Oh, my dbt (data build tool)

My experience and a couple of notes of using this superb tool for a month

Tomas Peluritis Jul 12 · 10 min read ★

Introduction

All my life, I was working with data. Somehow it sounds dramatic when I put it like that. Basically, I've done some analysis and basic work with SQL as a Business analyst, but nothing where I'd need templating. So-called BI career I started in 2013. Being a consultant and working mostly with MSSQL on multiple similar projects, it would have been a blessing to have something like dbt (or at least to know about Jinja at that time…); let's write it off as a lack of experience.

Funny that I tried dbt only now. If I'm honest with you — I've been using it for ~month, so keep in mind that I'm not a pro, just spreading the knowledge and sharing what I've found. You can find many other medium articles on some specifics or go straight to the source of dbt.

Prerequisites

First of all, for this to work, you'd need Docker. If you're not familiar with docker, I will promote my older blog post I wrote some time ago about it. When working in docker-created environments, I prefer to use VSCode with its dev container option, where it basically creates an isolated environment with all my configs, mounts, etc. If you make any changes to the existing docker image, you can choose the option rebuild image, and it will compose it and open it for you with all changes. Super handy if you're developing things so that you can skip manually doing docker-compose.

In my docker image, I've created a specific docker-compose file with two components — simple postgres:13-alpine and python 3.8. Choosing python 3.8.11 over 3.9 — had some issues trying to install dbt because of compatibility issues. I'm also using the mount option in my docker-compose file to pass the proper profiles.yml file for this specific project.

Postgres Dockerfile:

```
FROM postgres:13-alpine
ENV POSTGRES_PASSWORD=nopswd
ENV POSTGRES_DB db
COPY init.sql /docker-entrypoint-initdb.d/
```

In the **init.sql** file, I just created a database named **db**.

Python Dockerfile:

```
FROM python:3.8
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
```

Nothing fancy in requirements, just the dbt library.

> *If you have already a production environment with dbt and you're setting up a local one — always use the same dbt version as you have in production. Had trouble on dbt run, but my colleagues didn't. Root cause — everyone was using 0.19.0, and I installed the latest at that time 0.19.2 and some compatibility issues occurred for dbt deps we had in the packages.yml file.*

Docker-compose, as I mentioned, has some more things, but nothing fancy:

```
1  version: "3.8"
2  services:
3    db :
4      build: ./postgres/
5      ports:
```

```
    ports:
      - 5432:5432
  python:
    build: ./python/
    ports:
      - 8001:8001
    volumes:
      - ./.dbt/profiles.yml:/root/.dbt/profiles.yml
    depends_on:
      - db
    links:
        - "db:database"
```

You might be wondering why I'm opening the 8001 port — it's needed for some dbt feature you'll see later on.

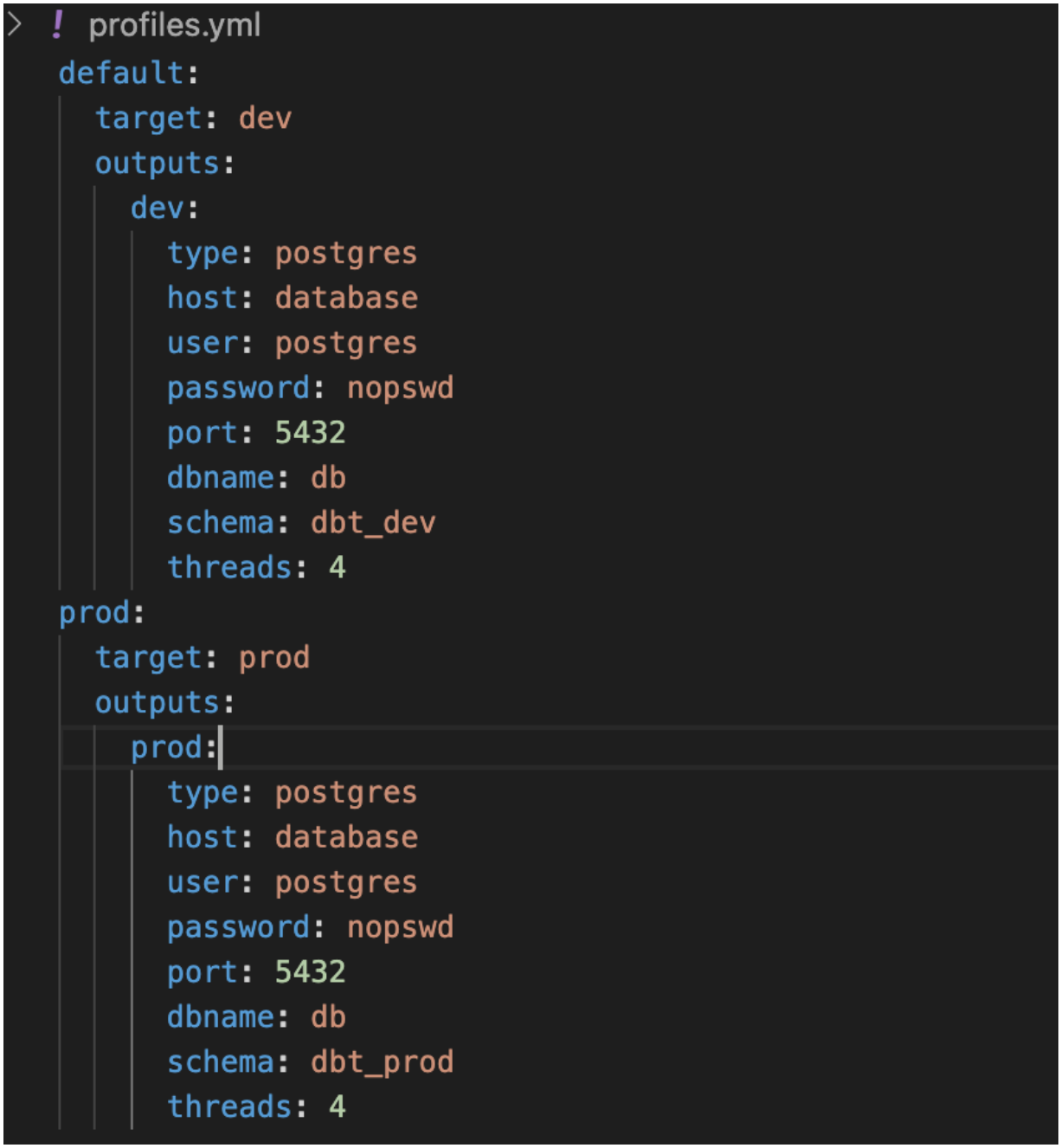
Getting started with dbt

Ok, what is this dbt, you might be wondering. Basically, it's an amazing tool to ease your transformation part in your ELT flow give you data lineage, documentation, and full control on data refreshes if some underlying data changes in one of the models somewhere in the middle. I really don't want (and usually don't like) to go to product details since I'm a more technical person, not a product one.

Ok, so there are a couple of important files in dbt.

- profiles.yml — file where you set up all connections and how you're going to use them
- dbt-project.yml — specific configuration for specific dbt project you have this file in.

Let's go over the profiles.yml file:



We have to have a default profile; this will be where everything is run if nothing else is specified. Different profiles will allow you to easily test pipelines on different environments (i.e., test and prod):

```
# Running on default:
dbt run


# Running on prod:
dbt run --profile prod


# Running on default with specified profile:
dbt run --profile default
```

After playing around in VSCode opening my folder in the development container, it's interesting to see if all works as intended.

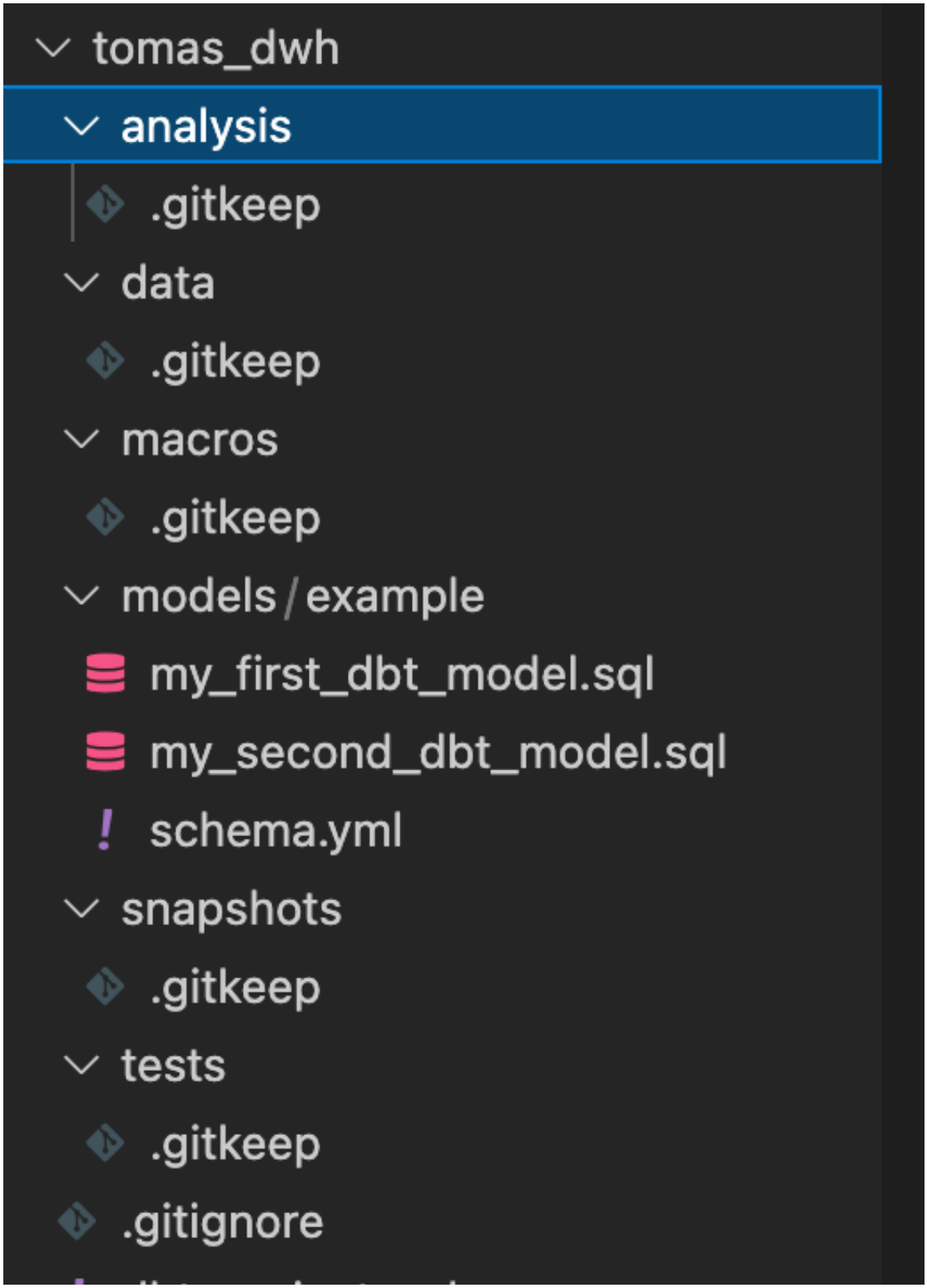
```
root@866eec0e9c16:/workspace# dbt debug
Running with dbt=0.19.2
dbt version: 0.19.2
python version: 3.8.11
python path: /usr/local/bin/python
os info: Linux-5.10.25-linuxkit-x86_64-with-glibc2.2.5
Using profiles.yml file at /root/.dbt/profiles.yml
Using dbt_project.yml file at /workspace/dbt_project.yml

Configuration:
  profiles.yml file [OK found and valid]
  dbt_project.yml file [ERROR not found]

Required dependencies:
 - git [OK found]

Could not load dbt_project.yml
```

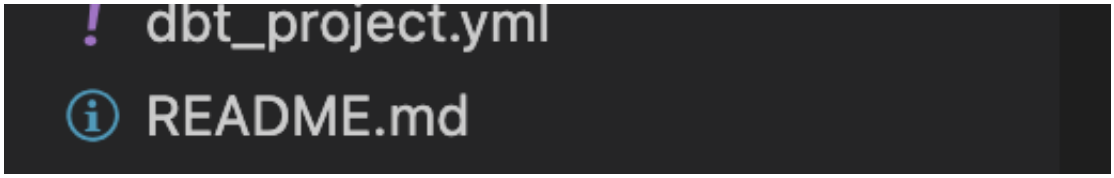
As you see, we have one error on dbt_project.yml. Let's fix it.

For simplicity and keeping the original dbt structure, we can initialize it. To do this, let's run this command:

```
dbt init MY_DBT_PROJECT_NAME
```

Now we can see what's the structure dbt expects us and how it works with:





Initialized dbt project structure. Image by Author

Let's check if everything else is working from this folder and properly created profiles.yml

```
root@8f8a9b0625d1:/workspace# cd tomas_dwh
root@8f8a9b0625d1:/workspace/tomas_dwh# dbt debug
Running with dbt=0.19.2
dbt version: 0.19.2
python version: 3.8.11
python path: /usr/local/bin/python
os info: Linux-5.10.25-linuxkit-x86_64-with-glibc2.2.5
Using profiles.yml file at /root/.dbt/profiles.yml
Using dbt_project.yml file at /workspace/tomas_dwh/dbt_project.yml

Configuration:
  profiles.yml file [OK found and valid]
  dbt_project.yml file [OK found and valid]

Required dependencies:
 - git [OK found]

Connection:
  host: database
  port: 5432
  user: postgres
  database: db
  schema: dbt_dev
  search_path: None
  keepalives_idle: 0
  sslmode: None
  Connection test: OK connection ok
```

dbt debug results. Image by Author

Great success! Our environment is fully functional and ready for us to check all things out.

Let's try dbt run on the default profile:

```
root@8f8a9b0625d1:/workspace/tomas_dwh# dbt run --profile default
Running with dbt=0.19.2
Found 2 models, 4 tests, 0 snapshots, 0 analyses, 138 macros, 0 operations, 0 seed files, 0 sources, 0 exposures

20:06:21 | Concurrency: 4 threads (target='dev')
20:06:21 |
20:06:21 | 1 of 2 START table model dbt_dev.my_first_dbt_model..................... [RUN]
20:06:21 | 1 of 2 OK created table model dbt_dev.my_first_dbt_model.............. [SELECT 2 in 0.12s]
20:06:21 | 2 of 2 START view model dbt_dev.my_second_dbt_model..................... [RUN]
20:06:21 | 2 of 2 OK created view model dbt_dev.my_second_dbt_model.............. [CREATE VIEW in 0.08s]
20:06:21 |
20:06:21 | Finished running 1 table model, 1 view model in 0.34s.

Completed successfully

Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

dbt run on the default profile. Image by Author

We see that we have two models (which correspond to two files named my_first_dbt_model.sql and my_second_dbt_model.sql), but what are these tests? Where do they come from? Let's dig deeper into the model's folder.

We can see we have schema.yml file with contents

```yaml
version: 2

models:
    - name: my_first_dbt_model
      description: "A starter dbt model"
      columns:
          - name: id
            description: "The primary key for this table"
            tests:
                - unique
                - not_null

    - name: my_second_dbt_model
      description: "A starter dbt model"
      columns:
          - name: id
            description: "The primary key for this table"
            tests:
                - unique
                - not_null
```

Schema.yml file. Image by Author

We can see that we have two columns described plus tests — a column has to be unique and not null.

I found that my colleagues are creating a yml file per each model. In my opinion, this is a better option:

- visually looks more clear
- no merge conflicts because, most likely, there will be one developer per one model!

If we'd look at the queries their straightforward. Creates a table with 1 and null, creates a view out of the first table where id = 1. But wait — our tests didn't say that we failed. We have a null value! That's because it doesn't have any data to test upon. So after we run our model, we need to test it.

To run tests:

```
dbt test --model example
```

Output in the console will look like this:

```
root@8f8a9b0625d1:/workspace/tomas_dwh# dbt test --model example
Running with dbt=0.19.2
Found 2 models, 4 tests, 0 snapshots, 0 analyses, 138 macros, 0 operations, 0 seed files, 0 sources, 0 exposures

20:18:43 | Concurrency: 4 threads (target='dev')
20:18:43 |
```

```
20:18:43 | 1 of 4 START test not_null_my_first_dbt_model_id..... [RUN]
20:18:43 | 2 of 4 START test not_null_my_second_dbt_model_id..... [RUN]
20:18:43 | 3 of 4 START test unique_my_first_dbt_model_id..... [RUN]
20:18:43 | 4 of 4 START test unique_my_second_dbt_model_id..... [RUN]
20:18:44 | 3 of 4 PASS unique_my_first_dbt_model_id..... [PASS in 0.11s]
20:18:44 | 2 of 4 PASS not_null_my_second_dbt_model_id..... [PASS in 0.11s]
20:18:44 | 1 of 4 FAIL 1 not_null_my_first_dbt_model_id..... [FAIL 1 in 0.13s]
20:18:44 | 4 of 4 PASS unique_my_second_dbt_model_id..... [PASS in 0.13s]
20:18:44 |
20:18:44 | Finished running 4 tests in 0.25s.

Completed with 1 error and 0 warnings:

Failure in test not_null_my_first_dbt_model_id (models/example/schema.yml)
  Got 1 result, expected 0.

  compiled SQL at target/compiled/my_new_project/models/example/schema.yml/schema_test/not_null_my_first_dbt_model_id.sql
```

Failed test. Image by Author

Clearly, we can see that there are some issues on our end and, we need to fix them.

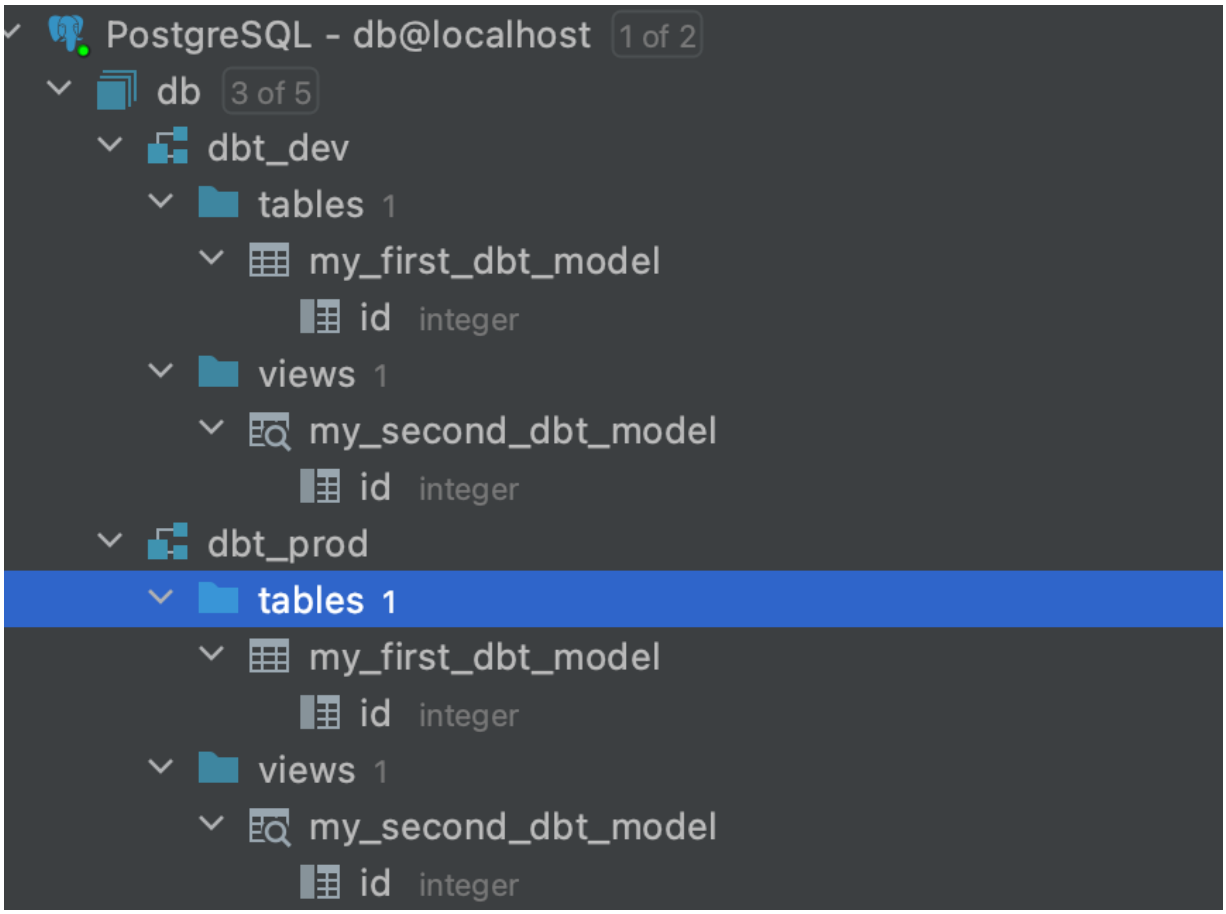
The fix is easy. Let's switch from null to some number and test again. We'd still see the same state if we'd run directly "dbt test" after the fix. We didn't run the model, so underlying data didn't change. We need to run and test it.

```
root@8f8a9b0625d1:/workspace/tomas_dwh# dbt run
Running with dbt=0.19.2
Found 2 models, 4 tests, 0 snapshots, 0 analyses, 138 macros, 0 operations, 0 seed files, 0 sources, 0 exposures

20:21:20 | Concurrency: 4 threads (target='dev')
20:21:20 |
20:21:20 | 1 of 2 START table model dbt_dev.my_first_dbt_model..... [RUN]
20:21:20 | 1 of 2 OK created table model dbt_dev.my_first_dbt_model..... [SELECT 2 in 0.13s]
20:21:20 | 2 of 2 START view model dbt_dev.my_second_dbt_model..... [RUN]
20:21:20 | 2 of 2 OK created view model dbt_dev.my_second_dbt_model..... [CREATE VIEW in 0.08s]
20:21:20 |
20:21:20 | Finished running 1 table model, 1 view model in 0.38s.

Completed successfully

Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
root@8f8a9b0625d1:/workspace/tomas_dwh# dbt test
Running with dbt=0.19.2
Found 2 models, 4 tests, 0 snapshots, 0 analyses, 138 macros, 0 operations, 0 seed files, 0 sources, 0 exposures

20:22:13 | Concurrency: 4 threads (target='dev')
20:22:13 |
20:22:13 | 1 of 4 START test not_null_my_first_dbt_model_id..... [RUN]
20:22:13 | 2 of 4 START test not_null_my_second_dbt_model_id..... [RUN]
20:22:13 | 3 of 4 START test unique_my_first_dbt_model_id..... [RUN]
20:22:13 | 4 of 4 START test unique_my_second_dbt_model_id..... [RUN]
20:22:13 | 1 of 4 PASS not_null_my_first_dbt_model_id..... [PASS in 0.15s]
20:22:13 | 3 of 4 PASS unique_my_first_dbt_model_id..... [PASS in 0.15s]
20:22:13 | 4 of 4 PASS unique_my_second_dbt_model_id..... [PASS in 0.16s]
20:22:13 | 2 of 4 PASS not_null_my_second_dbt_model_id..... [PASS in 0.18s]
20:22:13 |
20:22:13 | Finished running 4 tests in 0.31s.

Completed successfully

Done. PASS=4 WARN=0 ERROR=0 SKIP=0 TOTAL=4
```

dbt run and dbt test view in the terminal. Image by Author

Hooray, we just fixed and ran our models successfully!

If we'd run dbt run on both dev/default and prod, we'd see in DB all of this

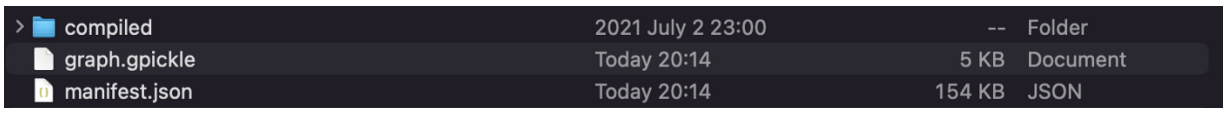


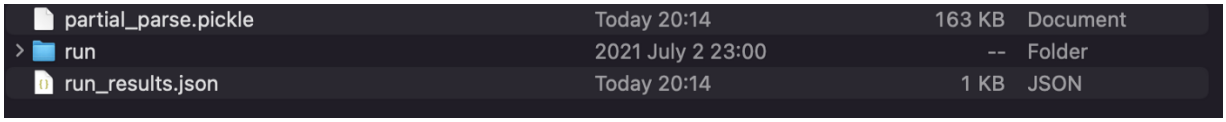
DB view after dbt run on default and prod. Image by Author

dbt specifics

Target folder

After our **dbt run**, we had this folder created. Its contents:





Target folder structure. Image by Author

For me, the interesting files are in the **compiled/run** directory. If we go down the rabbit hole, we can find our SQL queries parsed.

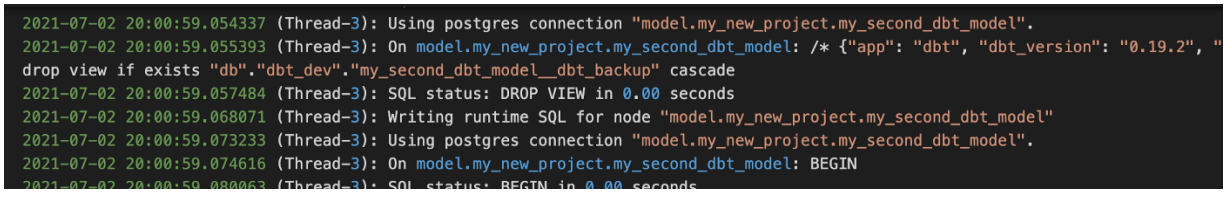
We also could compile our files by executing:

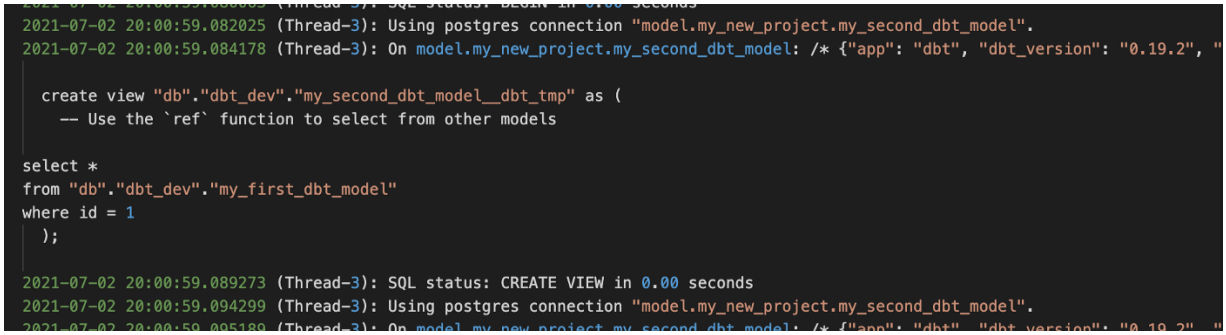
```
dbt compile
```

Run would create or update files in **compiled** and **run** folders. You'll have tests SQL compiled as well, so you can understand what was being run in your specified tests.

Logs

If any issues occur and it's not quite clear what it is - check logs/dbt.log. i.e., At work, I got "Database Error: permission denied for database X." I have no clue what permissions I was lacking. I got a link to debugging page of dbt, and my colleague said to check the logs. From there, I found what permissions I was missing.





A snippet of logs file. Image by Author

Incremental model

Let's imagine we have a situation where our data residing in DB is big, and we want to add incremental load. Generically we'd do one script if a table exists — create it from scratch, else — insert and (or) update it. So basically, we have repetitive parts of code, and we have to maintain it in two places. It doesn't comply with DRY (Don't Repeat Yourself). Luckily dbt has an amazing feature like an incremental load. For this, we're going to create an additional source table using Mockaroo. I've executed 01_mock_users_data.sql on my local Postgres database. I also made a small change and converted the created_at column to be a timestamp column instead of a date.

Created a simple model to use is_incremental macro:

```
{{
    config(
        materialized='incremental'
    )
}}

select  * from {{ source('ops_db', 'mock_users_data') }}
{% if is_incremental() %}

```

```
10    -- this filter will only be applied on an incremental run
11    where created_at >= (select max(created_at) from {{ this }})
12
```

If we'd run it now and check target/run:

```
create  table "db"."dbt_dev"."mock_users"
as (
select  * from "db"."operational_db"."mock_users_data"
);
```

Let's run **02_more_mock_users_data.sql** and do dbt run again. In target/run, we can see different outputs!

```
select  * from "db"."operational_db"."mock_users_data"
-- this filter will only be applied on an incremental run
where created_at >= (select max(created_at) from
"db"."dbt_dev"."mock_users")
```

Though nuance here that it will run exactly by filters you specified. The first run will be for ALL history; the next run will be for only new rows. The initial query might not even finish or encounter some other issues along the way (timeout, some hard limits on query run time, etc.). So you could go around and create an upper bound filter where you'd take only a couple of days/weeks/month and easily refresh it like this in several batches. Though it's tedious, and you'd have to run it manually to catch it up.

Macros + insert_by_period

> ***Disclaimer:*** *insert_by_period works with Redshift only, dbt-vault created vault_insert_by_period works on Snowflake. So basically, I'm just explaining my journey what I tried and checked along the way.*

I mentioned in incremental load "Macros," you might wonder what it is? It's some custom code, which is executed to add some missing functionality or more complex logic. I.e., mentioned before a tedious incremental load. In our case is a simple conditional insert that would load our initial data in multiple batches. You can check it out in the original discussion about this macro here. All in all, it's bundled in the dbt-utils package. We can import by specifying it in the packages.yml file. Version 0.7.0 wasn't compatible with my dbt version of 0.19.2 (asked for 0.20, which is only a release candidate at the moment this blog post was being written), so I used 0.6.4.

```
tomas_dwh > ! packages.yml
1   packages:
2     - package: fishtown-analytics/dbt_utils
3       version: 0.6.4
```

packages.yml content. Image by Author

and we can install dependencies with

```
dbt deps
```

If we'd follow all the information for the version for our Postgres use case, it won't work, since as it's written in the comments - it's suited for redshift only! After this, I went into the rabbit hole, checking dbt-vault, making some adjustments, and creating my own macro using comments in GitHub. But I guess I'm too new to macros, an advanced topic, and I couldn't make it work. I will have to dig deep on this later.

Snapshot model

The name of it doesn't really explain what it does. At least to me, a snapshot means the current state of the data. Though in the dbt case, if we create a snapshot model (they suggest putting it in the "snapshots" folder), we will have SCD type 2 (by the way, I wrote an article on SCD2 on spark some time ago, which covers what's an SCD).

So let's use the same mocked users data for this example. Let's add the updated_at column and make it match to created_at column (03_update_at.sql). Let's follow the basic example from dbt docs and run **dbt snapshot**. We can see how the snapshot looks like (only interested in newly added columns):

created_at	updated_at	dbt_scd_id	dbt_updated_at	dbt_valid_from	dbt_valid_to
2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	5ecda49f07401ecb61301cfe9ab4da5b	2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	<null>
2021-01-02 00:00:00.000000	2021-01-02 00:00:00.000000	5cb5c6f39a2cb5711cb981bd74955380	2021-01-02 00:00:00.000000	2021-01-02 00:00:00.000000	<null>
2021-01-02 00:00:00.000000	2021-01-02 00:00:00.000000	59fde6cd23d80b184c9d3091e777455b	2021-01-02 00:00:00.000000	2021-01-02 00:00:00.000000	<null>
2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	7eb40838336a413d335ad5e9ea04c51a	2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	<null>
2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	abb7995229cc3a96786ad9e14d5bd282	2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	<null>
2021-01-05 00:00:00.000000	2021-01-05 00:00:00.000000	f4fd903ede2e88efeaa1423b92b8fe99	2021-01-05 00:00:00.000000	2021-01-05 00:00:00.000000	<null>
2021-01-03 00:00:00.000000	2021-01-03 00:00:00.000000	97c773766a20e9632151c8eec085c6ed	2021-01-03 00:00:00.000000	2021-01-03 00:00:00.000000	<null>
2021-01-04 00:00:00.000000	2021-01-04 00:00:00.000000	584e01ed90d4d162640615bafb53065d	2021-01-04 00:00:00.000000	2021-01-04 00:00:00.000000	<null>
2021-01-05 00:00:00.000000	2021-01-05 00:00:00.000000	a7363114de512ecf23007b8896abe1b9	2021-01-05 00:00:00.000000	2021-01-05 00:00:00.000000	<null>

2021-01-06 00:00:00.000000	2021-01-06 00:00:00.000000	ea0ac93010ae258c74350fc6ca12b41d	2021-01-06 00:00:00.000000	2021-01-06 00:00:00.000000	<null>
2021-01-02 00:00:00.000000	2021-01-02 00:00:00.000000	bd97112767e942b0e17d90f612bbf964	2021-01-02 00:00:00.000000	2021-01-02 00:00:00.000000	<null>
2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	8a5ad5bd3761d73b78f76377008bedff	2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	<null>
2021-01-04 00:00:00.000000	2021-01-04 00:00:00.000000	afde881d5067422fb81ec6350eb1bde4	2021-01-04 00:00:00.000000	2021-01-04 00:00:00.000000	<null>
2021-01-06 00:00:00.000000	2021-01-06 00:00:00.000000	539bfb5e1d408be39eabe51ec787a0de	2021-01-06 00:00:00.000000	2021-01-06 00:00:00.000000	<null>

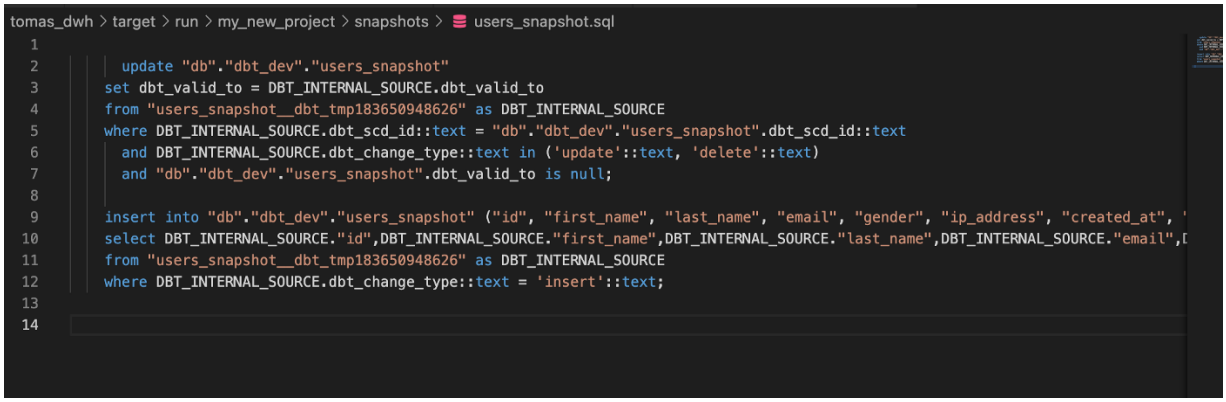
SCD type 2 of our mock users' data. Image by Author

We can see that we have **dbt_scd_id** and **dbt_valid_from** and **dbt_valid_to**, corresponding to the changes. Let's execute 04_change_some_names.sql and run **dbt snapshot.**

created_at	updated_at	dbt_scd_id	dbt_updated_at	dbt_valid_from	dbt_valid_to
2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	5ecda49f07401ecb61301cfe9ab4da5b	2021-01-01 00:00:00.000000	2021-01-01 00:00:00.000000	2021-07-11 18:36…
2021-01-01 00:00:00.000000	2021-07-11 18:36:23.275227	ea253e1964a096e826d393529700e7cc	2021-07-11 18:36:23.275227	2021-07-11 18:36:23.275227	<null>

SCD type 2. Image by author

Ok, so basically, we just set up what's unique, and dbt took care of the rest. Now that would have been handy many times for me. Looking in the target/run/snapshots folder, we can see our snapshot code was generated for us too!



tomas_dwh > target > run > my_new_project > snapshots > users_snapshot.sql

```
1
2       update "db"."dbt_dev"."users_snapshot"
3     set dbt_valid_to = DBT_INTERNAL_SOURCE.dbt_valid_to
4     from "users_snapshot__dbt_tmp183650948626" as DBT_INTERNAL_SOURCE
5     where DBT_INTERNAL_SOURCE.dbt_scd_id::text = "db"."dbt_dev"."users_snapshot".dbt_scd_id::text
6       and DBT_INTERNAL_SOURCE.dbt_change_type::text in ('update'::text, 'delete'::text)
7       and "db"."dbt_dev"."users_snapshot".dbt_valid_to is null;
8
9     insert into "db"."dbt_dev"."users_snapshot" ("id", "first_name", "last_name", "email", "gender", "ip_address", "created_at", "
10    select DBT_INTERNAL_SOURCE."id",DBT_INTERNAL_SOURCE."first_name",DBT_INTERNAL_SOURCE."last_name",DBT_INTERNAL_SOURCE."email",D
11    from "users_snapshot__dbt_tmp183650948626" as DBT_INTERNAL_SOURCE
12    where DBT_INTERNAL_SOURCE.dbt_change_type::text = 'insert'::text;
13
14
```

So basically, we can see that it created a temporary table and then made all comparisons for us!

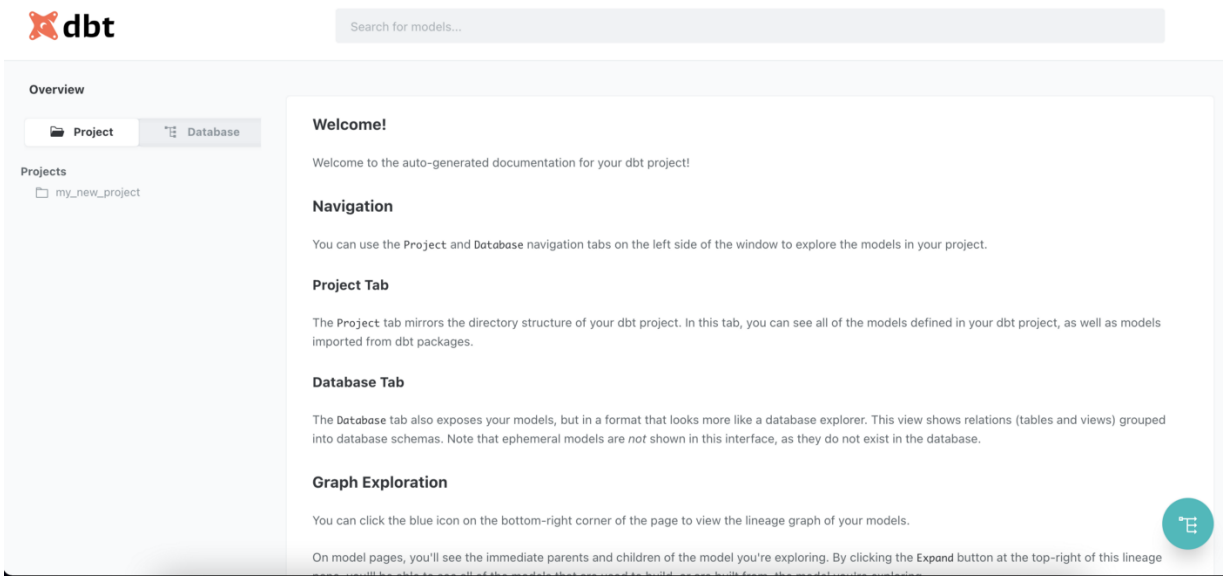
Generate docs

Data lineage and documentation. If you specified all relevant metadata in your yml files and used references to models and sources, you can generate documentation!

```
dbt docs generate
```

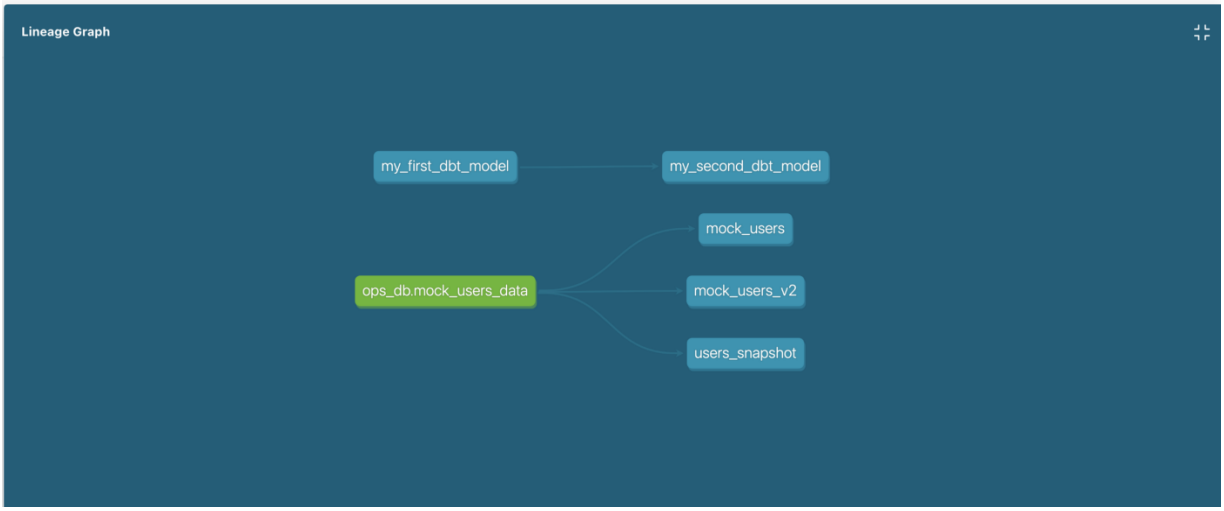
This method will generate a **catalog.json** file in your **target** directory. To check how it looks on the web:

```
dbt docs serve --port 8001 // or any other port you prefer
```



Documentation page generated by dbt. Image by author

If we'd click on the greenish icon bottom right, we'd see lineage!



Data lineage! Image by Author

Keep in mind that here I show basics. Tons of things are on the official dbt page (dbt-docs page)!

Summary

So we covered most of the basic things (I found out an area of interest -> macros). Strongly suggest to anyone who's working with the ELT approach to try out dbt. This will allow you to leverage it fully: full refreshes, downstream re-runs, documentation, and data lineage.

You can find my code in my GitHub repo.

Data Engineering | Data | Sql | Introduction | Pipeline As Code