DBT

Data Transformation Service

JERIN SAM

Table of Contents

[DBT 2](#_Toc185597691)

[What DBT Is 3](#_Toc185597692)

[What DBT Is Not 3](#_Toc185597693)

[How DBT Works 3](#_Toc185597694)

[DBT Helps To 4](#_Toc185597695)

[Setup DBT and Configure DBT Project 4](#_Toc185597696)

[Setup DBT 4](#_Toc185597697)

[Configure and create DBT project 5](#_Toc185597698)

[DBT Components 6](#_Toc185597699)

[Models 6](#_Toc185597700)

[Implementation 6](#_Toc185597701)

[Materialization 7](#_Toc185597702)

[Types of Materialization: 8](#_Toc185597703)

[Implementation 9](#_Toc185597704)

[Seeds 13](#_Toc185597705)

[Implementation 14](#_Toc185597706)

[Sources 15](#_Toc185597707)

[Source Freshness 16](#_Toc185597708)

[Implementation 16](#_Toc185597709)

[Sources vs Seeds 18](#_Toc185597710)

[Snapshots 18](#_Toc185597711)

[Implementation 19](#_Toc185597712)

[Tests 20](#_Toc185597713)

[Implementation 21](#_Toc185597714)

[Custom Tests – Using Macros & Packages 23](#_Toc185597715)

[Implementation 23](#_Toc185597716)

[Documentation 28](#_Toc185597717)

[Implementation 29](#_Toc185597718)

[Analyses 35](#_Toc185597719)

[Implementation 35](#_Toc185597720)

[Hooks 36](#_Toc185597721)

[Implementation 37](#_Toc185597722)

[Exposures 38](#_Toc185597723)

[Implementation 39](#_Toc185597724)

[Tests using *dbt-expectations* package 40](#_Toc185597725)

[Great Expectations 40](#_Toc185597726)

[dbt-expectations package Installation 40](#_Toc185597727)

[dbt-expectations tests implementation 42](#_Toc185597728)

[DBT Logging 43](#_Toc185597729)

[Implementation 44](#_Toc185597730)

[Variables 47](#_Toc185597731)

[Appendix 49](#_Toc185597732)

[Important Links 49](#_Toc185597733)

# DBT

DBT is an open-source data transformation tool designed for analytics engineering. It is widely used for ***transforming***, ***testing***, and ***documenting data*** in data warehouses. It is the “T” in ETL, it is used for transformation using SQL based syntax.

In DBT, Transformation is ***version controlled***.

A screenshot of a computer

Description automatically generated

DBT is not a processing engine. Instead, it is a data transformation tool that ***relies on the processing power of your data warehouse or query engine***.

## What DBT Is

* **SQL-Based Transformation Tool:** DBT uses SQL to define data transformations. You write SQL queries (models), and DBT compiles them into SQL scripts that your data warehouse executes.
* **Orchestration of SQL Jobs:** DBT organizes, manages, and orchestrates SQL transformations, but it does not directly handle the processing of data.
* **Workflow Management:** DBT handles dependencies between data models, allowing you to build complex workflows. It ensures that transformations run in the correct order based on these dependencies.
* **Code Management:** DBT provides a structured framework for managing SQL code, applying data tests, documenting transformations, and version-controlling data pipelines.

## What DBT Is Not

* **DBT is** **not a Database or Processing Engine**: DBT does not store data, nor does it have its own query execution engine. It depends on a data warehouse, such as:
  + BigQuery
  + Snowflake
  + Redshift
  + PostgreSQL
  + Databricks
  + Azure Synapse
* **Not an ETL Tool:** DBT focuses only on the Transformation (T) aspect of the ETL/ELT process. It does not handle Extraction (E) or Loading (L) of data from source systems into a data warehouse.

## How DBT Works

* **SQL Compilation:** DBT takes your SQL files (models) and Jinja templates, compiles them into raw SQL statements, and submits them to your data warehouse for execution.
* **Data Warehouse Execution:** The actual data processing happens in the data warehouse. DBT does not perform the heavy lifting; it delegates the query execution to the warehouse.
* **Results Storage:** Once the data warehouse completes the query, the results are stored in the tables or views specified by DBT.

## DBT Helps To

1. Modelling changes are easy to follow and revert by using version control feature of DBT.
2. View and create dependencies between models.
3. Data quality checks.
4. Error reporting.
5. Incremental load of fact tables.
6. Track history of dimension tables.
7. Easy-to-access documentation.

# Setup DBT and Configure DBT Project

## Setup DBT

***Refer:***

* ***Dockerfile -*** */dbt-learn-hands-on/install\_and\_config/2-dbt-install/Dockerfile*
* ***Python Package Requirement*** - */dbt-learn-hands-on/install\_and\_config/2-dbt-install/* *requirements.txt*
* ***Installation Details*** - */dbt-learn-hands-on/install\_and\_config/2-dbt-install/* *script-dockerfile-execute.md*
* ***SQL Server ODBC Installation -*** */dbt-learn-hands-on/install\_and\_config/3-install-sql-server-odbc-docker-container/install-odbc-driver-container.md*
* ***Host Machine VS Code Connection with Container’s Python Interpreter –***

*/dbt-learn-hands-on/install\_and\_config/4-connect-vscode-docker-container/config-vscode-connect-python.md*

In this tutorial, DBT is installed on docker container. Following are the steps –

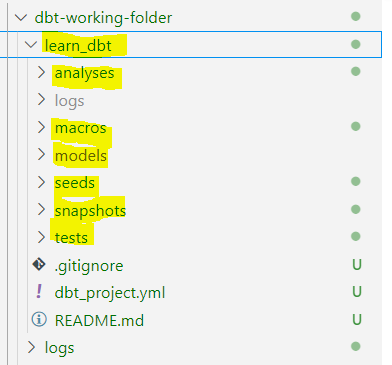
1. Create a *Dockerfile*, which contains Python 3.11.10 slim bookwork image.
   * Types of Docker Images
     + Bookworm - refers to the codename for the latest stable release of Debian.
     + Slim - indicates a minimal version of a Debian distribution with only the essential packages installed
     + Bullseye - is the codename for the previous stable Debian release
     + Alpine - is the “Dockerized” version of Alpine Linux
2. Install *dbt-core* and *dbt-sqlserver* *adapter* via *Dockerfile*.
3. Create working directory in host machine “***/dbt-learn-hands-on/main/dbt-working-folder***” and mount it with Docker container’s working directory using *Dockerfile*, so that any changes done through host machine can be synced in container’s working directory and vice versa.
4. Manually install SQL Server ODBC driver in the container.
5. After the ODBC setup, connect host machine’s VS Code to python interpreter present in Docker container.

## Configure and create DBT project

***Refer: /dbt-learn-hands-on/main/1-configure-dbt-project.md***

Following steps are taken to configure and create DBT folder structure in the container –

1. Create a Python Virtual Environment in the Docker Container and Activate It.
2. Create DBT Config/Profile Folder – Profile folder ***.dbt*** should be created at ***root*** directory. While initializing a DBT project using ***dbt init <project\_name>***a ***profiles.yml*** file will be created at this location to hold connection details of the database to be used by the dbt for the transformations. In this case its SQL Server.
3. Initialize DBT Project and Configure Database using ***dbt init <project\_name>*** - after executing the initialization script, a dbt folder structure will be created.



1. Test DBT Database Connection using ***dbt debug*** –
   * This command will be executed in the dbt project folder to test the connection with database used by DBT for the transformation, in this case its SQL Server.
   * This command uses ***profiles.yml*** file created in the above step.
   * ***Note -*** Generally, error occurs dung the testing, make sure that SQL Server connection details in the ***profiles.yml*** file should be inline with the reference profile file exist at ***“/main/dbt-working-folder/dbt-sql-server-profile.yml”,*** if required copy the same content in ***profiles.yml*** file.
2. Clean DBT Project Folder – This is required to delete examples exist in the dbt project folder.

# DBT Components

## Models

Models are the basic building blocks to implement the business logics. DBT models are SQL files that contain SQL queries used to define transformations i.e. In DBT, Models folder contains SQL files.

Models are materialized as views, tables etc. this means, it corresponds to a view or table in the data warehouse.

***Materialization*** *refers to how a model's query results are stored in the data warehouse. Models can be* ***materialized as views, tables, or other options*** *depending on the configuration and requirements.*

Models can refer other models and can also use templates and macros.

### Implementation

#### Dataset

Airbnb data is used in this tutorial. Refer: ***/dbt-learn-hands-on/main/2-get-data-for-tutorial.md*** for the details on the dataset.

***Dataset Reference:*** */dbt-learn-hands-on/main/2-get-data-for-tutorial.md*

#### Create Models

***DBT Model Reference:*** */dbt-learn-hands-on/main/dbt-working-folder/learn\_dbt/models/src*

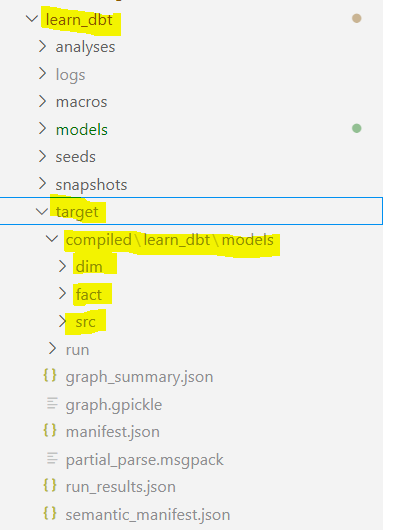
1. Create 3 models in ***/models/src*** folder of the dbt project “***learn\_dbt***”
   1. ***src\_listing.sql*** – This is used to read data from the table ***raw.listings*** and materialized as view ***dbo.listings***
   2. ***src\_reviews.sql*** – This is used to read data from the table ***raw.reviews*** and materialized as view ***dbo.reviews***
   3. ***src\_calendar.sql*** – This is used to read data from the table ***raw.calendar*** and materialized as view ***dbo.calendar***
2. Execute all the created models using ***dbt run.*** This command will create all the necessary DB objects in the database.

#### Debugging

During model execution process, DBT compiles the model sql files and save it at ***/target/compiled*** folder of dbt project folder ***“learn\_dbt”.***

While executing ***dbt run,*** if error occurs then go to ***/target/compiled*** folder in dbt project ***“learn\_dbt”*** and check the compiled sql query of the model, then copy it and execute that query in the database to understand the error.

***dbt compile*** command can also be used to generate the compiled query which can be accessible from ***/target/compiled*** folder



## Materialization

**Materialization** is the process of defining how ***dbt*** should build and store a model (a SQL query) in the database.

Materializations control whether the model's output is stored as a table, a view, or something else.

### Types of Materialization:

#### Table –

* ***What it does:***
  + Executes the query defined in the model and stores the results in a physical table in the database.
* ***When to use:***
  + When the data doesn’t change often.
  + When you need quick access to pre-computed results for performance reasons.
  + Ideal for intermediate or final datasets used in reporting.

1. View –

* ***What it does:***
  + Creates a database view that runs the query in real-time whenever the view is accessed.
* ***When to use:***
  + When the underlying data changes frequently and real-time results are needed.
  + When storage space is a concern (as views don’t store data).
  + **Drawback**: Slower query performance compared to tables since the query is executed every time the view is accessed.

1. Incremental –

* ***What it does:***
  + Adds only new or updated data to an existing table instead of rebuilding the entire dataset.
* ***When to use:***
  + When working with large datasets.
  + When data ingestion or refresh should be efficient and scalable.
* ***Key setup:***
  + An ***is\_incremental()*** macro to define incremental logic.
  + A unique key or timestamp to identify new data.

1. Ephemeral–

* ***What it does:***
  + Executes the query during the build process without materializing it in the database.
* ***When to use:***
  + When the model is only used as an intermediate step in downstream queries.
  + Useful for temporary transformations or intermediate logic.
  + ***Drawback***: Cannot be queried directly in the database.

### Implementation

During the implementation, additional models are created, in these models jinja templates are used to –

* Refer one models in another model.
* Implementation of Materialization.

***Steps:***

1. Create 2 models in ***/models/dim*** folder of the dbt project “***learn\_dbt***”
   1. ***dim\_listing.sql*** – Create dimension table for listings dataset by referencing source model at ***“/models/src***” and use sub-folder level materialization, which will be defined at ***dbt\_project.yml*** file (implementation explanation is given below).
   2. ***dim\_host.sql*** – Create dimension table for hosts dataset by referencing source model at ***“/models/src***” and use sub-folder level materialization, which will be defined at ***dbt\_project.yml*** file (implementation explanation is given below).
   3. ***dim\_listing\_and\_host.sql*** – Create a combined dimension table by referencing dimension models created in above steps and use sub-folder level materialization, which will be defined at ***dbt\_project.yml*** file (implementation explanation is given below).
2. Create 1 models in ***/models/fact*** folder of the dbt project “***learn\_dbt***”
   1. ***fact\_reviews.sql*** – Incremental Load Fact table for reviews, use model level materialization.
3. Update 3 models in ***/models/src*** folder of the dbt project “***learn\_dbt***”
   1. Update all the 3 source models to add model level materialization using jinja templat
4. Use ***dbt run*** to execute the models.
5. Use ***dbt compile*** to compile the models.

#### Model Reference in another Model

During model creation***, jinja templates*** can be used to refer other models as well.

***Syntax: {{ref(‘filename’)}}***

***{{}} this is the jinja template, it is used to tell DBT to dynamically change the content during the compilation.***

A screen shot of a computer

Description automatically generated

#### Materialization at global project level and model sub-folder level–

Materialization implementation can be done at global level, at model sub-folder level (where models are stored) and at individual model sql file level.

*In DBT project,* ***dbt\_project.yml*** *file is used to set the default materialization at global project level and at model’s sub-folder level, this folder exists in the DBT project folder –*

A screenshot of a computer

Description automatically generated

*In* ***dbt\_project.yml****, materialization is added as a config under models -*

A screenshot of a computer screen

Description automatically generated

#### Materialization at model level

In the model sql files, Materialization at global project level and sub-folder level can be overridden by using jinja template in the model file itself.

* + ***Reference- Materialization-Incremental****: Using Jinja template, config method is used to override default materialization -*

A screenshot of a computer

Description automatically generated

* + ***Reference- Materialization-Table****: Using Jinja template, config method is used to override default materialization –*

A screenshot of a computer screen

Description automatically generated

#### Defining Incremental load logic

To implement incremental load, Materialization should be incremental and jinja template should be used along with ***is\_incremental() macro to define the incremental load logic***

A screenshot of a computer program

Description automatically generated

During the execution of ***dbt run***, data will be fully loaded to the database table, and now during the next execution onwards, only incremental data will be loaded to the database table.

If data needs to be fully loaded again then ***dbt run --full-refresh*** can be used.

## 

## Seeds

Seeds are CSV files that are part of your DBT project and can be used to load static or reference data into the database.

**Key Features of Seeds:**

* ***Static Data:*** Seeds are useful for loading small, static datasets, such as lookup tables, mappings, or default configurations.
* ***Version Controlled:*** Since seeds are stored as CSV files in dbt project directory, they can be managed using version control systems like Git.
* ***Materialized as Tables:*** Executing command ***dbt seed***, the CSV files are uploaded to the database and materialized as tables.

**How to Use Seeds:**

Uploading CSV data to ***/seed/*** folder of ***dbt project folder "learn\_dbt"*** and then executing the command ***“dbt seed”*** will push the data to the database.

### Implementation

In this implementation, a onetime load data will be uploaded to the DB.

***Steps:***

1. A csv file will be added to the **/seed/** folder of dbt project folder **"learn\_dbt".**

***Refer*** ***data at location***: /main/dbt-working-folder/learn\_dbt/seeds/seed\_full\_moon\_dates.csv

A screenshot of a computer

Description automatically generated

1. Execute ***dbt seed*** to push the data to the database, which will be materialized as a table in the database.

A screenshot of a computer

Description automatically generated

1. Create a model in ***/models/*** folder in dbt project ***“learn\_dbt”***, where above created seed table will be used.

Jinja template is used to refer this seed table, the method to use it in a model is similar to references used in normal models.

A screenshot of a computer

Description automatically generated

## Sources

Sources represent tables or views that exist outside of dbt's control, typically raw data in the data warehouse that is loaded by an external process like ETL.

**Key Features of Sources:**

* ***External Data:*** Used to document and query raw or external datasets.
* ***Data Lineage:*** Helps track data lineage from raw sources to transformed outputs.
* ***Tests and Documentation:*** Tests can be applied (e.g., uniqueness, not null) and generate documentation for sources.

**How to Use Sources:**

* Add source definitions in a .yml file (***sources.yml***) inside the ***/models/***directory ***dbt project folder "learn\_dbt"***
* Use the ***source*** function in dbt models to reference sources.

### Source Freshness

***Source Freshness*** refers to the ability to monitor and ensure the timeliness of raw data sources in the database/ data warehouse.

This feature allows to check if your data sources (e.g., external tables or views) are up-to-date based on specified freshness criteria, such as how recently the data was updated.

**How Source Freshness Works:**

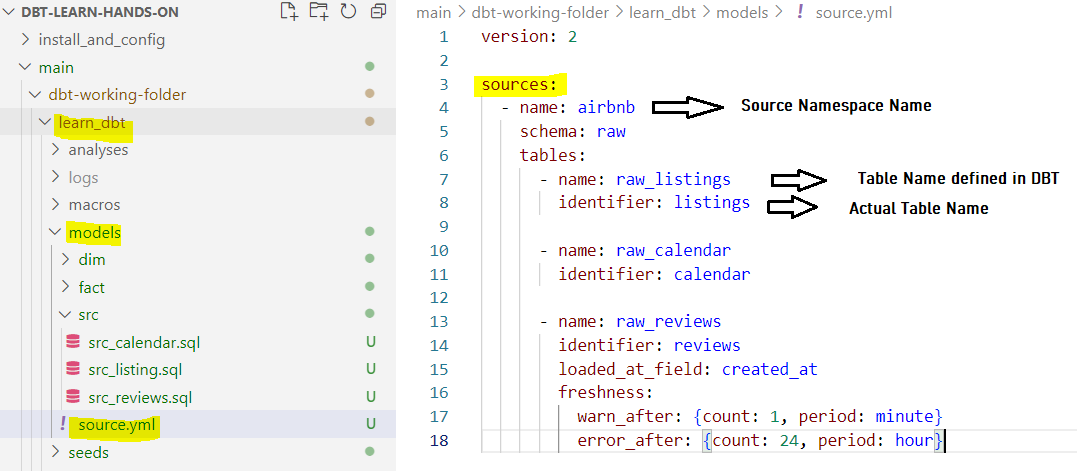
* **Define Freshness Criteria:** Specify freshness rules for your sources in the sources section of a .yml file.
  + The rules include:
    - **loaded\_at\_field:** The column that contains the timestamp of when the data was last updated.
    - **Freshness thresholds:** 
      * **warn\_after:** Time after which a warning will be issued if the data is not fresh.
      * **error\_after:** Time after which an error will be issued if the data is not fresh.
* **Run Source Freshness Checks:** Use ***dbt source freshness*** command to check whether the data meets the defined criteria.
* **View Results:** The results show whether each source is fresh, stale, or failing based on the thresholds you defined.

### Implementation

In this implementation, Sources yaml file will be created in ***/models/*** folder and then the models in /models/src folder will be updated to add reference from Source. yaml file.

***Steps***:

1. Create ***sources.yml*** file in ***/models/*** folder and add all the raw data tables in it.



1. Update models in ***/models/src*** folder to use sources mentioned in ***sources.yml*** file.

A screenshot of a computer

Description automatically generated

1. Add source freshness criteria in reviews dataset.

A screenshot of a computer program

Description automatically generated

1. Test the updated models by executing ***dbt run*** or ***dbt compile***
2. Check Source Freshness by executing ***dbt source freshness***

***Received below Error since data is stale beyond 24 hours as configured in sources.yml file***

A screenshot of a computer program

Description automatically generated

## Sources vs Seeds

|  |  |  |
| --- | --- | --- |
| Feature | Seeds | Sources |
| Definition | Local CSV files in the dbt project. | External tables/views in the database. |
| Purpose | Load static/reference data. | Access raw/external data. |
| Reference Method | {{ ref('seed\_name') }} | {{ source('source\_name', 'table\_name') }} |
| Location | Stored in /seed/ folder in the project. | Already present in the database. |
| Materialization | Always materialized as tables. | Materialized externally. |

## Snapshots

***Snapshot*** is a feature that allows to track changes in data over time by capturing and storing the historical state of a table or query results.

Snapshots are particularly useful for slowly changing dimensions (SCDs) or for maintaining a history of changes to records in your data warehouse.

**How Snapshots Work:**

1. ***Define a Snapshot***:
   1. Create a snapshot configuration in ***/snapshot/*** folder of ***dbt project “learn\_dbt”***, specifying how changes should be detected in data. Jinja template is used to add the configuration.
   2. Snapshots track changes by comparing the current state of a record with its previous state.
2. ***Materialize Changes:***
   1. Executing command ***dbt snapshot***, dbt creates or updates a table in the database that stores the historical records.
3. ***Store Historical Data:***
   1. Snapshots append new rows when changes are detected, allowing to maintain a historical log of data changes.

**Snapshot Strategies:**

1. ***Timestamp Strategy:***
   1. Uses a timestamp and unique identifier column (***e.g., customer\_id,*** ***updated\_at***) to detect changes.
   2. Best for tables that have a clear last-updated timestamp.
2. ***Check Strategy:***
   1. Compares specified columns (***check\_cols***) to detect changes.
   2. Useful for tables without a timestamp column.

### Implementation

In this implementation, SCD Type 2 will be implemented on raw listings dataset.

**Steps:**

1. Create a snapshot sql file for listing dataset (reference from sources.yml file) in ***/snapshot/*** folder of dbt project ***learn\_dbt***

A screenshot of a computer program

Description automatically generated

1. Execute command ***dbt snapshot*** to create a snapshot table, which will create additional columns to support SCD Type 2.
2. Update main raw listings table using below sql command and then re-execute ***dbt snapshot***

select minimum\_nights from [raw].[listings]

WHERE ID = '25840298'

UPDATE [raw].[listings]

SET minimum\_nights = 40, updated\_at = GETDATE()

WHERE ID = '25840298'

1. Check SCD Type 2 in snapshot table created by dbt.

A screenshot of a computer

Description automatically generated

## Tests

***Tests*** are used to ensure the quality and consistency of the data in the models, sources, or snapshots.

It helps to validate, data adheres to certain rules and constraints, ensuring that it is correct, complete, and in the expected format before it is used in downstream analyses or reporting.

**Types of Tests:**

* ***Singular Tests***: Singular tests are SQL queries stored in ***/tests/*** folder of dbt project ***“learn\_dbt”*** which are expected to return an empty result set.
* ***Generic Tests***: There are four built-in generic tests:
  + ***unique***: Checks that the values in a column are unique (i.e., no duplicates).
  + ***not\_null*** : Checks that a column does not contain any NULL values.
  + ***accepted\_values:*** Checks that the values in a column are within a set of predefined values.
  + ***relationships:*** Ensures that a key in one model exists as a in another model.

These Generic tests are defined in ***schema.yml*** file and will reside in ***/models/*** folder of dbt project ***“learn\_dbt” to test defined models.***

**NOTE:**

1. To test seeds, these Generic tests are defined in ***schema.yml*** file that reside in ***/seeds/*** folder of dbt project ***“learn\_dbt”***
2. To test sources, these Generic tests are defined in ***source.yml*** file present in ***/models/*** folder of dbt project ***“learn\_dbt”***

### Implementation

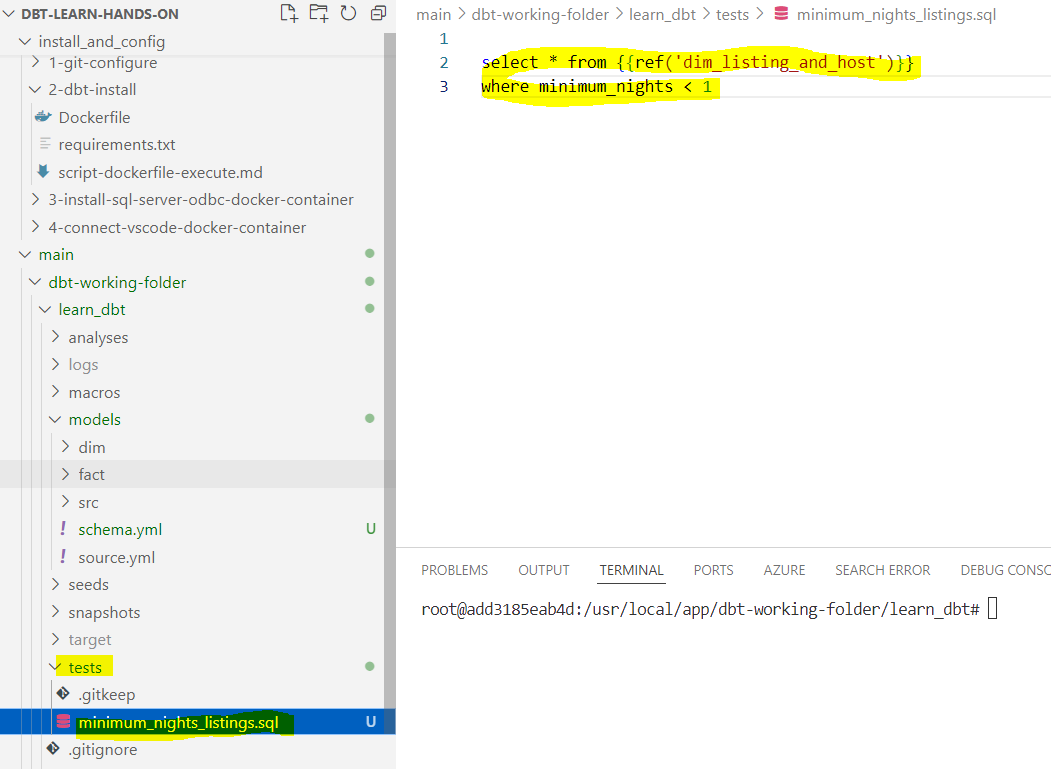
In this implementation, Tests will be defined for *“[dbo].[dim\_listing\_and\_host]”* dimension and following will be Test criteria –

1. Generic Test rules will be defined in ***schema.yml*** file in ***/models/*** folder of dbt project ***“learn\_dbt”.***
   1. Check not\_null and unique on column “*listing\_id*” of table *“[dbo].[dim\_listing\_and\_host]”*
   2. Check accepted\_values on column “*room\_type*” of table *“[dbo].[dim\_listing\_and\_host]”*
   3. Check relationships on column “*listing\_id*” of table *“[dbo].[dim\_listing\_and\_host]”*

A screenshot of a computer

Description automatically generated

1. Singular Test rules will be defined using SQL query and will be placed in ***/tests/*** folder of dbt project ***“learn\_dbt”*** which are expected to return an empty result set.
   1. Create a test to check if column *minimum\_nights* of table *“[dbo].[dim\_listing\_and\_host]”* is < 1



1. Execute ***dbt test*** to test the models by applying the defined generic and singular tests

A screenshot of a computer program

Description automatically generated

## Custom Tests – Using Macros & Packages

**Macro** is a reusable block of logic written in Jinja, a templating language. Macros allow to write functions that can be used throughout *dbt project*, enabling to avoid repetition and simplify complex logic.

* Macros are jinja templates created in the ***/macros/***folder
* There are many built-in macros in DBT
* Macros can be used in model definitions and tests
* A special macro, called ***test****,* can be used for implementing ***custom generic tests.*** *Details can be seen in Implementation section*
* ***dbt packages*** can be installed easily to get access to an additional set of macros and tests. *Details can be seen in Implementation section*

### Implementation

In this section, following implementations will be done –

1. Macro will be created which will be used to test the model columns for non-null values.
2. Test macro will be created to implement custom generic tests.

#### Macro Implementation Steps

1. Create a sql file in ***/macros/*** folder of dbt project ***“learn\_dbt”*** and create jinja macro template function ***“check\_not\_null\_all\_columns”***, which will take model as function parameter and add sql query to check nulls in columns.

A screenshot of a computer

Description automatically generated

1. Create a sql file in ***/test/*** folder of dbt project ***“learn\_dbt”*** and call the macro in it, this will be singular test to test the model ***“dim\_listings”***.

A screenshot of a computer

Description automatically generated

1. Test the macro by executing, ***dbt test --select <model\_name>*** *i.e dbt test –select dim\_listings*
2. In case of any error, refer ***/target/compiled/learn\_dbt/test*** folder and check the compiled sql query.

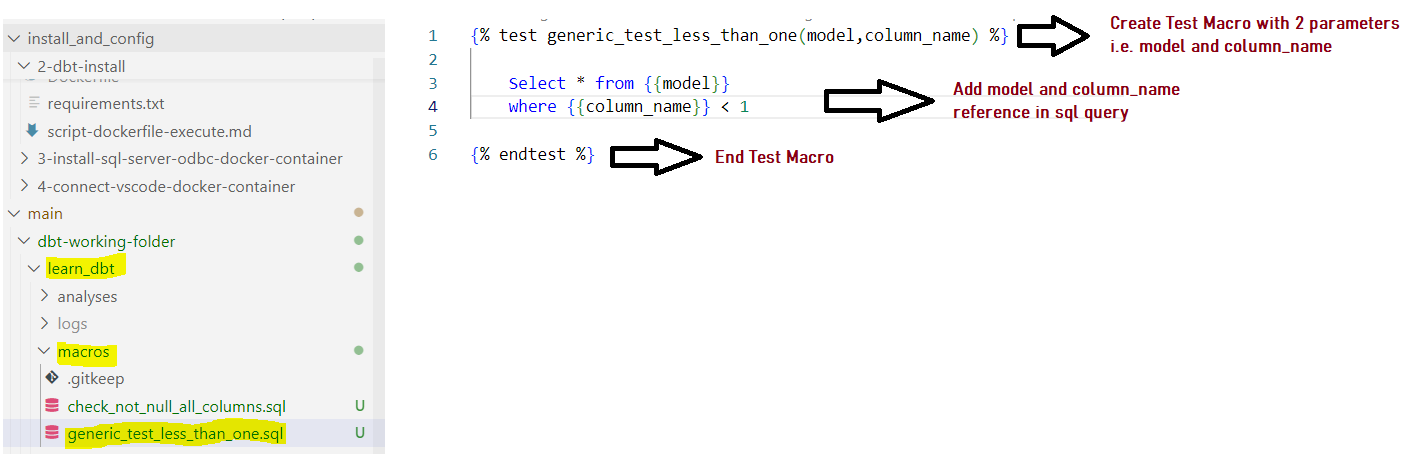
A screenshot of a computer

Description automatically generated

#### Test Macro Implementation Steps

1. Create a sql file in ***/macros/*** folder of dbt project ***“learn\_dbt”*** and create jinja ***test macro*** function ***“generic\_test\_less\_than\_one”***, which will take model and column name as function parameter and add sql query to check columns < 1.

***Note:*** Parameter name should always be ***model*** and ***column\_name***



1. Open ***/model/schema.yml*** file and add above created generic function on model ***“dim\_listings”***  and column ***“minimum\_nights”***

A screenshot of a computer

Description automatically generated

1. Test the macro by executing, ***dbt test --select <model\_name>*** *i.e dbt test –select dim\_listings*

#### DBT Package Implementation Steps

1. DBT package ***“dbt\_utils”*** will be installed to access the macros and test present in this package.
2. Access below link and copy the package installation steps to ***packages.yml*** file in dbt project ***/learn\_dbt/*** folder.

<https://hub.getdbt.com/dbt-labs/dbt_utils/latest/>

A screenshot of a computer

Description automatically generated

1. Execute ***dbt deps*** to install the package.

A screen shot of a computer code

Description automatically generated

1. Use ***dbt\_utils*** package to add surrogate key called ***“review\_key”*** in ***“facr\_reviews”*** model present in ***/models/fact/*** folder. It will be added as jinja template.

A screenshot of a computer

Description automatically generated

Below reference shows the addition of surrogate key called ***“review\_key”.***

A screenshot of a computer

Description automatically generated

## Documentation

***Documentation*** is the process of documenting metadata about models, sources, columns, and other entities in dbt project.

dbt makes it easy to document data project alongside the code, ensuring that stakeholders can understand the context, purpose, and lineage of the data.

This is the process to create the ***data catalog*** of dbt project.

**Key Components of Documentation:**

1. ***Schema YAML Files:***
   * ***DBT Documentation*** feature accesses the ***schema.yml*** file present in dbt project ***“learn\_dbt”*** to create the documentation.
   * ***schema.yml*** file is used to define metadata for models, sources, seeds, snapshots, and columns.
   * Adding descriptions, tests, and tags to models and their components like column in ***schema.yml*** file, will be accessed by ***DBT Documentation*** to create the documentation.
   * Command ***dbt docs generate*** is used to compile the ***schema.yml*** file and generate documentation.
   * Documentation details compiled by command ***dbt docs generate*** is stored in **/target/** folder of dbt project ***“learn\_dbt”***
   * ***Markdown files*** can also be created and added in ***schema.yml*** file for more sophisticated documentation. These markdown files will reside in ***/models/*** folder of dbt project ***“learn\_dbt”*** and this markdown file will be created using jinja template tag ***docs****. Details can be seen in Implementation section*
2. ***Auto-Generated Documentation Website:***

* dbt can generate a fully browsable documentation site that combines code, metadata, and data lineage.
* Command ***dbt docs serve*** is used to start the documentation webserver and it will show all the documentation details compile by command ***dbt docs generate.***
* DBT document webserver shows generic Welcome overview page, this page can be changed using markdown document. *Details can be seen in Implementation section*

1. ***Lineage Graph:***

* dbt’s documentation includes a lineage graph that shows dependencies between models, sources, and tests.

1. ***Assets folder***

* **/assets/ folder** is often used in dbt projects to store supporting files, particularly when users or teams create custom ***documentation***.
* **/assets/ folder** is not a standard folder generated or required by dbt itself.
* **/assets/ folder** needs to be added in the ***dbt\_project.yml*** file present in dbt project “***learn\_dbt***” folder. *Details can be seen in Implementation section*

### Implementation

In this section, following steps will be taken to create the documentation –

1. Add descriptions for tables and columns in ***schema.yml*** file.



1. Create a markdown file ***“fact\_review\_doc.md”*** in ***/models/*** folder and add the reference in ***schema.yml*** file. Markdown file will be created using jinja template tag ***docs.***

***markdown file “fact\_review\_doc.md”***

A screenshot of a computer

Description automatically generated

***markdown file “fact\_review\_doc.md” reference in schema.yml***

A screenshot of a computer

Description automatically generated

1. Execute command ***dbt docs generate*** to compile the documentation in ***/targets/*** folder of dbt project ***“learn\_dbt”***
2. Execute command ***dbt docs serve --host 0.0.0.0 --port 8080*** to start the dbt documentation web server and navigate to ***“http://localhost:8080/”*** to access the dbt document server.

***Below snip shows the model information of dbt docs server***

A screenshot of a computer

Description automatically generated

1. DBT Doc Server shows the lineage of the transformations as well.

***Below snippet shows the icon in document server, which will show the lineage graph.***

A screenshot of a computer

Description automatically generated

***Lineage Graph***

A blue background with white text

Description automatically generated

1. The Overview page of the dbt doc server can also be customized using markdown file and images can be added as an asset to it.

***Generic Overview Page, which shows the dbt welcome page***

A screenshot of a computer

Description automatically generated

This Overview page can be changed by creating an ***overview.md*** file in ***/models/*** folder and using jinja template in the markdown file with ***\_\_overview\_\_ as name of the markdown page in jinja template***

A screenshot of a computer

Description automatically generated

Once the overview markdown page is created, stop the dbt docs server and execute command ***dbt docs generate*** to re-generate the catalogand ***dbt docs serve --host 0.0.0.0 –port 8080*** to start the dbt docs server again.

A screenshot of a computer

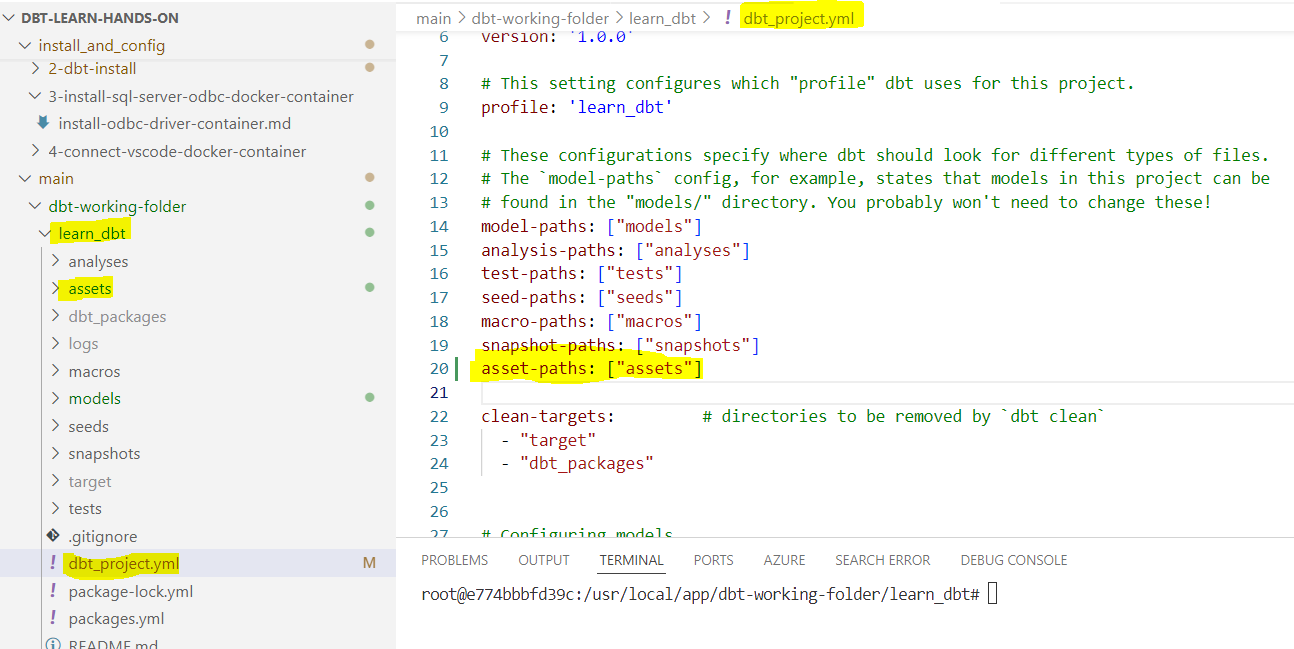
Description automatically generated

1. In the overview page, ER diagram snippet can also be added. For this –
   * Create an ***/assets/*** folder in dbt project ***“learn\_dbt”*** and add the ER diagram snippet in the folder.

A screenshot of a computer

Description automatically generated

* + Open ***dbt\_project.yml*** file and add ***/assets/*** folder in ***asset-paths*** sections.

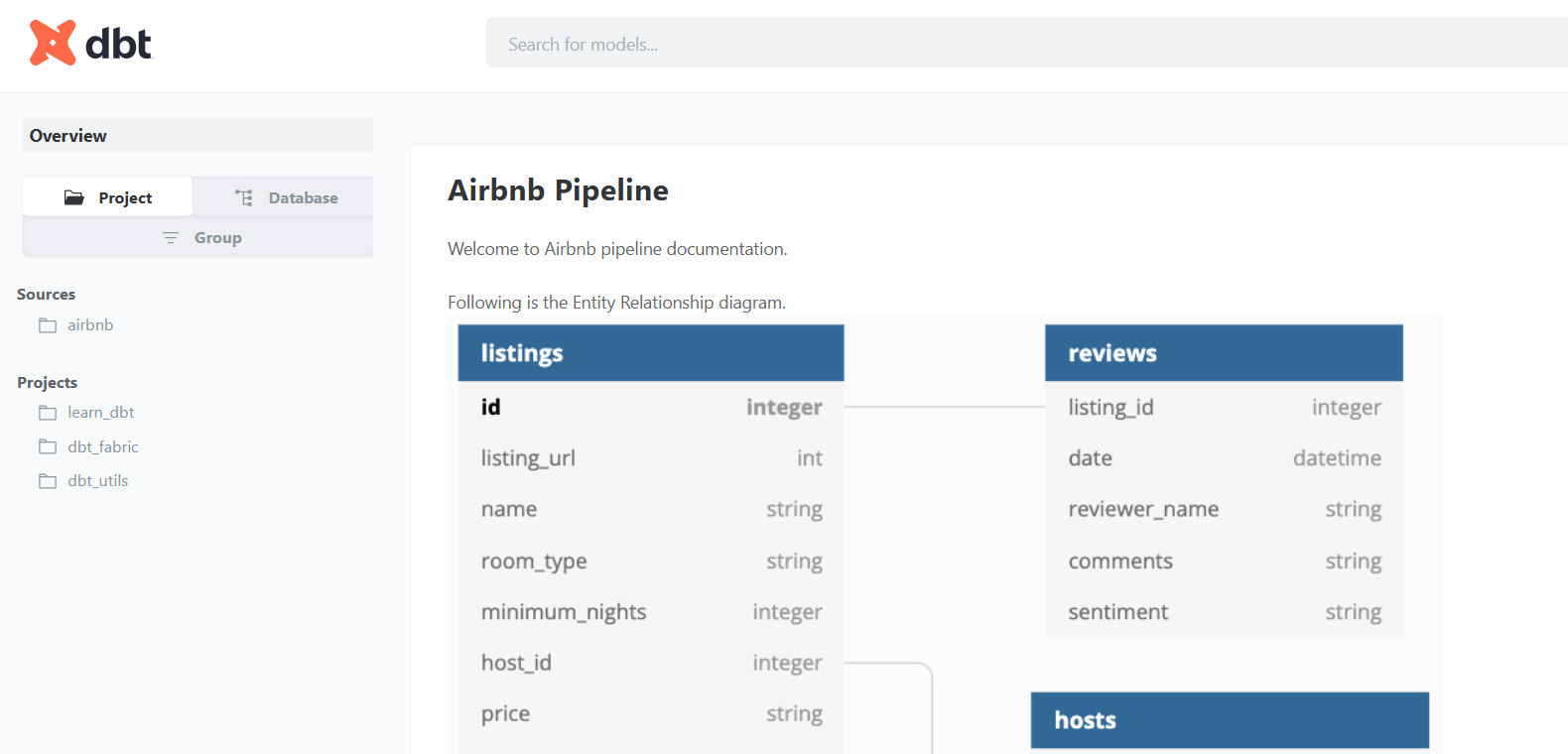


* + Open ***overview.md*** file present in ***/models/*** folder and add path of ER diagram snippet present in ***/assets/*** folder to it.

A screenshot of a computer

Description automatically generated

* + After updating the overview page, stop the dbt docs server and execute command ***dbt docs generate*** to re-generate the catalogand ***dbt docs serve --host 0.0.0.0 –port 8080*** to start the dbt docs server again.



## Analyses

***Analyses*** feature is used to create and manage ad-hoc queries, data analysis, and reporting logic that is not part of the regular transformation models.

**Key Characteristics:**

* Analyses queries ***are not materialized*** as tables in the database.
* Used for ***exploratory queries***, ***reporting logic***, or ***data investigation***.
* Analyses queriesare stored in the ***/analysis/*** folder of dbt project ***“learn\_dbt”***.
* Compile the analyses query by executing command ***dbt compile***, and the compiled query will be stored in ***/target/compiled/analyses*** folder

### Implementation

In this implementation, an analysis query is written to understand the count of reviews by full moon flag.

**Steps:**

1. Create a ***full\_moon\_analysis.sql*** file in ***/analysis/*** folder of the dbt project ***“learn\_dbt”***
2. Create a query which will calculate the *count of reviews by is\_full\_moon flag*, in this query “*mart\_seed\_full\_moon*” model will be referred.

A screenshot of a computer

Description automatically generated

1. Execute command ***dbt compile***, this will compile the query and store the compiled query in ***/target/compiled/analyses*** folder and copy the query from the compiled sql file and execute it in DB.

A screenshot of a computer

Description automatically generated

## Hooks

Hooks are SQLs that are ***executed at predefined times***. Hooks are defined in ***dbt\_project.yml*** file present in dbt project “***learn\_dbt***” and can be configured on the project, subfolder, or model level.

**Hook types:**

* ***on\_run\_start***: executed at the start of *dbt {run, seed, snapshot},* this is project level hook, it cannot be defined in models.
* ***on\_run\_end***: executed at the end of *dbt {run, seed, snapshot}*
* ***pre-hook***: executed before a model/seed/snapshot is built
* ***post-hook***: executed after a model/seed/snapshot is built

**Important Points while implementing hooks:**

* **Incorrect Placement of Hooks**
  + Hooks like ***on\_run\_start*** and ***on\_run\_end*** are project-level hooks. They cannot be specified inside the models section for individual models. They must be defined in the ***dbt\_project.yml*** file under the top-level configuration.
* **\*\*Scope of {{ this }}**
  + The {{ this }} macro is specific to a model being materialized. If you want to log actions at the model level, you should use pre-hook and post-hook model-level configurations.
* **SQL Syntax**
  + Multiline SQL statements should be wrapped in triple quotes (""") to avoid errors during YAML parsing.

### Implementation

In this implementation, “***pre-hook***” and “***post-hook***” hook will be added to capture the model execution start and end time.

**Steps:**

1. Create a Database table called “***logs***” by using below script.

|  |
| --- |
| CREATE TABLE dbt.dbo.logs  (  model varchar(100),  start\_date\_time datetime2,  end\_date\_time datetime2  ) |

1. Open ***dbt\_project.yml*** file present in dbt project “***learn\_dbt***” and add “***pre-hook***” and “***post-hook***” hook at ***model*** level, sql query in these hooks will *insert and update the log table to add model’s execution start datetime and end datetime of the model*.

A screenshot of a computer

Description automatically generated

1. Execute command ***dbt run***  and ***check table dbo.logs***

A screenshot of a computer

Description automatically generated

## Exposures

***Exposures*** are *used to document and track downstream dependencies* *of dbt models*, such *as reports, dashboards, or tools that consume the data*.

This feature is useful for understanding how the transformed data is being used by end users.

**Purpose:**

* Documenting and organizing data assets that depend on dbt models.
* Providing visibility into which dashboards, reports, or applications rely on specific models.
* Making it easier to identify the impact of changes in dbt models.

Exposures are defined in ***yaml file***, which reside in ***/models/*** folder of dbt project ***“learn\_dbt”***

### Implementation

In this implementation, a dummy report will be added in exposures yaml file.

**Steps:**

1. Create a yml file “***dashboard.yml***” in ***/models/*** folder.

A screenshot of a computer

Description automatically generated

1. Add details of the dummy dashboard in the exposure file

A screenshot of a computer

Description automatically generated

1. After adding the exposures, stop the dbt docs server and execute command ***dbt docs generate*** to re-generate the catalogand ***dbt docs serve --host 0.0.0.0 --port 8080*** to start the dbt docs server again.

A screenshot of a computer

Description automatically generated

# Tests using *dbt-expectations* package

### Great Expectations

Great Expectations is an open-source data quality tool that helps

* Understand, test, and maintain the quality of their data:
* Automated data quality checks
  + Users can define expectations for their data and have Great Expectations automatically check against them. This helps catch issues early.
* Data profiling
  + Great Expectations can profile datasets to help users understand their distribution, patterns, and quality over time.
* Integration with data platforms
  + Great Expectations integrates with many modern data platforms, including:
    - Apache Airflow
    - dbt
    - Prefect
    - Astro

***dbt-expectations*** package follows the idea of great-expectations, and this package will help to add tests to dbt models.

### dbt-expectations package Installation

For installation of *dbt-expectations* package, follow below steps –

1. Follow below link and copy the package installation command and add it to ***packages.yml*** file present in dbt project folder ***“learn\_dbt”.***

<https://github.com/calogica/dbt-expectations>

***Snippet from dbt expectations github page***

A screenshot of a computer program

Description automatically generated

***Snippet from packages.yml***

A screenshot of a computer

Description automatically generated

1. Execute command ***dbt deps*** to install the package.

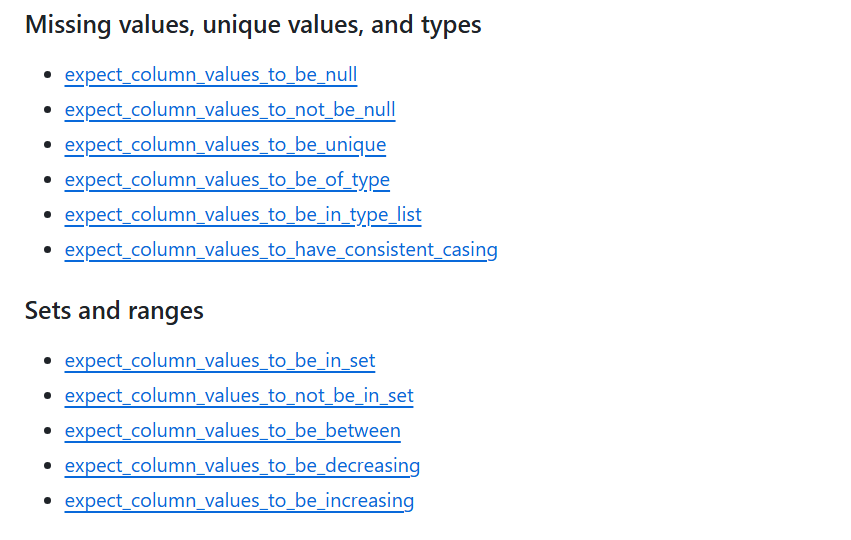
A screenshot of a computer

Description automatically generated

### dbt-expectations tests implementation

Different dbt-expectations tests can be found at below link, these tests are implemented in ***schema.yml*** file present in ***/models/*** folder of dbt project ***“learn\_dbt”*** to test the models.

<https://github.com/calogica/dbt-expectations>



These tests can be implemented for models, seeds and sources –

1. To test models, dbt-exceptions package tests are defined in ***schema.yml*** file that reside in ***/models/*** folder of dbt project ***“learn\_dbt”***
2. To test seeds, dbt-exceptions package tests are defined in ***schema.yml*** file that reside in ***/seeds/*** folder of dbt project ***“learn\_dbt”***
3. To test sources, dbt-exceptions package tests are defined in ***source.yml*** file present in ***/models/*** folder of dbt project ***“learn\_dbt”***

**Steps to implement dbt-exceptions uniqueness test –**

1. In ***schema.yml*** file that reside in ***/models/*** folder of dbt project ***“learn\_dbt”*** , add uniqueness test ***“dbt\_expectations.expect\_column\_values\_to\_be\_unique”***  in column *review\_key of model mart\_seed\_full\_moon*

A screenshot of a computer

Description automatically generated

1. Execute command “***dbt test --select mart\_seed\_full\_moon***” to run the test.

*If test fails, then Error will occur*

A close-up of text

Description automatically generated

# DBT Logging

**Logging** feature allows to output custom messages to the terminal or log files during a dbt run. This is especially useful for debugging or tracking the execution of models, tests, and macros.

The ***{{ log("message", "level") }}*** function is used for logging, where:

* ***"message"***: The content to log.
* ***"level"***: The logging level ("info" or "warn").

### Implementation

**Steps to create custom logging –**

1. Create a macro “” with 2 parameters, message and level. This macro will reside in ***/macros/*** folder of dbt project “***learn\_dbt***”

**Logging macro implementation – Added process to add comments in jinja**

A screenshot of a computer

Description automatically generated

1. Test log macro –
   * Delete log file “***dbt.log***” present in ***/logs/*** folder of dbt project ***“learn\_dbt”***

A screenshot of a computer

Description automatically generated

* + Test macro by executing “***dbt run-operation dbt\_custom\_log --args '{"message": "ABCD Custom Log Testing", "level": "info"}'*** ”

A white background with black text

Description automatically generated

* + After successful execution, the custom message will be logged in log file “***dbt.log***” present in ***/logs/*** folder of dbt project ***“learn\_dbt”***

A screenshot of a computer

Description automatically generated

1. Logging on screen –
   * If ***level parameter*** is ***Info*** or ***info = True*** is passed as the 2nd argument then the message will be displayed on screen as well along with log file ***dbt.log***

A screenshot of a computer program

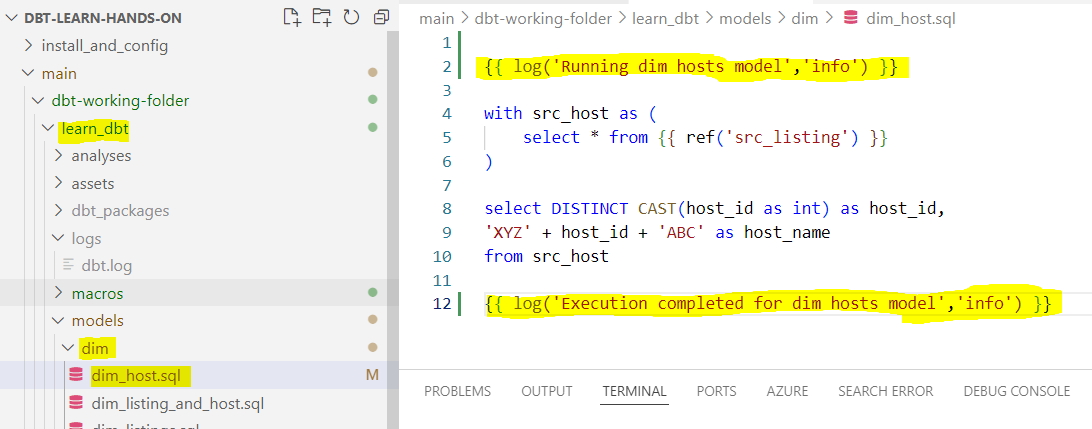
Description automatically generated

* + If in the macro ***info = False*** is passed as the 2nd argument then macro will not display the custom message on screen, it will only be logged in log file ***dbt.log***.

A screenshot of a computer

Description automatically generated

1. Call logging macro in ***model “dim\_host”*** using jinja template.



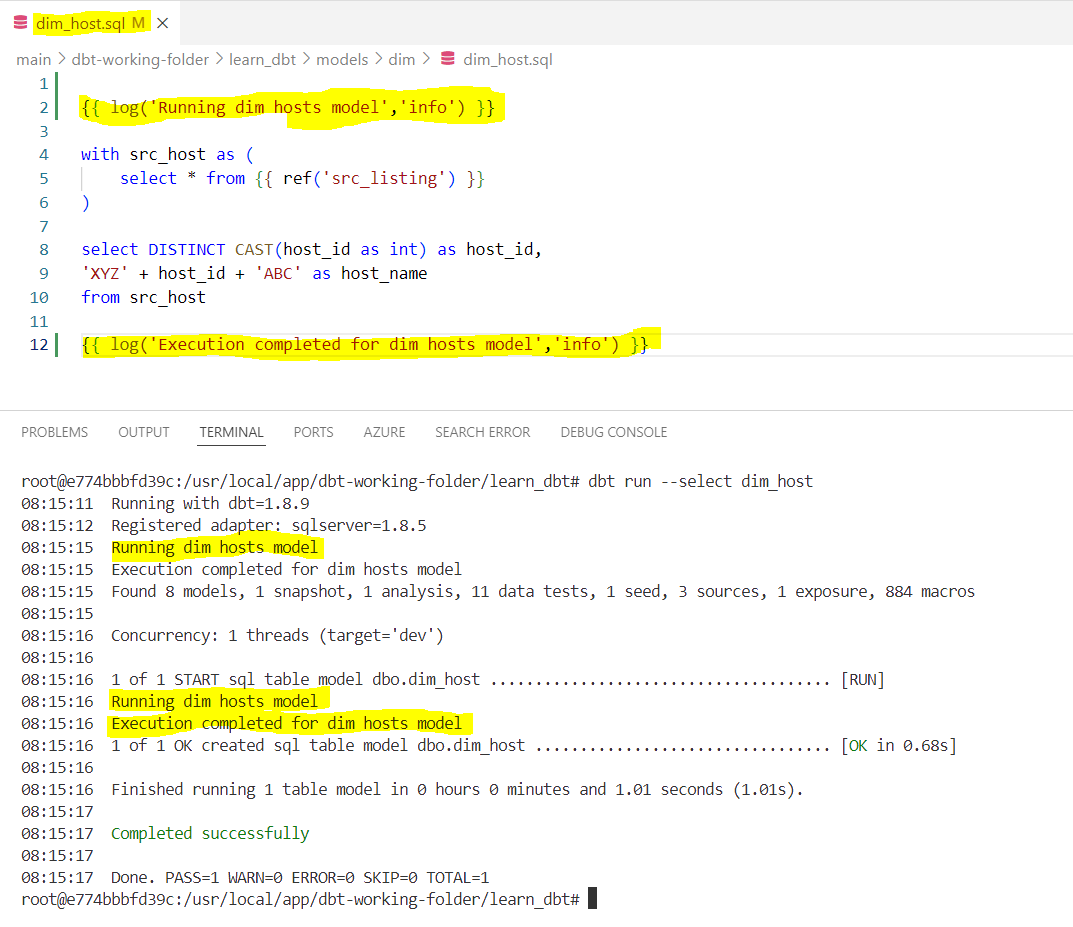
1. Execute command ***dbt run --select dim\_host*** to check whether logging is successfully done.

***Logging in dbt.log file***

A close up of a screen

Description automatically generated

***Logging on Screen***



# Variables

**Jinja Variables**

**Jinja variables** in dbt are Python-like placeholders used within dbt SQL models, macros, and YAML configurations. They are primarily used to inject dynamic content into SQL code or manage logic during model execution.

A screenshot of a computer code

Description automatically generated

***For more details on implementation, refer link –***

* <https://docs.getdbt.com/docs/build/jinja-macros>

**DBT Variables**

DBT variables provide data to models for compilation. Variables can be used to configure –

* *Timezones*
* *Avoid hardcoding table names*
* *Provide data to models to configure how they are compiled*.

**dbt variables** are defined in the ***dbt\_project.yml*** file or passed during a ***dbt run*** and are intended for customizing behaviour across models, macros, and projects. They allow external configuration and parameterization**.**

***dbt variables defined in dbt\_project.yml***

A screenshot of a computer program

Description automatically generated

***dbt variables defined in dbt run command***

A screenshot of a computer code

Description automatically generated

***For more details on implementation, refer link –***

* Var keyword: <https://docs.getdbt.com/reference/dbt-jinja-functions/var>
* DBT Project variables: <https://docs.getdbt.com/docs/build/project-variables>

# Appendix

## Important Links

* Dockerfile

<https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/>

* Dockerfile CMD, RUN and ENTRYPOINT

<https://www.docker.com/blog/docker-best-practices-choosing-between-run-cmd-and-entrypoint/>

* Docker mount

<https://docs.docker.com/engine/storage/bind-mounts/>

* DBT SQL Server Installation

<https://docs.getdbt.com/docs/core/connect-data-platform/mssql-setup>

* Jinja Template Documentation

<https://jinja.palletsprojects.com/en/stable/templates/#for>

* DBT Jinja and macro

<https://docs.getdbt.com/docs/build/jinja-macros>

* DBT Jinja functions

<https://docs.getdbt.com/reference/dbt-jinja-functions>

* DBT Jinja adapters

<https://docs.getdbt.com/reference/dbt-jinja-functions/adapter>

* DBR Packages

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

* Great Expectations Package

<https://github.com/great-expectations/great_expectations>

<https://greatexpectations.io/>

* DBT Expectations

<https://github.com/calogica/dbt-expectations>

* Var keyword

<https://docs.getdbt.com/reference/dbt-jinja-functions/var>

* DBT Project variables

<https://docs.getdbt.com/docs/build/project-variables>

* DBT Models can be orchestrated by using *Dagster or Airflow*.

<https://www.youtube.com/watch?v=MhCuxTDlVkE>