**dbt(Data Build Tool) Tutorial**

Sep 29, 2021 · 10 min read

* [1. Introduction](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#1-introduction)
* [2. Dbt, the T in ELT](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#2-dbt-the-t-in-elt)
* [3. Project](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#3-project)
  + [3.1. Prerequisites](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#31-prerequisites)
  + [3.2. Configurations and connections](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#32-configurations-and-connections)
    - [3.2.1. profiles.yml](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#321-profilesyml)
    - [3.2.2. dbt\_project.yml](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#322-dbt_projectyml)
  + [3.3 Data flow](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#33-data-flow)
    - [3.3.1. Source](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#331-source)
    - [3.3.2. Snapshots](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#332-snapshots)
    - [3.3.3. Staging](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#333-staging)
    - [3.3.4. Marts](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#334-marts)
      * [3.3.4.1. Core](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#3341-core)
      * [3.3.4.2. Marketing](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#3342-marketing)
  + [3.4. dbt run](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#34-dbt-run)
  + [3.5. dbt test](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#35-dbt-test)
  + [3.6. dbt docs](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#36-dbt-docs)
  + [3.7. Scheduling](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#37-scheduling)
* [4. Conclusion](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#4-conclusion)
* [5. Further reading](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#5-further-reading)
* [6. References](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/#6-references)

**1. Introduction**

If you are a student, analyst, engineer, or anyone in the data space and are curious about what dbt is and how to use it. Then this post is for you.

If you are keen to understand why dbt is widely used, please read [**this article**](https://www.startdataengineering.com/post/advantages-of-using-dbt-data-build-tool/).

**2. Dbt, the T in ELT**

In an ELT pipeline, the raw data is loaded(EL) into the data warehouse. Then the raw data is transformed into usable tables, using SQL queries run on the data warehouse.

**Note:** If you are interested in learning to write efficient SQL for data processing, checkout my e-book: [**Efficient Data Processing in SQL**](https://josephmachado.gumroad.com/l/analyticalsql)

dbt provides an easy way to create, transform, and validate the data within a data warehouse. dbt does the T in ELT (Extract, Load, Transform) processes.

In dbt, we work with **models, which is a sql file with a select statement**. These models can depend on other models, have tests defined on them, and can be created as tables or views. The names of models created by dbt are their file names.

E.g. The file **[dim\_customers.sql](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/marts/core/dim_customers.sql" \t "_blank)**represents the model named dim\_customers. This model depends on the models [**stg\_eltool\_\_customers**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/staging/stg_eltool__customers.sql)and [**stg\_eltool\_\_state**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/staging/stg_eltool__state.sql). The dim\_customers model can then be referenced in other model definitions.

with customers as (

select \*

from {{ ref('stg\_eltool\_\_customers') }}

),

state as (

select \*

from {{ ref('stg\_eltool\_\_state') }}

)

select c.customer\_id,

c.zipcode,

c.city,

c.state\_code,

s.state\_name,

c.datetime\_created,

c.datetime\_updated,

c.dbt\_valid\_from::TIMESTAMP as valid\_from,

CASE

WHEN c.dbt\_valid\_to IS NULL THEN '9999-12-31'::TIMESTAMP

ELSE c.dbt\_valid\_to::TIMESTAMP

END as valid\_to

from customers c

join state s on c.state\_code = s.state\_code

We can **define tests to be run on processed data** using dbt. Dbt allows us to create 2 types of tests, they are

1. Generic tests: Unique, not\_null, accepted\_values, and relationships tests per column defined in YAML files. E.g. see **[core.yml](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/marts/core/core.yml" \t "_blank)**
2. Bespoke (aka one-off) tests: Sql scripts created under the tests folder. They can be any query. They are successful if the sql scripts do not return any rows, else unsuccessful.

E.g. The **[core.yml](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/marts/core/core.yml" \t "_blank)**file contains tests for dim\_customers and fct\_orders models.

version: 2

models:

- name: dim\_customers

columns:

- name: customer\_id

tests:

- not\_null # checks if customer\_id column in dim\_customers is not null

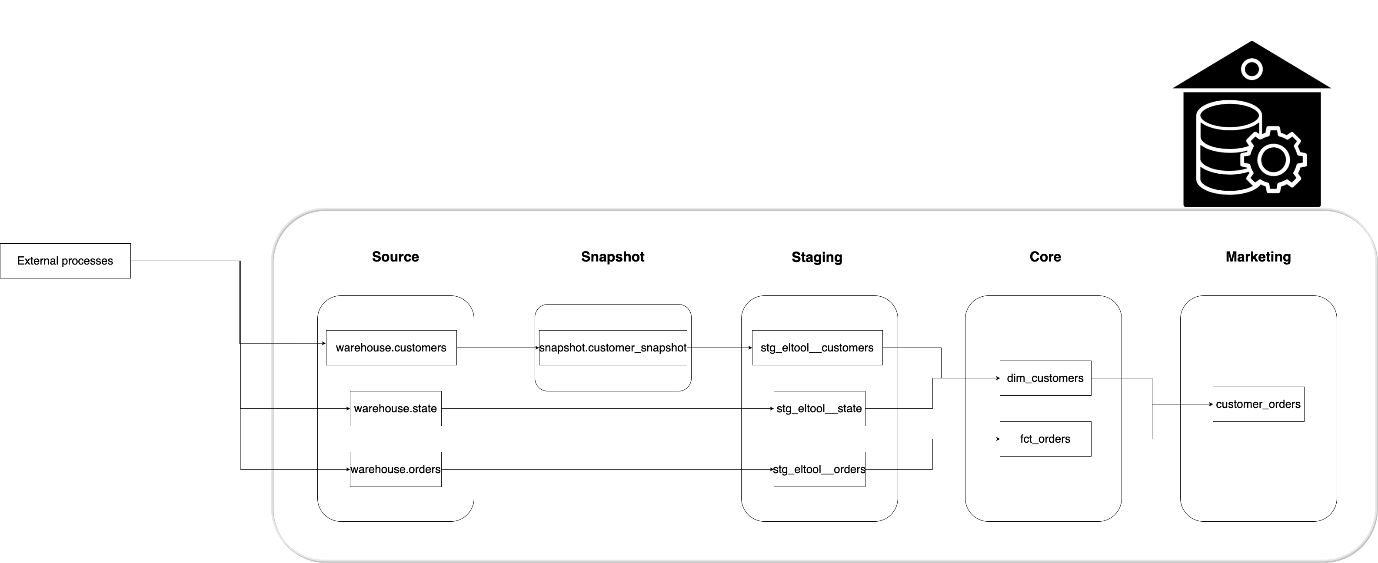
- name: fct\_orders

**3. Project**

We are asked by the marketing team to create a denormalized table customer\_orders, with information about every order placed by the customers. Let’s assume the customers and orders data are loaded into the warehouse by a process.

The process used to bring these data into our data warehouse is the EL part. This can be done using a vendor service like [Fivetran](https://fivetran.com/why-fivetran" \t "_blank) , [Stitch](https://www.stitchdata.com/) , or open-source services like [Singer](https://www.singer.io/) , [Airbyte](https://airbyte.io/" \t "_blank) or using a custom service.

Let’s see how our data is transformed into the final denormalized table.



We will follow data warehouse best practices like having [staging tables](https://www.startdataengineering.com/post/what-and-why-staging/) , testing, using [slowly changing dimensions type 2](https://www.startdataengineering.com/post/how-to-join-fact-scd2-tables/) and naming conventions.

**3.1. Prerequisites**

To code along you will need

1. [Docker](https://docs.docker.com/get-docker/) and [Docker compose](https://docs.docker.com/compose/install/)
2. [dbt](https://docs.getdbt.com/dbt-cli/installation/)
3. [pgcli](https://www.pgcli.com/install)
4. [git](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)

Clone the git repo and start the data warehouse docker container

git clone https://github.com/josephmachado/simple\_dbt\_project.git

export DBT\_PROFILES\_DIR=$(pwd)

docker compose up -d

cd simple\_dbt\_project

By default dbt will look for warehouse connections in the file ~/.dbt/profiles.yml. The DBT\_PROFILES\_DIR environment variable tells dbt to look for the profiles.yml file in the current working directory.

You can also create a dbt project using dbt init. This will provide you with a sample project, which you can modify.

In the simple\_dbt\_project folder you will see the following folders.

.

├── analysis

├── data

├── macros

├── models

│ ├── marts

│ │ ├── core

│ │ └── marketing

│ └── staging

├── snapshots

└── tests

1. analysis: Any .sql files found in this folder will be compiled to raw sql when you run dbt compile. They will not be run by dbt but can be copied into any tool of choice.
2. data: We can store raw data that we want to be loaded into our data warehouse. This is typically used to store small mapping data.
3. macros: Dbt allows users to create macros, which are sql based functions. These macros can be reused across our project.

We will go over the models, snapshots, and tests folders in the below sections.

**3.2. Configurations and connections**

Let’s set the warehouse connections and project settings.

**3.2.1. profiles.yml**

Dbt requires a profiles.yml file to contain data warehouse connection details. We have defined the warehouse connection details at /simple\_dbt\_project/profiles.yml.

The target variable defines the environment. The default is dev. We can have multiple targets, which can be specified when running dbt commands.

The profile is sde\_dbt\_tutorial. The profiles.yml file can contain multiple profiles for when you have more than one dbt project.

**3.2.2. dbt\_project.yml**

In this file, you can define the profile to be used and the paths for different types of files (see \*-paths).

Materialization is a variable that controls how dbt creates a model. By default, every model will be a view. This can be overridden in dbt\_profiles.yml. We have set the models under models/marts/core/ to materialize as tables.

# Configuring models

models:

sde\_dbt\_tutorial:

# Applies to all files under models/marts/core/

marts:

core:

materialized: table

**3.3 Data flow**

We will see how the customer\_orders table is created from the source tables. These transformations follow warehouse and dbt best practices.

**3.3.1. Source**

Source tables refer to tables loaded into the warehouse by an EL process. Since dbt did not create them, we have to define them. This definition enables referring to the source tables using the [source](https://docs.getdbt.com/docs/building-a-dbt-project/using-sources/#using-sources) function. For e.g. {{ source('warehouse', 'orders') }} refers to the warehouse.orders table. We can also define tests to ensure that the source data is clean.

* Source definition: **[sde\_dbt\_tutorial/models/staging/src\_eltool.yml](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/staging/src_eltool.yml" \t "_blank)**
* Test definitions: **[sde\_dbt\_tutorial/models/staging/src\_eltool.yml](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/staging/src_eltool.yml" \t "_blank)**

**3.3.2. Snapshots**

A business entity’s attributes change over time. These changes should be captured in our data warehouse. E.g. a user may move to a new address. This is called slowly changing dimensions, in data warehouse modeling.

Read [**this article**](https://www.startdataengineering.com/post/how-to-join-fact-scd2-tables/)to understand the importance of storing historical data changes, and what slowly changing dimensions are.

Dbt allows us to easily create these slowly changing dimension tables (type 2) using the snapshot feature. When creating a snapshot, we need to define the database, schema, strategy, and columns to identify row updates.

dbt snapshot

Dbt creates a snapshot table on the first run, and on consecutive runs will check for changed values and update older rows. We simulate this as shown below

pgcli -h localhost -U dbt -p 5432 -d dbt

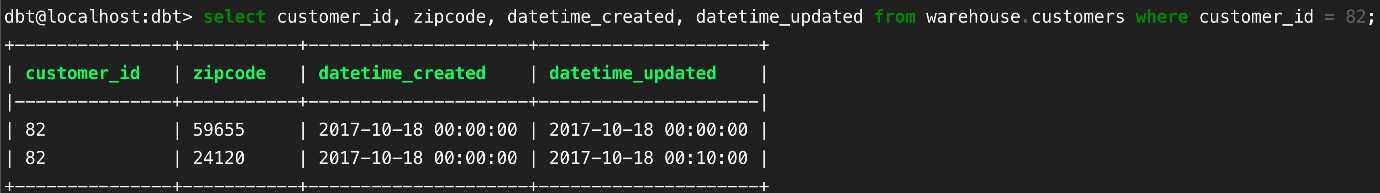
# password1234

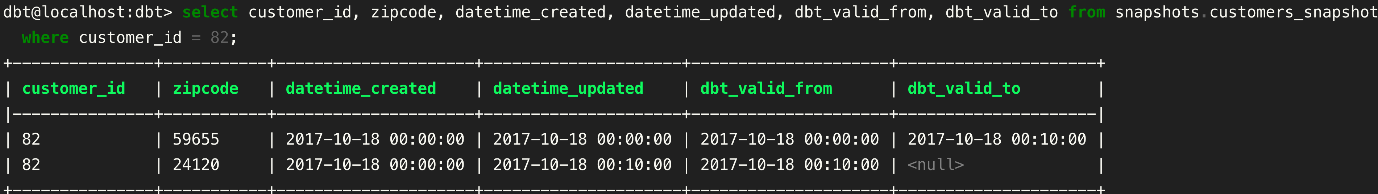
COPY warehouse.customers(customer\_id, zipcode, city, state\_code, datetime\_created, datetime\_updated)

FROM '/input\_data/customer\_new.csv' DELIMITER ',' CSV HEADER;

Run the snapshot command again

dbt snapshot

Raw data

Snapshot table

The row with zipcode 59655 had its dbt\_valid\_to column updated. The dbt from and to columns represent the time range when the data in that row is representative of customer 82.

* Model definition: **[sde\_dbt\_tutorial/snapshots/customers.sql](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/snapshots/customers.sql" \t "_blank)**

**3.3.3. Staging**

The staging area is where **raw data is cast into correct data types, given consistent column names, and prepared to be transformed into models used by end-users**.

You might have noticed the eltool in the staging model names. If we use Fivetran to EL data, our models will be named stg\_fivetran\_\_orders and the YAML file will be stg\_fivetran.yml.

In stg\_eltool\_\_customers.sql we use the ref function instead of the source function because this model is derived from the snapshot model. In dbt, we can use the ref function to refer to any models created by dbt.

* Test definitions: **[sde\_dbt\_tutorial/models/staging/stg\_eltool.yml](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/staging/stg_eltool.yml" \t "_blank)**
* Model definitions: [**sde\_dbt\_tutorial/models/staging/stg\_eltool\_\_customers.sql**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/staging/stg_eltool__customers.sql)**,**[**stg\_eltool\_\_orders.sql**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/staging/stg_eltool__orders.sql)**,**[**stg\_eltool\_\_state.sql**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/staging/stg_eltool__state.sql)

**3.3.4. Marts**

Marts consist of the core tables for end-users and business vertical-specific tables. In our example, we have a marketing department-specific folder to defined the model requested by marketing.

**3.3.4.1. Core**

The core defines the fact and dimension models to be used by end-users. The fact and dimension models are materialized as tables, for performance on frequent use. The fact and dimension models are based on [kimball dimensional model](https://www.kimballgroup.com/2003/01/fact-tables-and-dimension-tables/" \t "_blank) .

* Test definitions: **[sde\_dbt\_tutorial/models/marts/core/core.yml](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/marts/core/core.yml" \t "_blank)**
* Model definitions: [**sde\_dbt\_tutorial/models/staging/dim\_customers**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/marts/core/dim_customers.sql)**,**[**fct\_orders.sql**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/marts/core/fct_orders.sql)

Dbt offers four generic tests, unique, not\_null, accepted\_values, and relationships. We can create one-off (aka bespoke) tests under the Tests folder. Let’s create a sql test script that checks if any of the customer rows were duplicated or missed. If the query returns one or more records, the tests will fail. Understanding this script is left as an exercise for the reader.

* One-off test: [**sde\_dbt\_tutorial/tests/assert\_customer\_dimension\_has\_no\_row\_loss.sql**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/tests/assert_customer_dimension_has_no_row_loss.sql)

**3.3.4.2. Marketing**

In this section, we define the models for marketing end users. A project can have multiple business verticals. Having one folder per business vertical provides an easy way to organize the models.

* Test definitions: [**sde\_dbt\_tutorial/models/marts/marketing/marketing.yml**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/marts/marketing/marketing.yml)
* Model definitions: [**sde\_dbt\_tutorial/models/marts/marketing/customer\_orders.sql**](https://github.com/josephmachado/simple_dbt_project/blob/master/sde_dbt_tutorial/models/marts/marketing/customer_orders.sql)

**3.4. dbt run**

We have the necessary model definitions in place. Let’s create the models.

dbt snapshot

dbt run

...

Finished running 4 view models, 2 table models ...

The stg\_eltool\_\_customers model requires snapshots.customers\_snapshot model. But snapshots are not created on dbt run ,so we run dbt snapshot first.

Our staging and marketing models are as materialized views, and the two core models are materialized as tables.

**The snapshot command should be executed independently from the run command** to keep snapshot tables up to date. If snapshot tables are stale, the models will be incorrect. There is snapshot freshness monitoring in [dbt cloud UI](https://docs.getdbt.com/docs/dbt-cloud/using-dbt-cloud/cloud-snapshotting-source-freshness/" \l "source-freshness-snapshot-frequency" \t "_blank) .

**3.5. dbt test**

With the models defined, we can run tests on them. Note that, unlike standard testing, these tests run after the data has been processed. You can run tests as shown below.

dbt test

...

Finished running 10 tests...

The above command runs all the tests defined within the project. You can log into the data warehouse to see the models.

pgcli -h localhost -U dbt -p 5432 -d dbt

# password is password1234

select \* from warehouse.customer\_orders limit 3;

\q

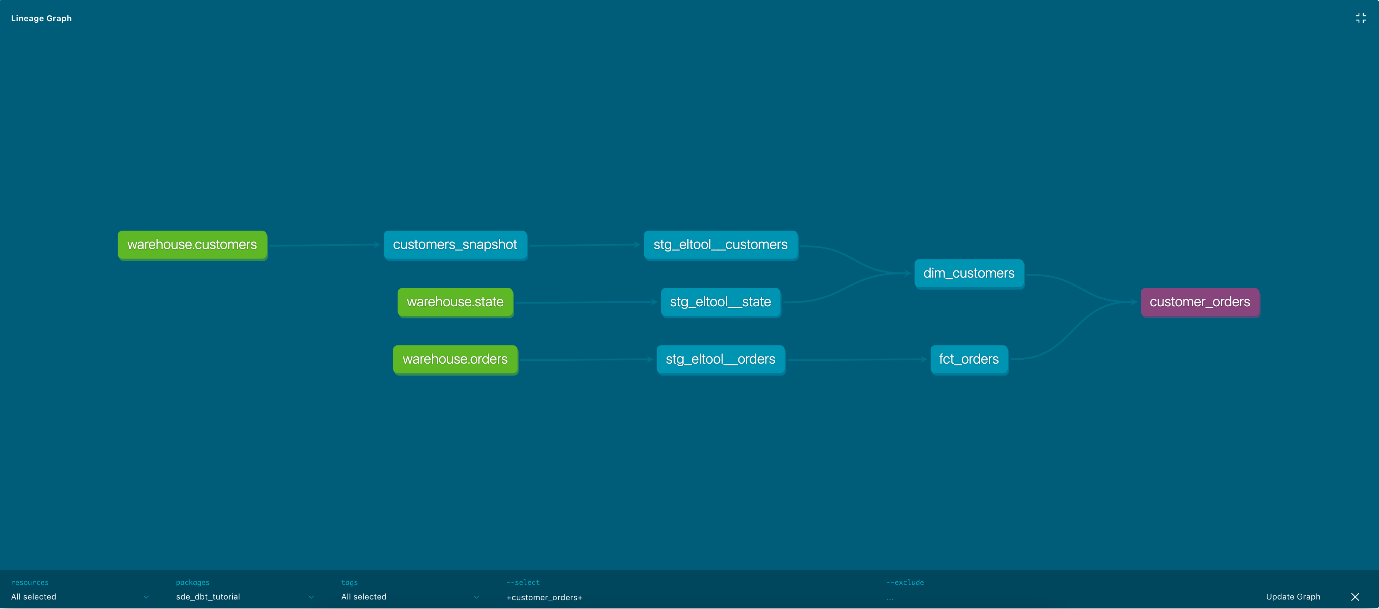
**3.6. dbt docs**

One of the powerful features of dbt is its docs. To generate documentation and serve them, run the following commands:

dbt docs generate

dbt docs serve

You can visit http://localhost:8080 to see the documentation. Navigate to customer\_orders within the sde\_dbt\_tutorial project in the left pane. Click on the view lineage graph icon on the lower right side. The lineage graph shows the dependencies of a model. You can also see the tests defined, descriptions (set in the corresponding YAML file), and the compiled sql statements.



**3.7. Scheduling**

We have seen how to create snapshots, models, run tests and generate documentation. These are all commands run via the cli. Dbt compiles the models into sql queries under the target folder (not part of git repo) and executes them on the data warehouse.

To schedule dbt runs, snapshots, and tests we need to use a scheduler. Dbt cloud is a great option to do easy scheduling. Checkout [**this article**](https://www.startdataengineering.com/post/cicd-dbt/#schedule-jobs)to learn how to schedule jobs with dbt cloud. The dbt commands can be run by other popular schedulers like cron, Airflow, Dagster, etc.

**4. Conclusion**

Dbt is a great choice to build your ELT pipelines. Combining data warehouse best practices, testing, documentation, ease of use, [**data CI/CD**](https://www.startdataengineering.com/post/cicd-dbt/), [community support](https://discourse.getdbt.com/) and a great cloud offering, dbt has set itself up as an essential tool for data engineers. **Learning and understanding dbt can significantly improve your odds of landing a DE job** as well.

To recap, we went over

1. Dbt project structure
2. Setting up connections
3. Generating SCD2 (aka snapshots) with dbt
4. Generating models following best practices
5. Testing models
6. Generating and viewing documentation

dbt can help you make your ELT pipelines stable and development fun. If you have any questions or comments, please leave them in the comment section below.

**5. Further reading**

1. [dbt CI/CD](https://www.startdataengineering.com/post/cicd-dbt/)
2. [Why use dbt?](https://www.startdataengineering.com/post/advantages-of-using-dbt-data-build-tool/)
3. [ETL v ELT](https://www.startdataengineering.com/post/elt-vs-etl/)
4. [Unit test SQL in dbt](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/)
5. [Simple data platform with Stitch and dbt](https://www.startdataengineering.com/post/build-a-simple-data-engineering-platform/)

**6. References**

1. [dbt snapshot best practices](https://docs.getdbt.com/docs/building-a-dbt-project/snapshots/#snapshot-query-best-practices)
2. [dbt project structure](https://discourse.getdbt.com/t/how-we-structure-our-dbt-projects/355)

**How to set up a dbt data-ops workflow, using dbt cloud and Snowflake**

Feb 28, 2021 · 11 min read

* [Introduction](https://www.startdataengineering.com/post/cicd-dbt/#introduction)
* [Pre-requisites](https://www.startdataengineering.com/post/cicd-dbt/#pre-requisites)
* [Setting up the data-ops pipeline](https://www.startdataengineering.com/post/cicd-dbt/#setting-up-the-data-ops-pipeline)
  + [Snowflake](https://www.startdataengineering.com/post/cicd-dbt/#snowflake)
  + [Local development environment](https://www.startdataengineering.com/post/cicd-dbt/#local-development-environment)
  + [dbt cloud](https://www.startdataengineering.com/post/cicd-dbt/#dbt-cloud)
    - [Connect to Snowflake](https://www.startdataengineering.com/post/cicd-dbt/#connect-to-snowflake)
    - [Link to github repository](https://www.startdataengineering.com/post/cicd-dbt/#link-to-github-repository)
    - [Setup deployment(release/prod) environment](https://www.startdataengineering.com/post/cicd-dbt/#setup-deploymentreleaseprod-environment)
    - [Setup CI](https://www.startdataengineering.com/post/cicd-dbt/#setup-ci)
    - [PR -> CI -> merge cycle](https://www.startdataengineering.com/post/cicd-dbt/#pr---ci---merge-cycle)
    - [Schedule jobs](https://www.startdataengineering.com/post/cicd-dbt/#schedule-jobs)
    - [Host data documentation](https://www.startdataengineering.com/post/cicd-dbt/#host-data-documentation)
* [Conclusion and next steps](https://www.startdataengineering.com/post/cicd-dbt/#conclusion-and-next-steps)
* [Further reading](https://www.startdataengineering.com/post/cicd-dbt/#further-reading)
* [References](https://www.startdataengineering.com/post/cicd-dbt/#references)

**Introduction**

With companies realizing the importance of having correct data, there has been a lot of attention on the data-ops side of things. Data-ops refers to managing different environments and ensuring software engineering best practices(eg CI/CD) for your data pipelines. If you are using dbt and wondering

How do I test my transformation logic locally ?

How do I ensure my data transformation logic works as expected on production data ?

How to automate data CI on pull requests ?

How do I grant appropriate permissions to dbt in snowflake ?

Then this post is for you. In this post we go over the entire process of setting up dbt cloud with snowflake as the data warehouse and code repository hosted on github.

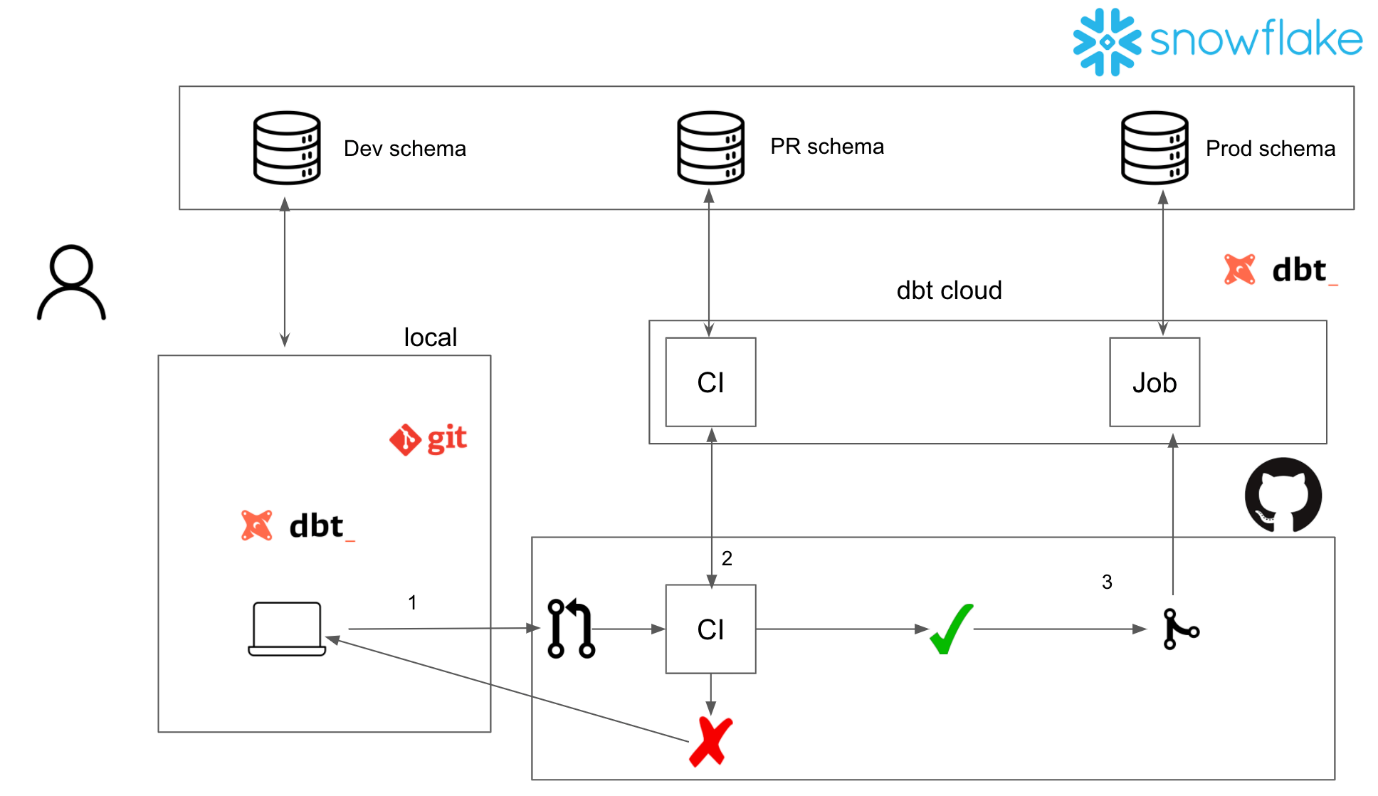
**Pre-requisites**

1. [Snowflake account](https://signup.snowflake.com/)
2. [dbt account](https://www.getdbt.com/signup/)
3. [github account](https://github.com/)
4. [docker](https://docs.docker.com/get-docker/)
5. [Basic understanding of git branch, PR and merge](https://guides.github.com/introduction/flow/)
6. [dbt basics](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/)

PR refers to a pull request.

**Setting up the data-ops pipeline**

At a high level, this is what our data-ops setup will look like.



**Snowflake**

Snowflake offers extremely granular access controls. We will follow the patterns mentioned in [this post](https://blog.getdbt.com/how-we-configure-snowflake/) to setup our snowflake permissions.

We will create the following entities

1. WAREHOUSE: TRANSFORMING, in snowflake a warehouse is the machine in which the execution happens.
2. DATABASE: PROD, denotes where the data is stored.
3. SCHEMA: RAW, ANALYTICS, schemas within a database. The raw incoming data will land in the RAW schema. End users should only be provided access to the ANALYTICS schema.
4. ROLE: TRANSFORMER. In snowflake a role can be assigned permissions. Then users are allocated roles as necessary.
5. USER: DBT\_CLOUD. A user account for dbt cloud.

We also do the same for a development environment, with the text \_DEV appended to it.

| **User** | **Role** | **Schema** | **Database** | **Warehouse** |
| --- | --- | --- | --- | --- |
| DBT\_CLOUD | TRANSFORMER | RAW (R), ANALYTICS (RW) | PROD | TRANSFORMING |
| DBT\_CLOUD\_DEV | TRANSFORMER\_DEV | RAW (R), ANALYTICS (RW) | DEV | TRANSFORMING\_DEV |

Log into your snowflake UI, open a worksheet and run the following commands as ACCOUNTADMIN role to create the entities specified above.

USE ROLE ACCOUNTADMIN; -- you need accountadmin for user creation, future grants

DROP USER IF EXISTS DBT\_CLOUD;

DROP USER IF EXISTS DBT\_CLOUD\_DEV;

DROP ROLE IF EXISTS TRANSFORMER;

DROP ROLE IF EXISTS TRANSFORMER\_DEV;

DROP DATABASE IF EXISTS PROD CASCADE;

DROP DATABASE IF EXISTS DEV CASCADE;

DROP WAREHOUSE IF EXISTS TRANSFORMING;

DROP WAREHOUSE IF EXISTS TRANSFORMING\_DEV;

-- creating a warehouse

CREATE WAREHOUSE TRANSFORMING WITH WAREHOUSE\_SIZE = 'XSMALL' WAREHOUSE\_TYPE = 'STANDARD' AUTO\_SUSPEND = 300 AUTO\_RESUME = TRUE COMMENT = 'Warehouse to transform data';

-- creating database

CREATE DATABASE PROD COMMENT = 'production data base';

-- creating schemas

CREATE SCHEMA "PROD"."RAW" COMMENT = 'landing zone for raw data';

CREATE SCHEMA "PROD"."ANALYTICS" COMMENT = 'data layer for end user';

-- creating an access role

CREATE ROLE TRANSFORMER COMMENT = 'Role for dbt';

-- granting role permissions

GRANT USAGE,OPERATE ON WAREHOUSE TRANSFORMING TO ROLE TRANSFORMER;

GRANT USAGE,CREATE SCHEMA ON DATABASE PROD TO ROLE TRANSFORMER;

GRANT USAGE ON SCHEMA "PROD"."RAW" TO ROLE TRANSFORMER;

GRANT ALL ON SCHEMA "PROD"."ANALYTICS" TO ROLE TRANSFORMER;

GRANT SELECT ON ALL TABLES IN SCHEMA "PROD"."RAW" TO ROLE TRANSFORMER;

GRANT SELECT ON FUTURE TABLES IN SCHEMA "PROD"."RAW" TO ROLE TRANSFORMER;

-- creating user and associating with role

CREATE USER DBT\_CLOUD PASSWORD='abc123' DEFAULT\_ROLE = TRANSFORMER MUST\_CHANGE\_PASSWORD = true;

GRANT ROLE TRANSFORMER TO USER DBT\_CLOUD;

-----------------------------------------------------------------------------------------------

-- DEV

-- creating a warehouse

CREATE WAREHOUSE TRANSFORMING\_DEV WITH WAREHOUSE\_SIZE = 'XSMALL' WAREHOUSE\_TYPE = 'STANDARD' AUTO\_SUSPEND = 300 AUTO\_RESUME = TRUE COMMENT = 'Dev warehouse to transform data';

-- cloning prod database (this clones schemas and tables as well)

CREATE DATABASE DEV CLONE PROD;

-- creating an access role

CREATE ROLE TRANSFORMER\_DEV COMMENT = 'Dev role for dbt';

-- granting role permissions

GRANT USAGE,OPERATE ON WAREHOUSE TRANSFORMING\_DEV TO ROLE TRANSFORMER\_DEV;

GRANT USAGE,CREATE SCHEMA ON DATABASE DEV TO ROLE TRANSFORMER\_DEV;

GRANT USAGE ON SCHEMA "DEV"."RAW" TO ROLE TRANSFORMER\_DEV;

GRANT ALL ON SCHEMA "DEV"."ANALYTICS" TO ROLE TRANSFORMER\_DEV;

GRANT SELECT ON ALL TABLES IN SCHEMA "DEV"."RAW" TO ROLE TRANSFORMER\_DEV;

GRANT SELECT ON FUTURE TABLES IN SCHEMA "DEV"."RAW" TO ROLE TRANSFORMER\_DEV;

-- creating user and associating with role

CREATE USER DBT\_CLOUD\_DEV PASSWORD='abc123' DEFAULT\_ROLE = TRANSFORMER\_DEV MUST\_CHANGE\_PASSWORD = true;

GRANT ROLE TRANSFORMER\_DEV TO USER DBT\_CLOUD\_DEV;

With this setup, adding a new developer to your team would just be creating a user and granting them TRANSFORMER\_DEV role permissions in snowflake. Now that we have our snowflake access setup, we can use this for development and deployment of dbt. Log in to snowflake console with

username: DBT\_CLOUD

password: abc123

You will be prompted for a password change. Do the same for DBT\_CLOUD\_DEV. You should have 2 logins now, in addition to your original account. Note that DBT\_CLOUD and DBT\_CLOUD\_DEV has access to only PROD and DEV databases respectively.

**Local development environment**

Create a new repository called dbt\_development in your github account. After that, in your terminal clone the repository at <https://github.com/josephmachado/dbt_development.git> . Initialize your own repository and link it to the github repository that you created.

git clone https://github.com/josephmachado/dbt\_development.git

cd dbt\_development

rm -rf .git # remove git information

git init

# You need a `dbt\_development` repository in your github account.

git remote add origin https://github.com/your-github-username/dbt\_development.git

git branch -M main

git add .

git commit -m 'first commit'

git push -u origin main

or [fork](https://docs.github.com/en/github/getting-started-with-github/fork-a-repo) it to make a contribution.

In the dbt\_development directory, fill in profiles.yml file with the DBT\_CLOUD\_DEV password. For the account use your snowflake account. Use [this to figure out your account](https://docs.getdbt.com/reference/warehouse-profiles/snowflake-profile#account) .

config:

send\_anonymous\_usage\_stats: False

default:

outputs:

dev: # User-Password config

type: snowflake

account: your-snowflake-account-id-here

user: DBT\_CLOUD\_DEV

password: your-password-here

role: TRANSFORMER\_DEV

database: DEV

warehouse: TRANSFORMING

schema: dbt\_your-username-here

threads: 1

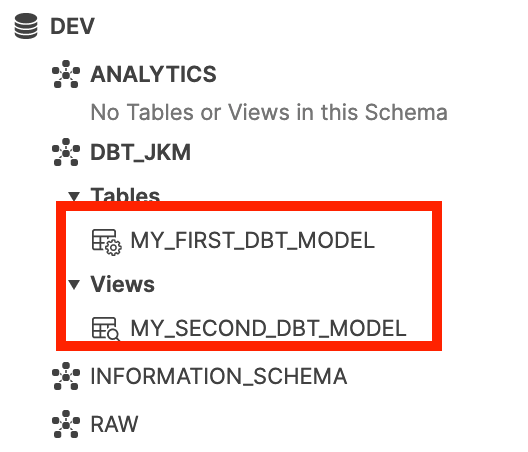
client\_session\_keep\_alive: False

target: dev

Now that you have the credentials setup, you can run dbt locally. We will use the official docker image from Fishtown Analytics to run the dbt commands.

docker run --rm -v $(pwd):/usr/app -v $(pwd):/root/.dbt fishtownanalytics/dbt:0.19.0 run

If you log in to your snowflake console as DBT\_CLOUD\_DEV, you will be able to see a schema called dbt\_your-username-here(which you setup in profiles.yml). This schema will contain a table my\_first\_dbt\_model and a view my\_second\_dbt\_model. These are sample models that are generated by dbt as examples.



You can also run tests, generate documentation and serve documentation locally as shown below.

docker run --rm -v $(pwd):/usr/app -v $(pwd):/root/.dbt fishtownanalytics/dbt:0.19.0 test

docker run --rm -v $(pwd):/usr/app -v $(pwd):/root/.dbt fishtownanalytics/dbt:0.19.0 docs generate

docker run --rm -ip 8080:8080 -v $(pwd):/usr/app -v $(pwd)/:/root/.dbt fishtownanalytics/dbt:0.19.0 docs serve

# Ctrl+C to exit

docker run --rm -ip 8080:8080 -v $(pwd):/usr/app -v $(pwd)/:/root/.dbt fishtownanalytics/dbt:0.19.0 clean

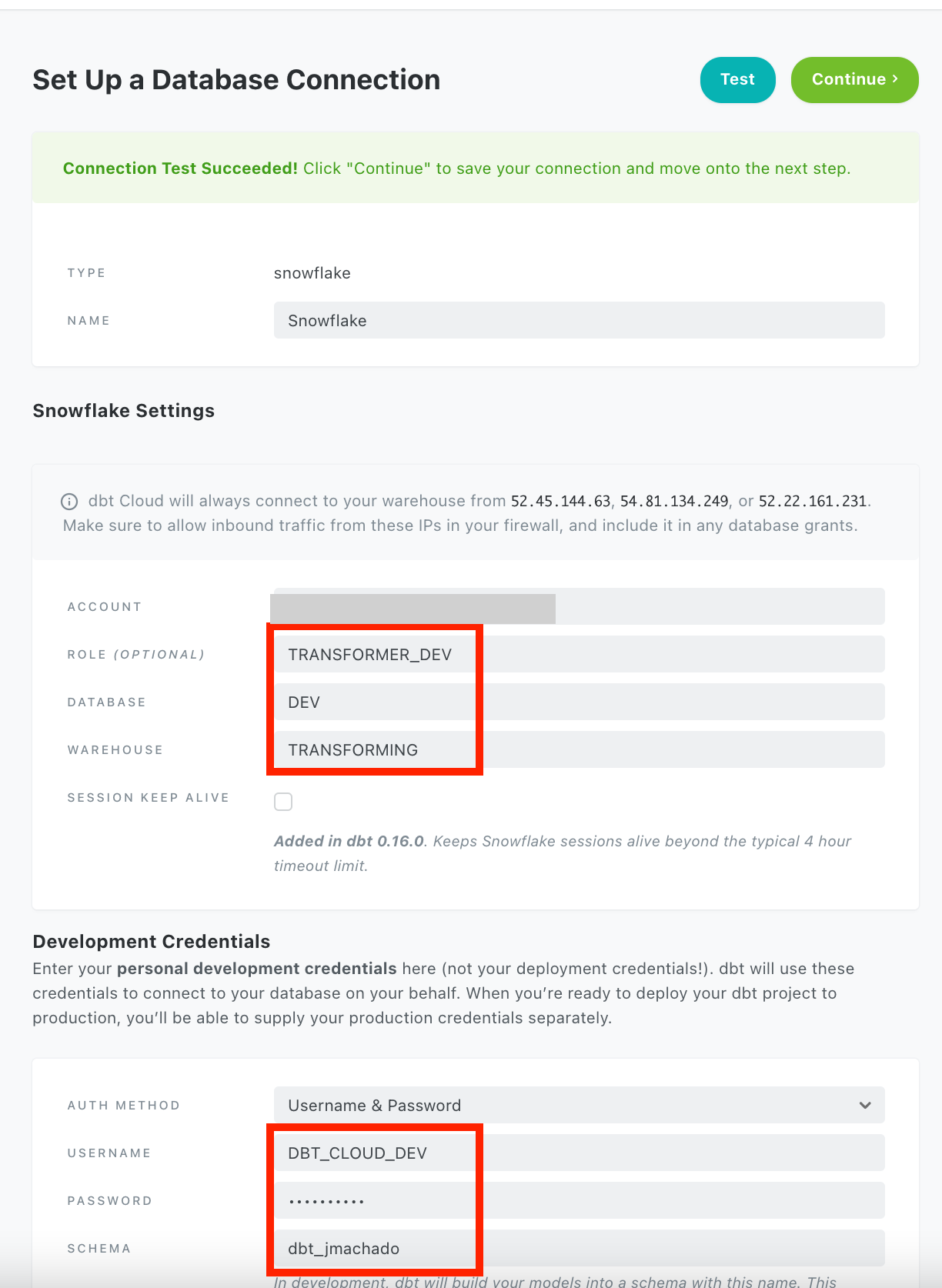
When using dbt locally, it will create the schema you specified in the profiles.yml file. Assuming each developer has a unique name, this will create a unique schema per developer in the DEV database. This way, multiple engineers can develop simultaneously, have access to the same data and create their data models in their custom schema without affecting others’.

**dbt cloud**

Now that we have a local development environment set up, we can set up our dbt cloud account. The first time you log in, you will be taken through the setup process.

**Connect to Snowflake**

Choose Snowflake as your data warehouse and in the connection settings, use your DBT\_CLOUD\_DEV username and password for the development environment.



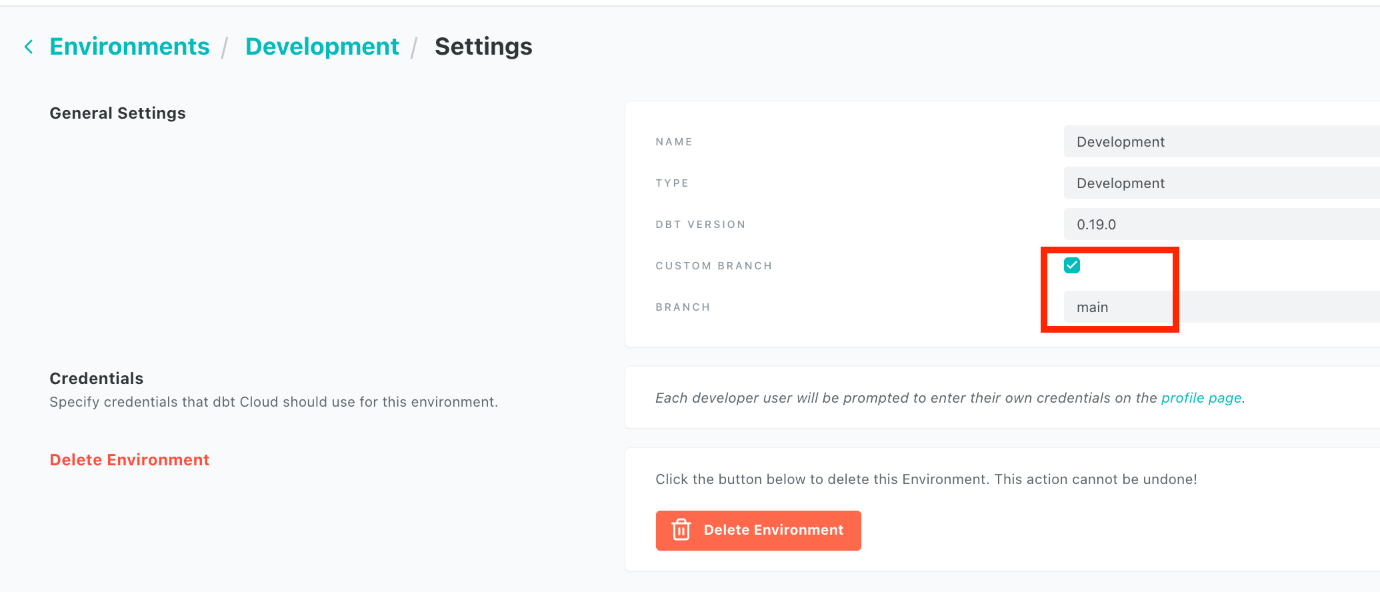
This enables you to develop in dbt cloud’s IDE.

**Link to github repository**

The next step will be connecting to a repository. When connecting to your github account, make sure to connect using the Github button and not the git url. This will enable us to do CI on pull requests on that repository. As a rule of thumb, only provide dbt app with access to your dbt repository.

| **Click github icon** | **grant dbt github repo access** |
| --- | --- |
| dbt github | dbt github 2 |

After linking, make sure to go to Hmaburger icon -> Environments and choose Development and select the custom branch checkbox and set it to main. This will ensure that dbt cloud uses the main branch and not the default setting of master branch.



**Setup deployment(release/prod) environment**

In the above section you have set up a dbt development environment. In order to use dbt in production, you will need to configure a deployment environment. This corresponds to your production(aka release, prod) environment. Go to your project by clicking on Hamburger icon -> project and then select the Deployment Environment link. In the next page, enter your deployment credentials and make sure to choose the custom branch checkbox and set it to main.

| **Click deployment** | **use production details** |
| --- | --- |
| dbt deploy env 1 | dbt deploy env 2 |

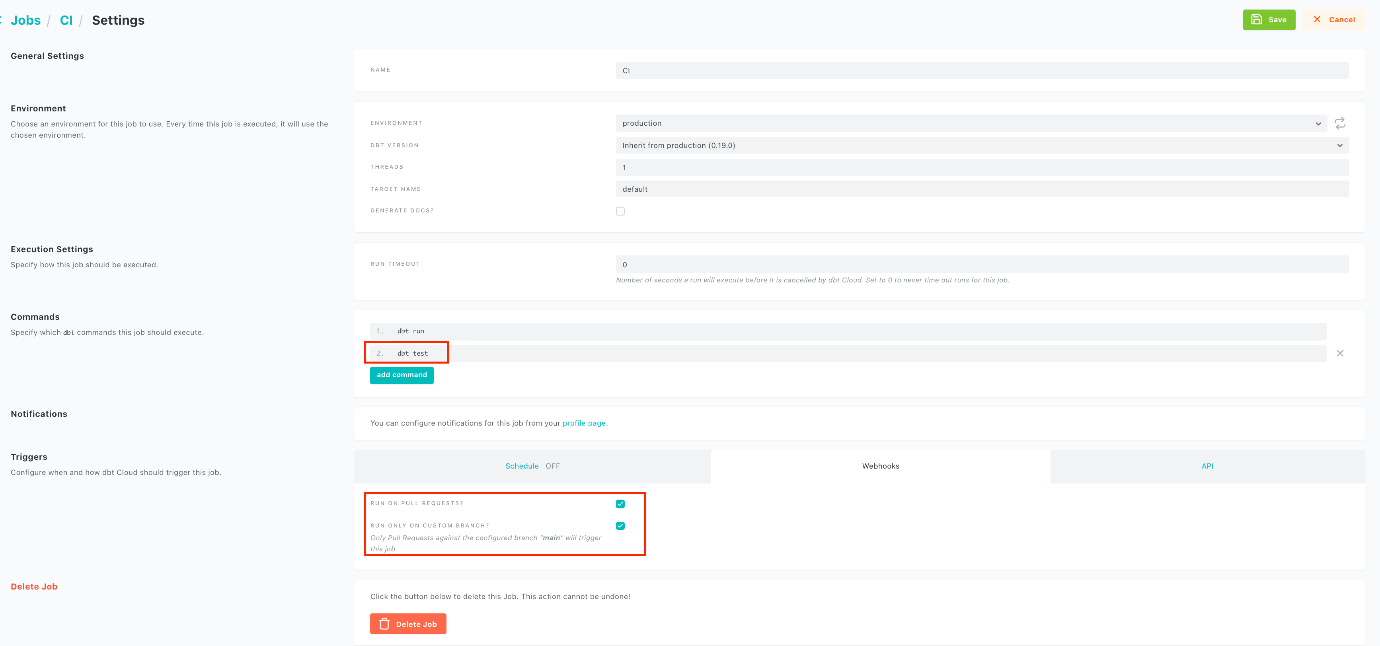
**Setup CI**

You can setup dbt cloud to run a CI job on every pull request into your main branch. After this, on every PR, dbt cloud will do the following

1. Create a unique schema in your chosen environment. Schema name is determined by the PR id and prefixed with dbt\_cloud\_pr\_\*.
2. Create the models in the unique schema and run the defined tests.
3. After the run, add ✅ or ❌ indicator to the PR.

This CI feature **allows us to test against the production data without actually creating production data assets**. In order to leverage this feature, we need to create a job in the dbt cloud UI. Click on the hamburger icon -> Jobs, in the job creation UI make sure to

1. add dbt test as an additional command.
2. switch off the schedule checkbox.
3. check Run on Pull Requests checkbox.



**PR -> CI -> merge cycle**

Now that you have a CI job setup, lets see how this works. Make a git branch for your project.

git checkout -b simple-test-branch

Make a simple change to models/example/my\_first\_dbt\_model.sql file by adding the following two lines.

**my\_first\_dbt\_model.sql**

select 1 as id

union all

select null as id

+ union all

+ select 2 as id

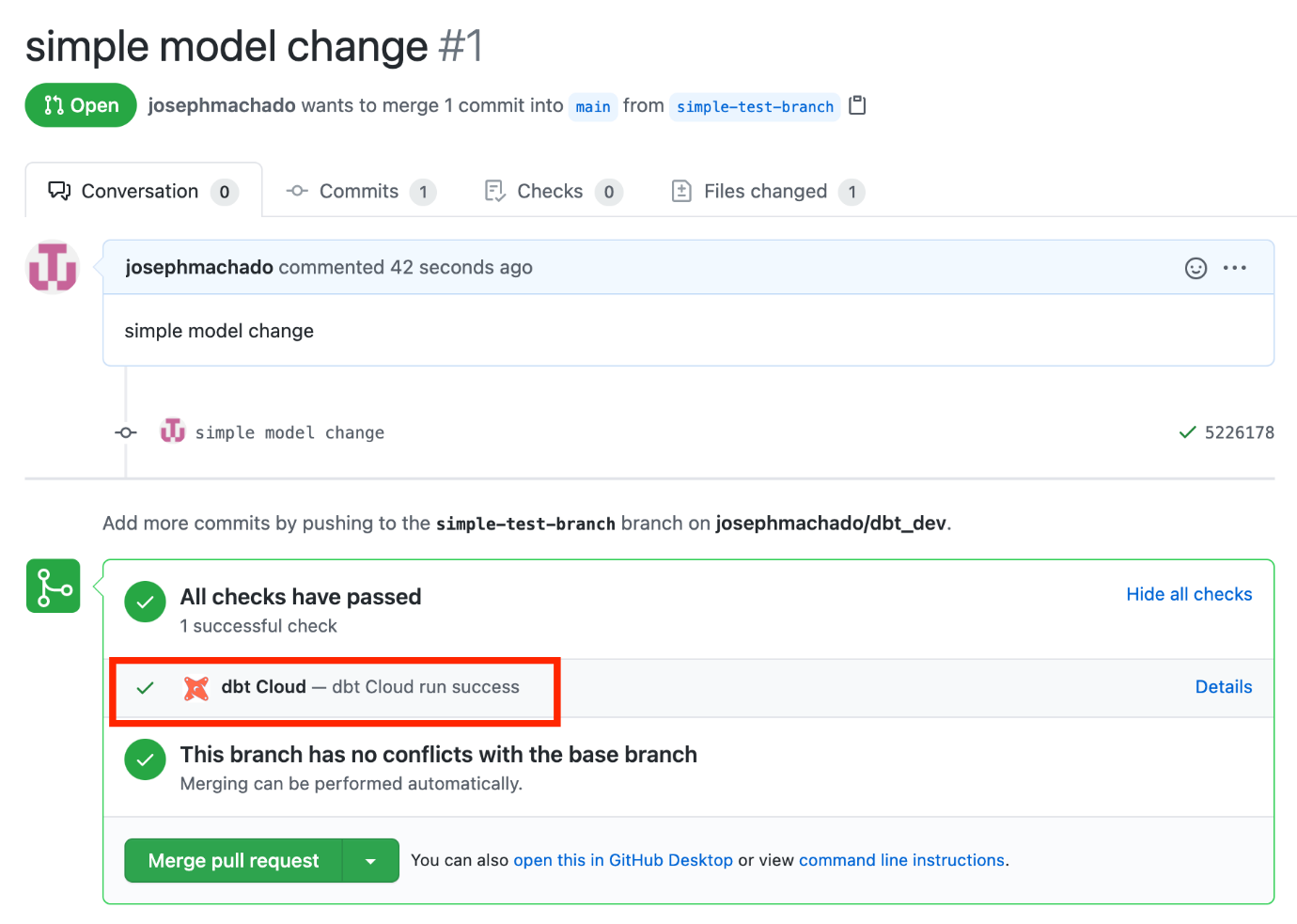
Now commit and push to remote repository that you setup in the [Github](https://www.startdataengineering.com/post/cicd-dbt/" \l "github) section, as shown below.

git add models/example/my\_first\_dbt\_model.sql

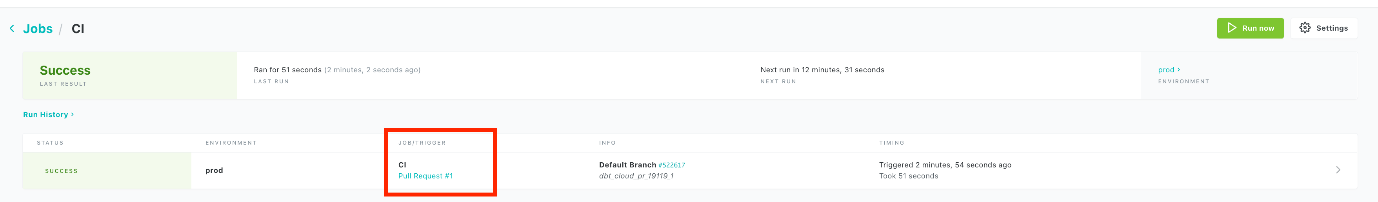
git commit -m 'simple model change'

git push -f origin simple-test-branch

In the dbt\_development repository in your github UI click on the create pull request prompt. When you create a PR, a CI job is run and its status is added to your PR.



You can see the list of runs in the dbt UI under the respective job. In our case, it’s the CI job that we created in the previous section.



Note that this creates a temporary schema in your production database. The data models are created in this schema. Dbt cloud runs tests on the data in this PR specific schema. From the above image, you can see that the temporary schema created for that PR was named dbt\_cloud\_pr\_19119\_1.

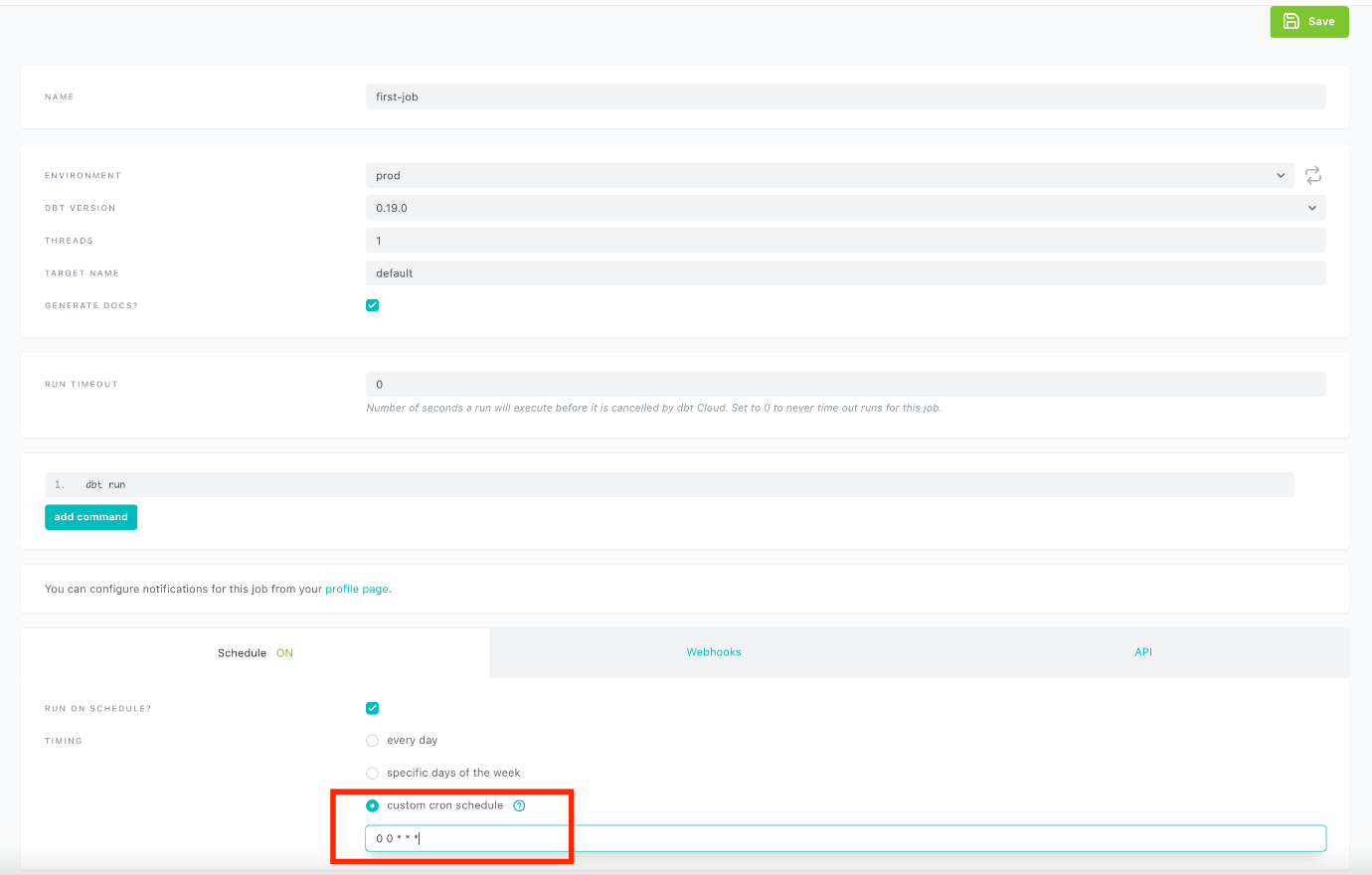
**NOTE:** This pr specific schema should be removed by dbt cloud after your PR is merged, [but there have been cases where it is not removed](https://stackoverflow.com/questions/64670623/ci-temporary-schema-not-deleting) . This is a known issue.

After the CI jobs pass, you can be confident that your code will work as expected on your current production data. You can merge your pull request.

**Schedule jobs**

Now that our code is in production, we can start scheduling jobs. Dbt is a compiler and runner, but not a scheduler. This is where dbt cloud helps, by letting us schedule jobs with the required frequency.

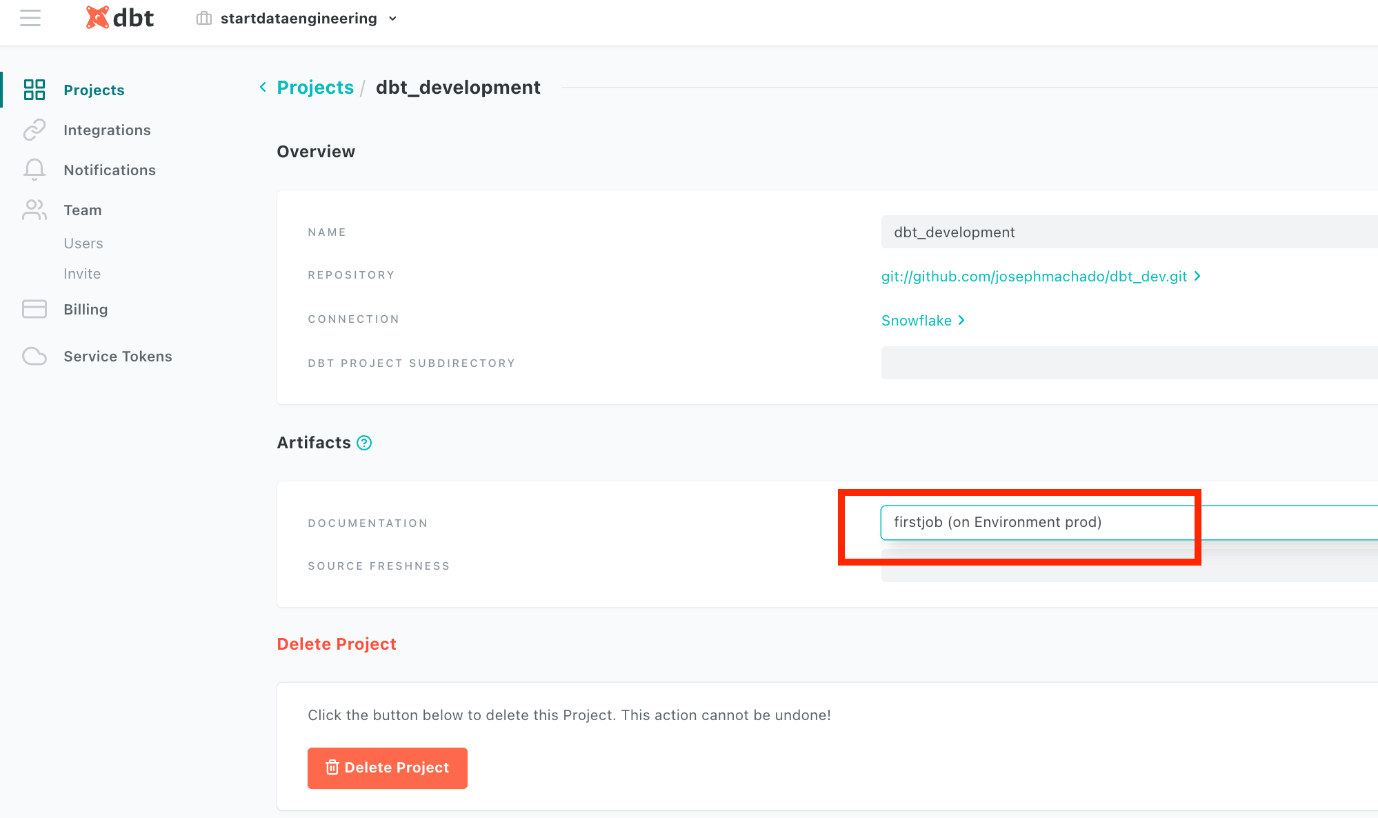
Let’s schedule a job to run all the models every day. You can go to the Job section and create a job as shown below.



Now dbt cloud will take care of running this job [every day as 12 AM](https://crontab.guru/#0_0_*_*_*) for you. Note that we also check the GENERATE DOCS checkbox.

**Host data documentation**

One of the most powerful features of dbt is its hosted documentation. Dbt allows us to specify column and table descriptions as part of your schema files. This documentation can be viewed by anyone with access to dbt cloud. We can generate documentation for any job from the project settings page. Click on hamburger icon -> Account Settings -> Project. On this page, select the job whose data documentation you want hosted by dbt cloud. Dbt cloud will show the documentation from the latest run of the selected job.



You can view the docs by clicking on hamburger icon -> Documentation.

**Conclusion and next steps**

Hope this article gave you a good understanding of how to setup an ELT data-ops workflow using dbt cloud and snowflake. There are more advanced features such as using [custom schema names based on environments](https://docs.getdbt.com/docs/building-a-dbt-project/building-models/using-custom-schemas#an-alternative-pattern-for-generating-schema-names) , [using variables](https://docs.getdbt.com/docs/building-a-dbt-project/building-models/using-variables) , [hooks for permission grants](https://docs.getdbt.com/reference/project-configs/on-run-start-on-run-end) , etc. These can be used when required by your project.

To recap, we saw

1. How to setup Snowflake permissions for dbt cloud
2. How to setup dbt cloud + github integration
3. How to enable CI jobs on github PRs(Pull requests)
4. How to schedule jobs from dbt cloud
5. How to host documentation in dbt cloud

The next time you are building an ELT data stack, consider using dbt cloud and Snowflake. These tools make managing data pipelines seamless and easy. As always, please feel free to leave any questions or comments in the comments section below.

**Further reading**

1. [Advantages of using dbt](https://www.startdataengineering.com/post/advantages-of-using-dbt-data-build-tool/)
2. [dbt tutorial](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/)
3. [ELT: stitch + dbt](https://www.startdataengineering.com/post/build-a-simple-data-engineering-platform/)
4. [ETL & ELT, a comparison](https://www.startdataengineering.com/post/elt-vs-etl/)
5. [What is a data warehouse](https://www.startdataengineering.com/post/what-is-a-data-warehouse/)
6. [Unit test in dbt](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/)

**References**

1. [Snowflake permission model](https://blog.getdbt.com/how-we-configure-snowflake/)
2. [Using correct branch, dbt](https://discourse.getdbt.com/t/dbt-cloud-how-to-rename-your-github-branch-from-master-to-main/2221)
3. [dbt CI docs](https://docs.getdbt.com/docs/dbt-cloud/using-dbt-cloud/cloud-enabling-continuous-integration-with-github)
4. [dbt dev branch name](https://discourse.getdbt.com/t/dbt-cloud-how-to-rename-your-github-branch-from-master-to-main/2221)

**How to unit test sql transforms in dbt**

Jan 16, 2021 · 7 min read

* [Introduction](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#introduction)
* [Setup](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#setup)
* [Code](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#code)
  + [Conditional logic to read from mock input](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#conditional-logic-to-read-from-mock-input)
  + [Custom macro to test for equality](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#custom-macro-to-test-for-equality)
  + [Setup environment specific test](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#setup-environment-specific-test)
  + [Run ELT using dbt](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#run-elt-using-dbt)
* [Conclusion](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#conclusion)
* [Further reading](https://www.startdataengineering.com/post/how-to-test-sql-using-dbt/#further-reading)

**Introduction**

With the recent advancements in data warehouses and tools like dbt most transformations(T of ELT) are being done directly in the data warehouse. While this provides a lot of functionality out of the box, it gets tricky when you want to test your sql code locally before deploying to production. If you have wondered

how do I implement unit test on SQL transforms?

How do I create the test-data to test SQL logic?

What is an easy and dependable way to test my SQL transforms before deploying my code?

then, this post is for you. In this post we build on top of a very popular open source tool called [dbt](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/) . We will see how we can run unit test against mock data locally. In software engineering, a unit test refers to testing a single piece of logic, be it encapsulated in a function or method.

In the context of sql testing, we defined a unit as a single model(which is a sql select statement in dbt). If you are unfamiliar with how dbt works, it is recommended to read [this dbt tutorial](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/) before proceeding.

**Setup**

If you would like to follow along you will need

1. [docker](https://docs.docker.com/get-docker/)
2. [git](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)
3. [dbt](https://docs.getdbt.com/dbt-cli/installation/)
4. [psql](https://stackoverflow.com/questions/44654216/correct-way-to-install-psql-without-full-postgres-on-macos)

We will build on a simple ELT pipeline using dbt and Postgres. In real projects you usually have a local environment where you develop and a stage and a production environment. Production environment will have the data used by the end users. For this example we will assume that

1. the local environment where you develop is called dev
2. the data used by end users are in the prod environment.

**Code**

You can clone the repository, setup a dev and prod data warehouse using the setup\_script.sh as shown below.

git clone https://github.com/josephmachado/unit\_test\_dbt.git && cd unit\_test\_dbt

chmod u+rwx setup\_script.sh tear\_down\_script.sh run\_dbt.sh # permissions to execute script

export PGPASSWORD=password # set password as env variable to not have to type it again

./setup\_script.sh

The setup\_script.sh creates dev and prod data warehouses, creates startdataengg.raw\_greeting\_events table in both environments and loads some data for that table in prod.

This ELT pipeline transforms a base\_greeting\_events table that is already present in the data warehouse into a usable fct\_greeting\_events table.



Let’s say we want to test if the sql transformation from base\_greeting\_events to fct\_greeting\_events is correct. dbt tests for data correctness after the data has been processed and this would be not ideal in a production environment. We want to test for logic correctness locally using a mock input and comparing it against an expected mock output.

We can do this using

1. A conditional logic that uses mock input instead of data from base\_greeting\_events only in dev environment.
2. Writing a custom macro to check if two sets of data are the same. In our case we will use this to check the mock inputs and mock outputs equality.

**Conditional logic to read from mock input**

In the model file at models/staging/stg\_greeting\_events.sql you can see that we have a logic which selects the input model based on the environment.

with source as (

{% if target.name == 'dev' %}

select \* from {{ ref('input\_base\_greeting\_events') }}

{% else %}

select \* from {{ ref('base\_greeting\_events') }}

{% endif %}

),

--select statement

When running dbt in dev environment we use the model input\_base\_greeting\_events which is the mock input present in the location data/input\_base\_greeting\_events.csv. In any other environment we use the table base\_greeting\_events.

**Custom macro to test for equality**

We can write custom sql queries and reuse them across our project. These custom queries are called macros. There are 2 main types of tests in dbt, they are

1. **Schema tests**: Queries which return the number 0 to pass else fails.
2. **Data test**: Queries that return 0 records to pass else fails.

We will write a custom Schema test which returns 0 when the models being compared are the same. You can see our macro in the location macros/test\_equality.sql.

{% macro test\_equality(model) %} -- macro definition

{% set compare\_model = kwargs.get('compare\_model') %} -- get compare\_model input parameter

{% set env = kwargs.get('env') %} -- get env input parameter

{%- if target.name == env -%} -- check if env input parameter matches the current environment

select count(\*) from ((select \* from {{ model }} except select \* from {{ compare\_model }} ) union (select \* from {{ compare\_model }} except select \* from {{ model }} )) tmp

{%- else -%}

select 0 -- if no input or different env return true

{%- endif -%}

{% endmacro %}

The [jinja templates](https://jinja.palletsprojects.com/en/2.11.x/) are a way to write conditional, loop and other control flow logic into sql. In the above script we

1. Define a macro.
2. Get compare\_model and env input parameters.
3. If env parameter is the current environment run the equality check sql query and return 0 or >0 which translates to true or false respectively.
4. If not return true.

The equality sql script is a query to check if the models being compared are the same. The mock output is defined at data/expected\_transformed\_greeting\_events.csv. The parsed sql script is shown below.

select count(\*)

from (

(

select \*

from devWarehouse.startdataengg.fct\_greeting\_events

except

select \*

from devWarehouse.startdataengg.expected\_transformed\_greeting\_events

)

union

(

select \*

from devWarehouse.startdataengg.expected\_transformed\_greeting\_events

except

select \*

from devWarehouse.startdataengg.fct\_greeting\_events

)

) tmp

**Setup environment specific test**

In dbt the test are run after the models are materialized. The tests are defined in a .yml file. The equality test is defined at models/marts/schema.yml.

version: 2

models:

- name: fct\_greeting\_events

# other tests

tests:

- unit\_test\_dbt.equality:

compare\_model: ref('expected\_transformed\_greeting\_events')

env: dev

From the above snippet you can see that the equality macro is called with the input parameters compare\_model: ref('expected\_transformed\_greeting\_events') and env: dev.

**Run ELT using dbt**

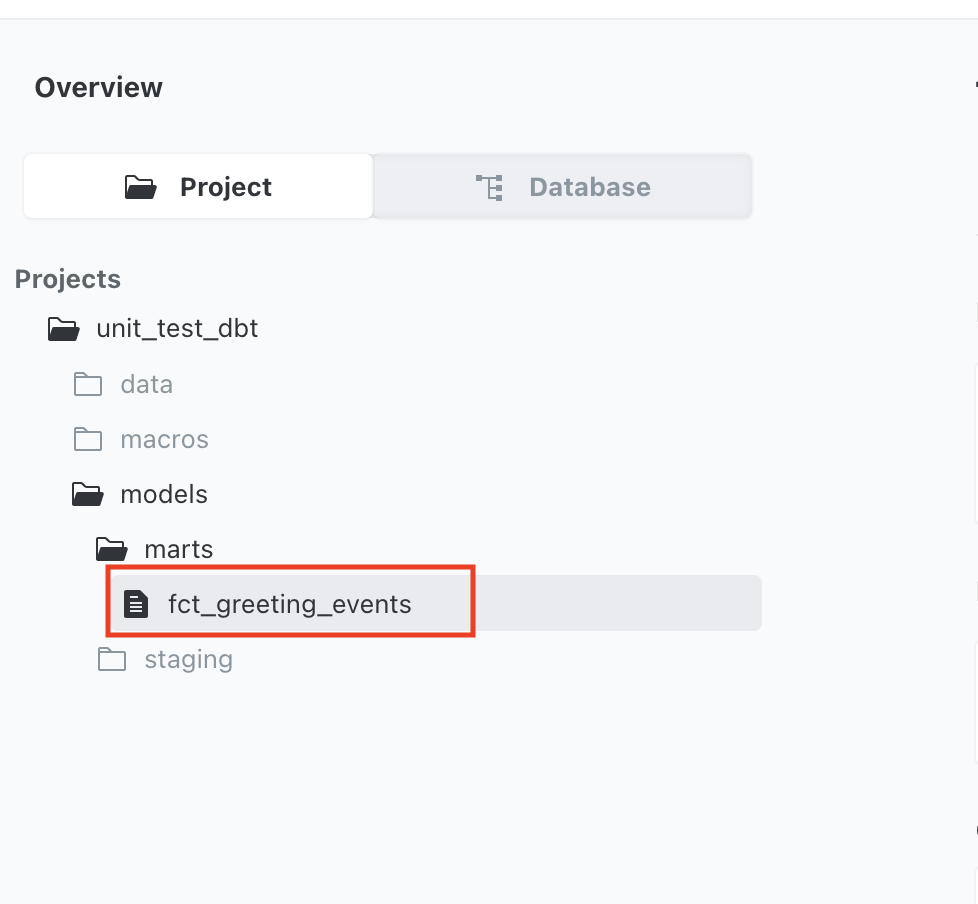
You can run dbt for dev environment by running the script.

./run\_dbt.sh

If you look at the run\_dbt.sh script you can see that we run the following commands

1. **debug**: to check if the connections work.
2. **seed**: we use this to load in mock data from the data folder. We only run this for dev environment.
3. **run**: to run the transformations.
4. **test**: to run the tests defined in schema.yml files.
5. **docs generate**: to generate documentation for the UI.
6. **docs serve**: start web-server to view the documentation and compiled sql scripts.

In the UI, if you go to fct\_greeting\_events and click on the data lineage icon on the bottom right, you will see that the base dataset used is input\_base\_greeting\_events which is the mock input present in the location data/input\_base\_greeting\_events.csv.



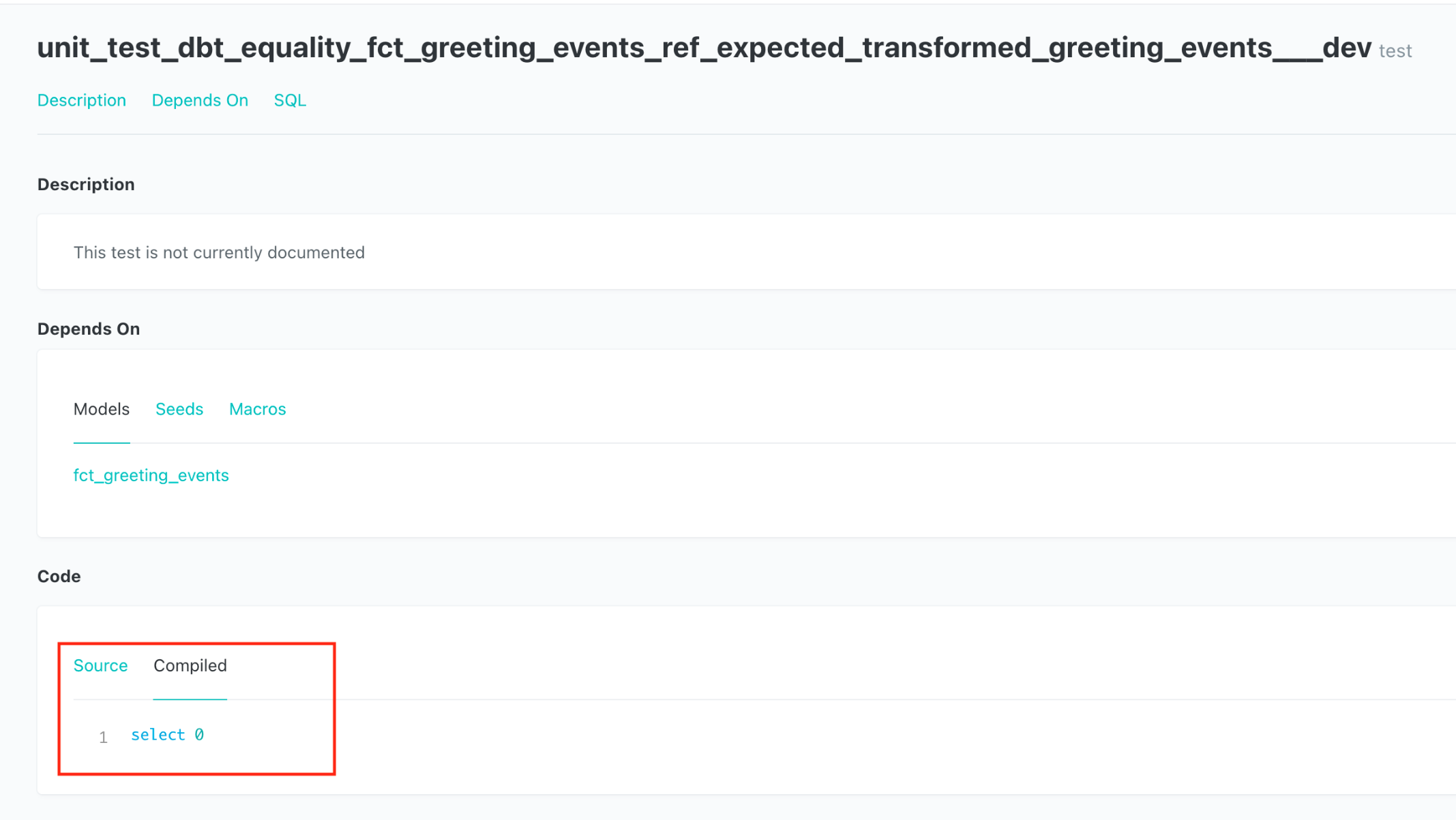
If you click on the unit\_test\_dbt\_equality\_fct\_greeting\_events\_ref\_expected\_transformed\_greeting\_events\_\_\_dev test in the fct\_greeting\_events UI, you can see the compiled sql used to check for equality.



Use ctrl+c to stop the web-server. You can run dbt in prod environment using

./run\_dbt.sh prod

You will see that the base dataset used is base\_greeting\_events and the equality test sql is just select 0.



You can also use pre built packages available for dbt. One of the popular ones is [dbt utils](https://hub.getdbt.com/fishtown-analytics/dbt_utils/latest/" \t "_blank) , this package also has a equality test with better functionality, but is not env specific.

You can tear down the docker containers using

./tear\_down\_script.sh

**Conclusion**

Hope this article gives you a good idea of how to get started with **unit testing sql transforms in dbt**. This will help keep development cycles shorter and prevents unintended modification of production data. The next time you are writing a ELT pipeline in dbt consider writing a unit test case to test the sql script locally before deploying.

There are multiple approaches for unit testing sql, that you will encounter in the wild. A good approach would be to start with something small and evolve the testing pattern as your data testing needs and team size grows.

Please leave any questions or comments in the comment section below.

**Further reading**

1. [dbt tutorial](https://www.startdataengineering.com/post/dbt-data-build-tool-tutorial/)
2. [advantages of dbt](https://www.startdataengineering.com/post/advantages-of-using-dbt-data-build-tool/)
3. [state of testing in dbt](https://discourse.getdbt.com/t/state-of-testing-in-dbt/1778/5)

**References:**

1. [dbt docs](https://docs.getdbt.com/)
2. [dbt discourse](https://discourse.getdbt.com/t/how-do-you-test-your-data/149)