DBT

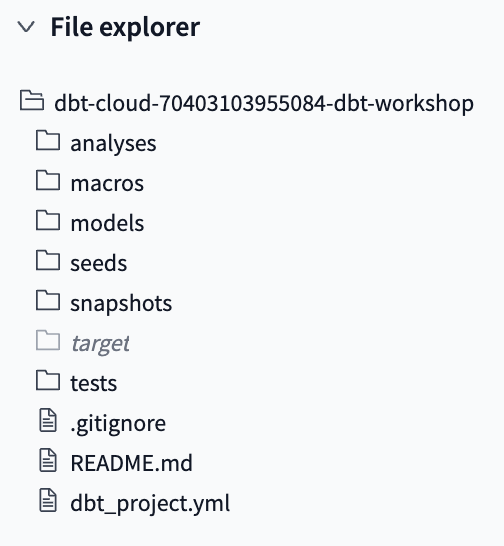
Data build tool (dbt) is an [open-source](https://en.wikipedia.org/wiki/Open-source_software" \o "Open-source software) command line tool that helps analysts and engineers [transform](https://en.wikipedia.org/wiki/Data_transformation" \o "Data transformation) data in their [warehouse](https://en.wikipedia.org/wiki/Data_warehouse) more effectively.

It takes care of DDL commands while you can focus on implementing the business logic and designing great models. The code lies in git and it runs in the data warehouse. dbt is just an interface that facilitates the process.

Tips & Tricks

In the IDE type underscore ( \_ ) twice to show the intelligence dropdown.

**dbt project structure**



**dbt\_project.yml**

dbt project configuration file. Here we set the project name, materialization modes, etc. If you don’t specify the materialization method, objects will be created as a view (default)

Materialization example:

models:

jaffle\_shop:

+materialized: table

**Models**

All queries that create the model live here.

Similar to python, simple and straightforward code is better than complex difficult to read; that’s why it is encouraged to use CTEs and write comments when needed.

As a best practice you should split queries that clean the data from those that transform it.

Materialization can be set by either adding the following header at the top of the sql file

{{ config(materialized='view') }} or updating the dbt\_project.yml file

**Tests**

Typical tests to ensure basic data quality checks such as count, duplicates, unique keys, etc.

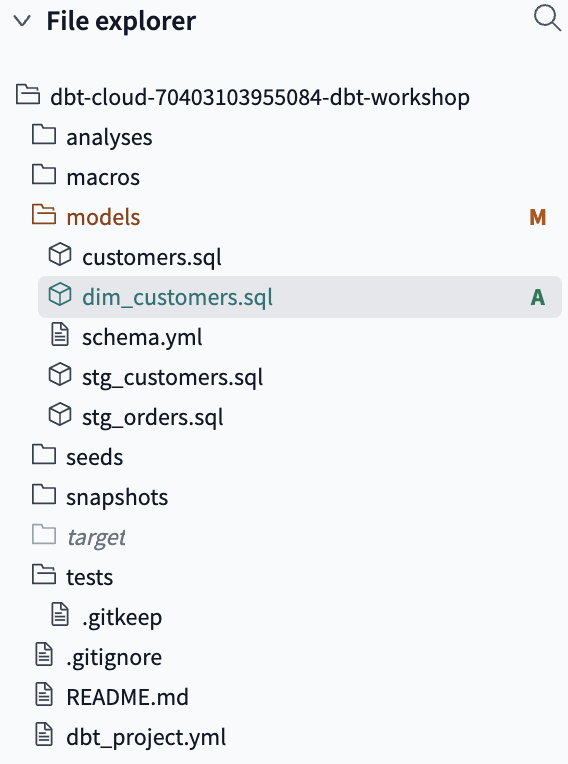
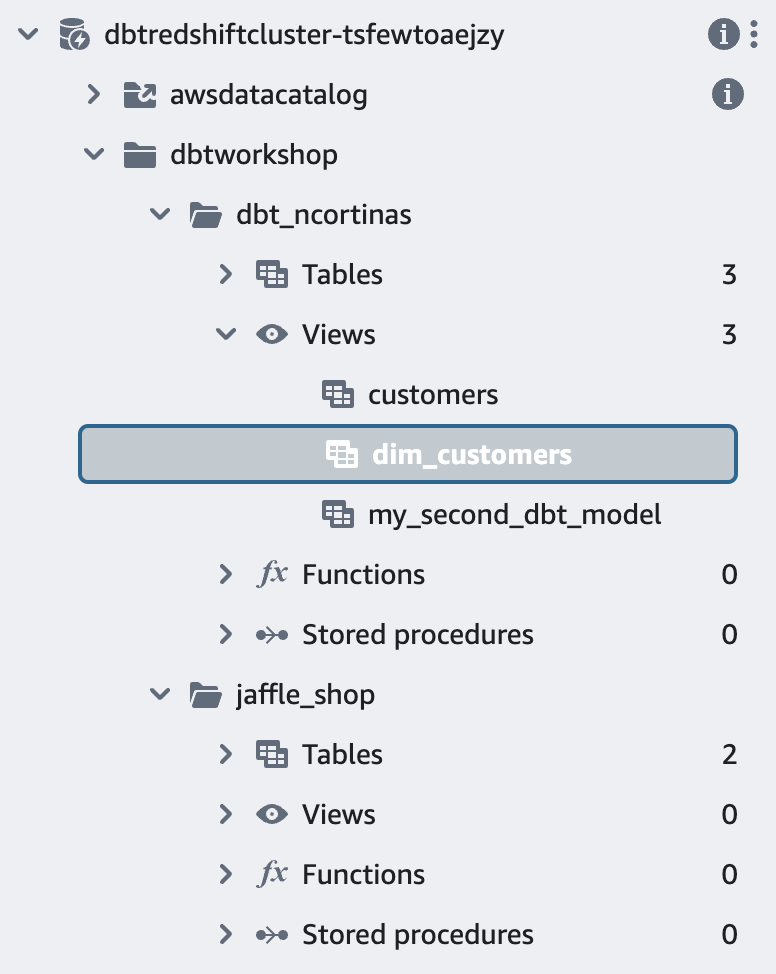
**Commands and Buttons**

**Preview**: executes the code in the active sql file by sending it to the data warehouse and retrieving the first 100 rows.

**Compile**: Adds the final sql code into the preview window resolving all ref functions into actual sql code. This won’t bring the code that creates the module being referenced with the ref function though.

**Format:**

**dbt run**: Finds all models in the project, and creates them in the data warehouse. dbt run creates all new tables/views in your schema in the database. In the example below, we have just created the dim\_customer.sql file and executed the dbt run command. dbt cloud on the left, redshift on the right.

Keep in mind that dbt run will execute all sql files. If you want to run only one or run all files in a given folder, you should add the select parameter: dbt run --select model\_name > dbt run --select dim\_customers dbt run -s folder\_name

By adding a + after the model name you will ensure all downstream models are built as well.

Upstream means going closer to raw while downstream is getting closer to the end product.

**dbt source freshness:** Checks when the source table was updated last. For this to work you need to set up freshness parameters in the source .yml file (see below)

**dbt test:**

Run tests by using the dbt test command: dbt test

You can run tests for a specific file using the select parameter: dbt test --select file\_name

For generic or singular tests only, you can use the following syntax

dbt test --select test\_type:generic

dbt test --select test\_type:singular

You can run tests for a specific group of tables:

dbt test --select sources:\*

**dbt build:**

Mix of dbt run and dbt test. dbt build executes run and test layer by layer, so you don’t create any table if the previous model didn’t pass the tests.

**dbt docs generate:**

Parses all files and generates the documentation accordingly.

**dbt Functions**

**ref**

Pulls information from a model that has been already materialized while creating the dependencies and lineage between them. This approach of creating multiple smaller modules linked together is called modularity.

In order to use this function you have to create the models in another sql file and the pointing to them in the final model using the function.

with customers as (

select \* from {{ ref('stg\_customers') }}

)

Notice you are not adding .sql at the end, this is because you are pointing to the obect that was built, not to the sql file.

**source**

Pointer to raw tables. Its syntax is as follows:

{{ source('jaffle\_shop\_sources', 'orders') }} > "dbtworkshop"."jaffle\_shop"."orders"

For this to work, you need to create a source .yml file (see below)

**Naming Conventions**

* **Sources** (src) refer to the raw table data that have been built in the warehouse through a loading process. (We will cover configuring Sources in the Sources module)
* **Staging** (stg) refers to models that are built directly on top of sources. These have a one-to-one relationship with sources tables. These are used for very light transformations that shape the data into what you want it to be. These models are used to clean and standardize the data before transforming data downstream. Note: These are typically materialized as views.
* **Intermediate** (int) refers to any models that exist between final fact and dimension tables. These should be built on staging models rather than directly on sources to leverage the data cleaning that was done in staging.
* **Fact** (fct) refers to any data that represents something that occurred or is occurring. Examples include sessions, transactions, orders, stories, votes. These are typically skinny, long tables.
* **Dimension** (dim) refers to data that represents a person, place or thing. Examples include customers, products, candidates, buildings, employees.

**Sources in dbt**

Sources are raw tables coming from the ELT tool. Sources are configured in a .yml file in the models directory that looks like this:

version: 2

sources:

- name: jaffle\_shop\_sources

database: dbtworkshop

schema: jaffle\_shop

freshness:

warn\_after: {count: 12, period: hour}

error\_after: {count: 24, period: hour}

loaded\_at\_field: \_etl\_loaded\_at

tables:

- name: customers\_dt

loaded\_at\_field: etl\_load\_dt

freshness:

warn\_after: {count: 1, period: minute}

error\_after: {count: 24, period: hour}

- name: orders

Replace hard coded sources or raw tables in the from clasue with:

{{ source('jaffle\_shop\_sources', 'orders') }} > "dbtworkshop"."jaffle\_shop"."orders"

The identifier parameter is useful when the table name is too long or complex and you want to use a alias for simplicity in your model:

- name: alias

identifier: actual table name in database

Are Freshness parameter allows you to run sanity checks before runing your model. It checks how recent/fresh the data is. It can be applied in a table or schema level. loaded\_at\_field is the column in the table with timestamps we have to check.

The benefits of using the Sources in dbt are:

* Multiple tables from a single source can be configured in one place.
* Sources are easily identified as green nodes in the Lineage Graph.
* You can use dbt source freshness to check the freshness of raw tables.

**Tests in dbt**

Tests are sql queries which sole purpose is to check whether everything is fine and you can build the model.

In dbt, there are two types of tests - generic tests and singular tests:

* **Generic tests** are a way to validate your data models and ensure data quality. These tests are predefined and can be applied to any column of your data models to check for common data issues. They are written in YAML files.
* **Singular tests** are data tests defined by writing specific SQL queries that return records which fail the test conditions. These tests are referred to as "singular" because they are one-off assertions that are uniquely designed for a single purpose or specific scenario within the data models.

There are 4 types of built-in generic tests:

* **unique**

asserts that each value in a column is unique.

* **not\_null**

asserts there are no null values in a column.

* **relationships**

asserts that every value in a column exits in a predefined list.

* **accepted\_values**

asserts that every value in a column exists in the table of another column to maintain referential integrity.

Generic tests are configured in the schema.yml file. This file will have other information such as table and column descriptions as well. It looks like this:

version: 2

models:

- name: customers

freshness: null

description: One record per customer

columns:

- name: customer\_id

description: Primary key

tests:

- unique

- not\_null

- name: first\_order\_date

description: NULL when a customer has not yet placed an order.

For Singular test we have to create a sql file that checks that specific case in the test directory. This query should return values when the assertion fails. For example, if we want to check if the amount column has negative values, the query should select those rows that have negative values.

In other words, if the test query returns 0 rows, the test is passed.

Tests can be applied to source tables or models. All .yml files can have a test segmet

Run tests by using the dbt test command: dbt test

You can run tests for a specific file using the select parameter: dbt test --select file\_name

For generic or singular tests only, you can use the following syntax

dbt test --select test\_type:generic

dbt test --select test\_type:singular

**Documentation in dbt**

dbt has made documenting your code seemless and easy, this is because it generates your documentation reading from the schema .yml file. It might not be a flashy documentation but it will serve its purpose. Documentation can be added to models and sources alike.

Document your code by adding a description tag under name in the schema .yml file.

For thorough descriptions, consider using the docs macro. This macro reads from a .md file and feeds the documentation from there. It is useful to keep the .yml file nice and tidy.

The docs markdown file should have a begin and end clause, as well as a name. This file is added to the schema file using the following syntax. Note that this description could be added under any name tag (model, source, table, column, etc)

description: '{{ doc("order\_status") }}'

docs.md file example:

{% docs order\_status %}

One of the following values:

| status | definition |

|----------------|--------------------------------------------------|

| placed | Order placed, not yet shipped |

| shipped | Order has been shipped, not yet been delivered |

| completed | Order has been received by customers |

| return pending | Customer indicated they want to return this item |

| returned | Item has been returned |

{% enddocs %}

Generate the documentation by running the command dbt docs generate

**Understanding Deployment**

Deployment and Production environmnet are the same thing. It is the main branch where the business runs. It usually has it owns production schema. Dev schema should be a mirror of the production one.