') }}

```

### ⚠ OJOO!

Ahora ya tenemos casi todos los modelos que irán en la zona Oro, listos para ser distribuidos y utilizados según nuestras necesidades. Sin embargo, hay un problema en los modelos... Tómense un momento para analizar qué puede estar faltando.

► [Haz clic aquí para ver la solución](#)

## 10. GENERANDO REPORTES MULTI IDIOMA

Como bien sabemos, tener las tablas en la zona **Oro** (Gold), sin importar el enfoque, representa datos limpios, refinados y listos para ser consumidos.



Ahora que ya tenemos todas las tablas depuradas, podemos generar nuevos reportes a partir de ellas. En este caso, implementaremos una estrategia para soportar reportes multi idioma.

1. Definir materialización y esquema para la zona de reportes. Vamos a indicar que todos los modelos ubicados en la carpeta `reports/` se deben materializar como `view` y guardar en el esquema `gold`. En tu archivo `dbt_project.yml`, agrega lo siguiente dentro de `models:`:

```
models:
  tu_nombre_de_proyecto:
    reports:
      +materialized: view
      +schema: gold
```

2. Crear tabla de traducciones (lookup table): A veces se usan pequeñas tablas llamadas **lookups** que contienen catálogos o datos auxiliares. Estas no suelen cambiar mucho, pero deben mantenerse versionadas junto al proyecto dbt.

2.1. Crea un archivo CSV llamado `translate.csv` en la carpeta `seeds/`

2.2. Agrega el siguiente contenido:

```
ENGLISH_DAY, MANDARIN, SPANISH, FRENCH, GERMAN
MONDAY, 星期一, LUNES, LUNDI, MONTAG
TUESDAY, 星期二, MARTES, MARDI, DIENSTAG
WEDNESDAY, 星期三, MIÉRCOLES, MERCREDI, MITTWOCH
THURSDAY, 星期四, JUEVES, JEUDI, DONNERSTAG
FRIDAY, 星期五, VIERNES, VENDREDI, FREITAG
SATURDAY, 星期六, SÁBADO, SAMEDI, SAMSTAG
SUNDAY, 星期日, DOMINGO, DIMANCHE, SONNTAG
```

3. Configurar almacenamiento de los seeds: Los archivos `seeds` se convierten en tablas dentro de dbt, por lo tanto también debemos indicar en qué esquema deben almacenarse.

Agrega lo siguiente en tu archivo `dbt_project.yml`:

```
seeds:
  tu_nombre_de_proyecto:
    translate:
      +schema: gold
```

💡 Al estar versionado en Git, este archivo puede ser extendido, revertido o auditado fácilmente.

4. Una vez creado el archivo `translate.csv` en la carpeta `seeds/`, debes registrarlo en tu base ejecutando el siguiente comando:

```
dbt seed
```

5. Finalmente, procedemos a crear los modelos restantes que utilizarán tanto las tablas en oro como la tabla `translate` para generar reportes multi idioma. Recuerda ubicar estos modelos en la carpeta `models/reports/` y utilizar correctamente las referencias con `{{ ref('...') }}` para asegurar la trazabilidad en el DAG de dbt.

- `int_french.sql`

```
SELECT
fed.event_year,
day_map.FRENCH AS translated_day,
total_events
FROM {{ ref('fct_event_days') }} AS fed
INNER JOIN {{ ref('translate') }} AS day_map
    ON UPPER(fed.day_name) = day_map.ENGLISH_DAY
ORDER BY total_events DESC
```

- `int_german.sql`

```
SELECT
    mission,
    event_hour,
    event_minute,
    event_second,
    COUNT(*) AS simultaneous_events
FROM {{ ref('stg_events') }}
GROUP BY mission, event_hour, event_minute, event_second
HAVING COUNT(*) > 1
ORDER BY simultaneous_events DESC
```

- `int_mandarin.sql`

```
SELECT
    fed.event_year,
    day_map.MANDARIN AS translated_day,
    total_events
FROM {{ ref('fct_event_days') }} AS fed
INNER JOIN {{ ref('translate') }} AS day_map
    ON UPPER(fed.day_name) = day_map.ENGLISH_DAY
ORDER BY total_events DESC
```

- `int_spanish.sql`

```
SELECT
    fed.event_year,
    day_map.SPANISH AS translated_day,
    total_events
FROM {{ ref('fct_event_days') }} AS fed
INNER JOIN {{ ref('translate') }} AS day_map
    ON UPPER(fed.day_name) = day_map.ENGLISH_DAY
ORDER BY total_events DESC
```

6. Ejecutar `dbt run` para compilar todo

## 11. MODULARIZAR PARA NO REPETIR TANTO

Ya que ha pasado la euforia (o el miedo) de que las cosas no funcionen, y ahora que todo está corriendo correctamente, es momento de analizar qué se puede mejorar para evitar redundancias y preparar el proyecto para futuras implementaciones.

### ¿Qué creen ustedes que se puede mejorar?

Tómense un momento para revisar los SQL...

► Haz clic aquí para ver la solución

## 12. GENERAR PRUEBAS

A veces no es necesario empezar desde cero, dbt permite importar paquetes de terceros para reutilizar macros, tests, modelos y configuraciones útiles desarrolladas por la comunidad. Para explorar la variedad de paquetes disponibles, puedes visitar:

<https://hub.getdbt.com/> (<https://hub.getdbt.com/>)

1. Una vez conoces los paquetes disponibles, puedes instalarlos fácilmente agregando la dependencia en el archivo `packages.yml` de tu proyecto. Por ejemplo vamos a validar que la columna:

```
packages:
  - package: metaplane/dbt_expectations
    version: 0.10.8
```

Luego ejecuta para instalar el paquete:

```
dbt deps
```

🔗 si el archivo `packages.yml` no existe, crealo.

2. Ahora nos dirigimos al archivo `test/models/sources.yml`. Sabemos que dentro de nuestro modelo tenemos una columna llamada `device_status`. No es que desconfíe de ustedes... pero por si las moscas, quiero asegurarnos de que esta columna solo contenga los valores permitidos.

Para eso, vamos a agregar la siguiente información dentro de la definición de la tabla `events`:

```
columns:
  - name: device_status
    description: "Status of the device in the raw source."
    tests:
      - dbt_expectations.expect_column_to_exist
      - accepted_values:
          values: ['faulty', 'killed', 'unknown', 'warning', 'good', 'excel']
```

## 13. EL TERROR DE TODO DESARROLLADOR, DOCUMENTAR!

Hemos llegado al final del desarrollo del taller, pero aún nos falta algo importante (y lo más ignorado): la documentación. Sí, esa misma que todos reclaman cuando entran a un proyecto..., las frases típicas:

- ¡No entiendo nada, esto no tiene documentación!
- ¿Quien me puedes capacitar?
- Esto solo lo entiende dios y el que lo desarrollo.

Pero cuando es su turno de hacerla, **se hacen los locos y mágicamente se les olvida**

### ¿Por qué es importante documentar?

Porque una solución de datos sin documentación es como una nave espacial sin manual: puede despegar, pero nadie sabe a dónde va ni cómo mantenerla.

Documentar ayuda a:

- Entender el propósito de cada modelo o campo
- Facilitar el mantenimiento y la colaboración
- Reducir errores futuros
- Agilizar la incorporación de nuevos integrantes al equipo

Así que sí, ahora nos toca hacer lo que nadie quiere, pero todos agradecen:

1. En la carpeta `macros/` , crea un archivo llamado `macros_schema.yml` y agrega el siguiente contenido:

version: 2

macros:

- name: date\_info  
description: >  
Generates multiple derived date and time columns from a single input (arguments:
  - name: input\_date  
description: The column or expression representing the date to be pa
- name: generate\_schema\_name  
description: >  
Overrides the default dbt behavior for generating schema names. Return arguments:
  - name: custom\_schema\_name  
description: Custom schema name to use, can be `null`.
  - name: node  
description: dbt model node (required internally by dbt for context)
- name: status\_counters\_by\_partition  
description: >  
Creates a set of windowed counters for predefined status values over : This macro helps analyze the distribution of statuses within partition arguments:
  - name: status\_column  
description: The column name that contains status values (e.g., 'sta
  - name: partition\_by\_columns  
description: A list of column names to partition by in the window fi
- name: get\_view\_by\_language  
description: >  
Macro that generates views by language to avoid repeating the same que every time that a new language is added.  
arguments:
  - name: language  
description: Language code that corresponds to a column in the seed



2. En la carpeta `models/` , debes agregar un archivo `_schema.yml` dentro de cada subdirectorio correspondiente.

- `models/staging/schema.yml` :

version: 2

models:

- name: stg\_events

description: >

Staging model for events. This table contains enriched and formatted r for each event, including temporal breakdown, mission context, device and operational status.

columns:

- name: event\_id

description: "Unique identifier for each event."

- name: event\_year

description: "Year in which the event occurred."

- name: event\_month

description: "Month of the event."

- name: event\_day

description: "Day of the month on which the event occurred."

- name: event\_hour

description: "Hour of the day (0-23) when the event occurred."

- name: event\_minute

description: "Minute in which the event was recorded."

- name: event\_second

description: "Second in which the event was recorded."

- name: week\_day

description: "Numeric representation of the day of the week (1 = Sunday, 2 = Monday, etc.)."

- name: year\_day

description: "Day of the year (1-366) on which the event occurred."

- name: decade\_event

description: "Decade in which the event occurred (e.g., 202 for the 2020s)."

- name: era\_event

description: "Era or epoch classification of the event (e.g., 1 for the 21st century)."

- name: quarter\_event

description: "Calendar quarter in which the event occurred (1-4)."

- name: year\_week

description: "ISO week number of the year (1-53)."

- name: day\_name

description: "Name of the weekday in uppercase letters (e.g., MONDAY, TUESDAY, etc.)."

- name: month\_name

description: "Name of the month in uppercase letters (e.g., JANUARY, FEBRUARY, etc.)."

- name: time\_idicator

description: "AM/PM indicator based on the event timestamp."

- name: format\_event\_date

description: "Formatted version of the event timestamp as string (datetime format)."

- name: real\_event\_date

description: "Actual event timestamp with full datetime precision."

- name: mission

description: "Name or code of the mission related to the event."

- name: device

description: "Type of device associated with the event."

- name: status

description: "Operational status of the device at the time of the ev

- models/source/source\_schema.yml :

version: 2

models:

- name: src\_events

schema: bronze

description: >

Staging model for events, containing raw records from data lake.

Includes metadata such as mission identifier, device status, and hash

columns:

- name: event\_id

description: "Unique identifier for each event record."

- name: date

description: "Date when the event occurred."

- name: mission

description: "Code or name of the mission related to the event."

- name: device\_type

description: "Type or category of the device involved in the event."

- name: device\_status

description: "Operational status of the device at the time of the ev

- name: hash

description: "Hash value used for verifying data integrity or dedup



- models/marts/marts\_schema.yml :



version: 2

models:

- name: fct\_day\_events  
description: >  
Fact table that summarizes the number of unknown status events per mission and device. Used to monitor device connectivity or data quality.  
columns:
  - name: mission  
description: "Name or identifier of the mission associated with the events."
  - name: device  
description: "Type of device for which the unknown events were recorded."
  - name: number\_unknown  
description: "Total number of events with 'UNKNOWN' status for the given mission and device."
- name: fct\_disconnections  
description: >  
Fact table that tracks the number of disconnection events, identified by mission and device, for each mission and device. Useful for evaluating system reliability and detecting communication failures.  
columns:
  - name: mission  
description: "Identifier or name of the mission associated with the disconnections."
  - name: device  
description: "Type or category of the device that experienced disconnections."
  - name: number\_unknown  
description: "Count of events where the device reported an 'UNKNOWN' status due to disconnections."
- name: fct\_event\_days  
description: >  
Fact table that aggregates the total number of events per day of the week. Useful for identifying patterns and trends in event frequency based on the day of the week.  
columns:
  - name: event\_year  
description: "Year in which the events occurred."
  - name: day\_name  
description: "Name of the day of the week (e.g., MONDAY, TUESDAY)."
  - name: total\_events  
description: "Total number of events recorded on that day of the week for the given year."
- name: fct\_events  
description: >  
Fact table that summarizes the number of events by mission, device type, and operational status. Useful for analyzing event volume and device performance.  
columns:
  - name: mission  
description: "Identifier or name of the mission associated with the events."
  - name: device  
description: "Type or category of the device involved in the event."
  - name: status  
description: "Operational status reported for the device during the event."
  - name: number\_events  
description: "Total number of events for the given combination of mission, device, and status."

- name: fct\_mission\_analysis  
description: >  
Fact table that provides a daily operational breakdown of mission performance. It includes the count of events by status category and the total number of events per mission and date. This model helps identify operational trends, across columns:
  - name: format\_event\_date  
description: "Formatted string representing the date and time of the event"
  - name: mission  
description: "Identifier or name of the mission involved in the event"
  - name: status  
description: "Reported status of the device at the time of the event"
  - name: total\_faulty  
description: "Number of events with a 'FAULTY' status for the given mission"
  - name: total\_killed  
description: "Number of events where the device was reported as 'KILLED' for the mission"
  - name: total\_unknown  
description: "Number of events with 'UNKNOWN' status, indicating potential issues"
  - name: total\_warning  
description: "Number of events flagged as 'WARNING' for the mission"
  - name: total\_excellent  
description: "Number of events with an 'EXCELLENT' status indicating optimal performance"
  - name: total\_events\_mission  
description: "Total number of events recorded for the mission on the given date"
- name: fct\_missions  
description: >  
Fact table that summarizes the number of inoperable devices per mission. Inoperable devices include those with statuses such as 'FAULTY', 'KILLED', or 'UNKNOWN'. This model supports reliability and operational health analysis across columns:
  - name: mission  
description: "Name or code of the mission associated with the device"
  - name: inoperable\_devices  
description: "Total count of devices with inoperable status (FAULTY, KILLED, UNKNOWN) for the mission"
- name: fct\_operational\_duration  
description: >  
Fact table that summarizes the operational duration of each mission. It includes the first and last event timestamps, number of active days, total events, and average events per day. Useful for evaluating mission efficiency across columns:
  - name: mission  
description: "Name or identifier of the mission."
  - name: first\_event  
description: "Timestamp of the first recorded event for the mission"
  - name: last\_event  
description: "Timestamp of the last recorded event for the mission."
  - name: active\_days  
description: "Number of days between the first and last event (inclusive)"
  - name: total\_events  
description: "Total number of events recorded for the mission."

- name: avg\_events\_per\_day
  - description: "Average number of events per active day for the mission"
- name: fct\_percentages
  - description: >  
Fact table that calculates the percentage of events by mission and device in relation to the total number of events. Useful for understanding the distribution of activity across missions and device types.
  - columns:
    - name: mission
      - description: "Name or identifier of the mission."
    - name: device
      - description: "Type or category of the device."
    - name: percentage
      - description: "Percentage of total events represented by the given mission and device combination"
- name: fct\_ranks
  - description: >  
Fact table that ranks missions and devices based on the total number of events. Useful for identifying the most active mission-device combinations and their relative performance.
  - columns:
    - name: mission
      - description: "Name or identifier of the mission."
    - name: device
      - description: "Type or category of the device."
    - name: total
      - description: "Total number of events for the mission and device combination"
    - name: rank
      - description: "Dense rank of the mission-device combination based on the total number of events"
- name: fct\_simultaneous\_events
  - description: >  
Fact table that identifies simultaneous events within missions, based on the hour, minute, and second of occurrence. Useful for detecting peaks in activity or potential data collisions.
  - columns:
    - name: mission
      - description: "Name or identifier of the mission."
    - name: event\_hour
      - description: "Hour (0-23) when the event occurred."
    - name: event\_minute
      - description: "Minute (0-59) when the event occurred."
    - name: event\_second
      - description: "Second (0-59) when the event occurred."
    - name: simultaneous\_events
      - description: "Number of events that occurred at the exact same time and location within a mission"
- name: fct\_week\_tendencies
  - description: >  
Fact table that aggregates the total number of events per week of the year. Useful for analyzing weekly trends, identifying seasonal patterns, and understanding long-term activity.
  - columns:
    - name: year\_week
      - description: "Year and week number (e.g., 2023-01)"

```
description: "ISO week number of the year (1-53) in which the events  
- name: total_events  
description: "Total number of events recorded during the specified v
```

- models/reports/reports\_schema.yaml :

version: 2

models:

- name: fct\_day\_events  
description: >  
Fact table that summarizes the number of unknown status events per mission and device. Used to monitor device connectivity or data quality.  
columns:
  - name: mission  
description: "Name or identifier of the mission associated with the events."
  - name: device  
description: "Type of device for which the unknown events were recorded."
  - name: number\_unknown  
description: "Total number of events with 'UNKNOWN' status for the given mission and device."
- name: fct\_disconnections  
description: >  
Fact table that tracks the number of disconnection events, identified by mission and device, for each mission and device. Useful for evaluating system reliability and detecting communication failures.  
columns:
  - name: mission  
description: "Identifier or name of the mission associated with the disconnections."
  - name: device  
description: "Type or category of the device that experienced disconnections."
  - name: number\_unknown  
description: "Count of events where the device reported an 'UNKNOWN' status due to disconnections."
- name: fct\_event\_days  
description: >  
Fact table that aggregates the total number of events per day of the week. Useful for identifying patterns and trends in event frequency based on the day of the week.  
columns:
  - name: event\_year  
description: "Year in which the events occurred."
  - name: day\_name  
description: "Name of the day of the week (e.g., MONDAY, TUESDAY)."
  - name: total\_events  
description: "Total number of events recorded on that day of the week for the given year."
- name: fct\_events  
description: >  
Fact table that summarizes the number of events by mission, device type, and operational status. Useful for analyzing event volume and device performance.  
columns:
  - name: mission  
description: "Identifier or name of the mission associated with the events."
  - name: device  
description: "Type or category of the device involved in the event."
  - name: status  
description: "Operational status reported for the device during the event."
  - name: number\_events  
description: "Total number of events for the given combination of mission, device, and status."

- name: fct\_mission\_analysis  
description: >  
Fact table that provides a daily operational breakdown of mission performance. It includes the count of events by status category and the total number of events per mission and date. This model helps identify operational trends, across columns:
  - name: format\_event\_date  
description: "Formatted string representing the date and time of the event"
  - name: mission  
description: "Identifier or name of the mission involved in the event"
  - name: status  
description: "Reported status of the device at the time of the event"
  - name: total\_faulty  
description: "Number of events with a 'FAULTY' status for the given mission"
  - name: total\_killed  
description: "Number of events where the device was reported as 'KILLED' for the mission"
  - name: total\_unknown  
description: "Number of events with 'UNKNOWN' status, indicating potential issues"
  - name: total\_warning  
description: "Number of events flagged as 'WARNING' for the mission"
  - name: total\_excellent  
description: "Number of events with an 'EXCELLENT' status indicating optimal performance"
  - name: total\_events\_mission  
description: "Total number of events recorded for the mission on the given date"
- name: fct\_missions  
description: >  
Fact table that summarizes the number of inoperable devices per mission. Inoperable devices include those with statuses such as 'FAULTY', 'KILLED', or 'UNKNOWN'. This model supports reliability and operational health analysis across columns:
  - name: mission  
description: "Name or code of the mission associated with the device"
  - name: inoperable\_devices  
description: "Total count of devices with inoperable status (FAULTY, KILLED, UNKNOWN) for the mission"
- name: fct\_operational\_duration  
description: >  
Fact table that summarizes the operational duration of each mission. It includes the first and last event timestamps, number of active days, total events, and average events per day. Useful for evaluating mission efficiency across columns:
  - name: mission  
description: "Name or identifier of the mission."
  - name: first\_event  
description: "Timestamp of the first recorded event for the mission"
  - name: last\_event  
description: "Timestamp of the last recorded event for the mission."
  - name: active\_days  
description: "Number of days between the first and last event (inclusive)"
  - name: total\_events  
description: "Total number of events recorded for the mission."

- name: avg\_events\_per\_day
  - description: "Average number of events per active day for the mission"
- name: fct\_percentages
  - description: >  
Fact table that calculates the percentage of events by mission and device in relation to the total number of events. Useful for understanding the distribution of activity across missions and device types.
  - columns:
    - name: mission
      - description: "Name or identifier of the mission."
    - name: device
      - description: "Type or category of the device."
    - name: percentage
      - description: "Percentage of total events represented by the given mission and device combination."
- name: fct\_ranks
  - description: >  
Fact table that ranks missions and devices based on the total number of events. Useful for identifying the most active mission-device combinations and their relative performance.
  - columns:
    - name: mission
      - description: "Name or identifier of the mission."
    - name: device
      - description: "Type or category of the device."
    - name: total
      - description: "Total number of events for the mission and device combination."
    - name: rank
      - description: "Dense rank of the mission-device combination based on the total number of events."
- name: fct\_simultaneous\_events
  - description: >  
Fact table that identifies simultaneous events within missions, based on the hour, minute, and second of occurrence. Useful for detecting peaks in activity or potential data collisions.
  - columns:
    - name: mission
      - description: "Name or identifier of the mission."
    - name: event\_hour
      - description: "Hour (0-23) when the event occurred."
    - name: event\_minute
      - description: "Minute (0-59) when the event occurred."
    - name: event\_second
      - description: "Second (0-59) when the event occurred."
    - name: simultaneous\_events
      - description: "Number of events that occurred at the exact same time and location within a mission."
- name: fct\_week\_tendencies
  - description: >  
Fact table that aggregates the total number of events per week of the year. Useful for analyzing weekly trends, identifying seasonal patterns, and understanding long-term activity.
  - columns:
    - name: year\_week
      - description: "Year and week number (e.g., 2023-01) for aggregation."

```

description: "ISO week number of the year (1-53) in which the event:
- name: total_events
description: "Total number of events recorded during the specified v

```

### 💡 TIP

En este taller se utilizan diferentes nombres de archivos `schema` (por ejemplo: `macros_schema.yml`, `staging_schema.yml`, etc.) para mostrar que dbt **mapea y relaciona automáticamente estos archivos**, sin necesidad de que tengan un nombre estándar.

Lo único importante es que:

- El archivo tenga extensión `.yml`
- El nombre incluya el sufijo `_schema` (esto es buena práctica, no obligatorio)

Además, dentro de estos archivos puedes incluir pruebas ( `tests` ) asociadas a modelos o columnas, y dbt se encargará de **ejecutarlas automáticamente** al correr:

```
dbt test
```

3. Luego de crear los archivos de documentación , es necesario ejecutar los siguientes comandos para que dbt genere y disponibilice la información:

```
dbt docs generate
dbt docs serve
```

## 14. PERSONALIZAR DOCS CON ARCHIVOS MARCDOWN

Aunque dbt permite escribir documentación directamente en archivos `.yml`, esta opción puede resultar limitada cuando se necesita incluir explicaciones más detalladas, listas, ejemplos o formato enriquecido.

Al usar archivos Markdown:

- Mejoras la organización, separando la lógica del modelo de la documentación.
- Ganas expresividad, ya que puedes utilizar títulos, tablas, listas y formato enriquecido que no es posible en archivos `.yml`.



- Puedes reutilizar contenido, ya que un mismo archivo Markdown puede ser referenciado en varios modelos o columnas.
- Escalas mejor la documentación, haciéndola más fácil de mantener y actualizar cuando crece en complejidad.

## **Cómo documentar en dbt usando archivos Markdown:**

1. Crea una carpeta llamada `docs` en la raíz de tu proyecto dbt.
2. Para que dbt pueda reconocer los archivos Markdown, agrega esta línea en tu archivo `dbt_project.yml`:

```
docs-paths: ["docs"]
```

3. Puedes reemplazar la página principal de la documentación creando un archivo Markdown dentro de una macro. Ejemplo:

- `/docs/homepage.md` :

```
{% docs __overview__ %}
! [NASA Logo](https://upload.wikimedia.org/wikipedia/commons/e/e5/NASA_logo.)

# Welcome to the Space Missions Monitoring Project - NASA

The National Aeronautics and Space Administration (NASA) is the United States'
leading agency for civilian space exploration, scientific research, and aeronautics
research. Today, NASA is entering a new era of exploration, innovation, and global
inspiration.

- OrbitOne (ORBONE): Modernizing the satellite fleet to enhance performance.
- ColonyMoon (CLNM): Establishing a permanent lunar colony.
- VacMars (TMRS): Beginning manned tourist travel to Mars.
- GalaxyOne (GALXONE): A long-term project to explore galaxies beyond our
solar system.

These groundbreaking initiatives rely on cutting-edge technologies and have
revolutionized our understanding of the universe.

Its core objective is to detect anomalies early and enable proactive measures
to ensure mission success.

## Your Role

Due to the importance of this effort, you have been selected as the Chief
Travel to Cape Canaveral and lead the initial simulation of the monitoring
process.

This is a unique opportunity to contribute to one of the most significant
technological endeavors in history.

---

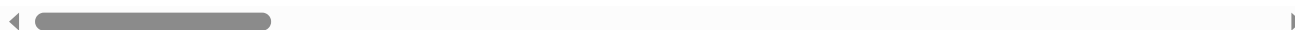
### What to Expect

Throughout this simulation, you will:

- Handle real-time telemetry and operational logs.
- Detect and document potential anomalies.
- Validate data pipelines and ensure system robustness.
- Collaborate with engineers, analysts, and mission control.

This project will test not only your technical skills but your ability to lead
a team through complex challenges.

{% enddocs %}
```



4. También puedes crear documentación detallada por campo. Por ejemplo, para mejorar la descripción del campo `device_type`, haz lo siguiente:

- `docs/devices/device_type.md` : Crear carpeta y archivo asociado a la tabla de eventos. Agregar el siguiente contenido:

```
{% docs device_status %}
# Field Documentation: `device_status`
```

## ## 1. Possible Status Values

The `device\_status` field tracks the current operational condition of a device.

- **faulty**: The device is experiencing critical issues and requires immediate attention.
- **killed**: The device has been deliberately shut down or terminated due to a critical error.
- **unknown**: The current status of the device cannot be determined due to a communication loss.
- **warning**: The device is operational but has registered conditions that may lead to failure.
- **good**: The device is functioning within expected parameters.
- **excellent**: The device is performing optimally with no anomalies detected.

## ## 2. Why Should This Be Monitored?

Monitoring the `device\_status` field is essential for maintaining the integrity and safety of the system.

- Prevention of mission failure
- Proactive maintenance and support
- Better allocation of resources and personnel
- Real-time response to emergencies

## ## 3. How It Supports Space Travel

In space missions, even minor malfunctions can have major consequences. By monitoring device status, we can:

- Engineers can isolate and mitigate faults before escalation.
- Astronauts are informed in real time about equipment conditions.
- Data scientists can build predictive models to foresee failures.

This contributes to mission longevity, astronaut safety, and optimal use of resources.

## ## 4. Scientific Context and Illustrations

### ### Speed of Light Formula

In understanding signal transmission delays and timing diagnostics from remote locations, the speed of light is a fundamental constant.

$$c = 299,792,458 \text{ m/s}$$

Where:

- ( $c$ ) is the speed of light in a vacuum

This helps in calculating communication delays:

$$t = \frac{d}{c}$$

Where:

- $t$ : time delay
- $d$ : distance from the device
- $c$ : speed of light

### Device Signal Reliability Model

A simplified reliability function over time:

$$R(t) = e^{-\lambda t}$$

Where:

- $R(t)$ : probability the device is still functioning at time  $t$
- $\lambda$ : failure rate constant

### Example Devices

Here are a few types of devices that report `device\_status`:

- Satellite transponders
- Onboard telemetry modules
- Environmental sensors
- Navigation and guidance units

![[Satellite Antenna]](<https://www.frederickscompany.com/wp-content/uploads/2019/01/Satellite-Antenna-Image.jpg>)  
 ![[Sensor Module]]([https://www.esa.int/var/esa/storage/images/esa\\_multimedia/images/2019/01/Sensor\\_Module\\_Image.jpg](https://www.esa.int/var/esa/storage/images/esa_multimedia/images/2019/01/Sensor_Module_Image.jpg))

## 5. Why This Field Matters

The `device\_status` field is a cornerstone of system diagnostics in space missions.

- Safer missions
- Better system designs
- Insightful data-driven decisions

In short, it's a small field with a huge impact.

5. Luego, en el archivo `models/sources/source_schema.yml` dentro de la carpeta `models`, reemplaza la descripción del campo con una referencia al archivo Markdown:

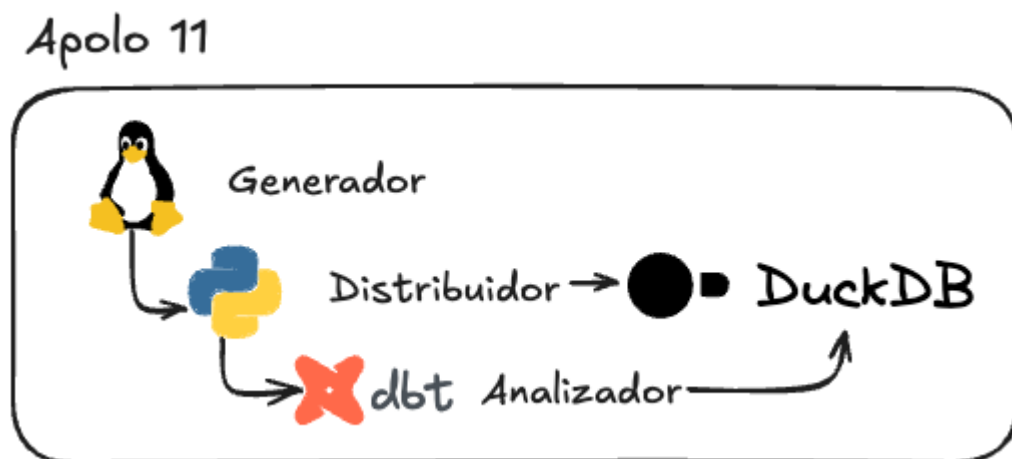
- name: device\_status
- description: "{ doc('device\_status') }"

## 15. INTEGRAR CON EL GENERADOR

Ahora que el flujo de trabajo con dbt está completamente funcional, incluyendo la instalación de dependencias, carga de semillas, ejecución de modelos y validación mediante pruebas, está listo para integrarse a la arquitectura analítica del proyecto Apolo 11.

Lo ideal es que completen la integración dentro de su flujo existente, asegurándose de que cada componente se conecte de manera coherente con los procesos ya implementados. Además, es importante que se tomen el tiempo para reflexionar sobre el propósito de lo que están construyendo: una solución real no es un único archivo, sino un conjunto de partes que trabajan juntas para resolver un problema.

Les dejo un diagrama de referencia para que puedan integrar y ajustar el trabajo existente de forma estructurada y eficiente.



## 16. COMANDOS UTILES DE DBT

Esta sección resume los comandos más comunes y útiles de dbt para desarrollar, ejecutar, probar y documentar tus modelos de datos en el proyecto Apolo 11.


### Configuración del proyecto

```

dbt init mi_proyecto      # Inicializa un nuevo proyecto dbt
dbt deps                  # Instala las dependencias del proyecto desde pa
  
```


### Ejecución de modelos

```
dbt run                                # Ejecuta todos los modelos
dbt run --select nombre_modelo         # Ejecuta un modelo específico
dbt run --exclude nombre_modelo       # Ejecuta todos los modelos excepto el indica
```




## Carga de seeds

```
dbt seed                              # Carga archivos CSV desde la carpeta seeds al d
```




## Pruebas

```
dbt test                              # Ejecuta todas las pruebas definidas en archivo
dbt test --select nombre_modelo       # Ejecuta pruebas para un modelo específico
```




## Compilación y construcción

```
dbt compile                           # Compila el código SQL al directorio target
dbt build                             # Ejecuta modelos, pruebas, snapshots y seeds (d
```



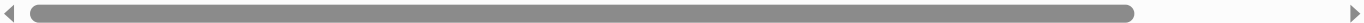
## Documentación

```
dbt docs generate                     # Genera la documentación en formato web
dbt docs serve                        # Lanza un servidor local para explorar la docum
dbt docs serve --port 8080           # Usa un puerto personalizado si el 8000 está en
```



## Debug y validación

```
dbt debug                            # Verifica la conectividad y configuración del e
dbt ls                               # Lista todos los nodos del proyecto (modelos, p
```



## Actualización o cambio de versión

```
pip install --upgrade dbt-core        # Actualiza dbt a la última versión di
pip install dbt-core==x.y.z           # Instala una versión específica de db
pip install --upgrade dbt-postgres    # Aplica también para adaptadores (sno
```



Estos comandos pueden ayudarte a mantener tu flujo de trabajo ordenado, reproducible y confiable.