Today's notebook

Resources

- List of dbt commands
- dbt node selection syntax

Accounts & Setup

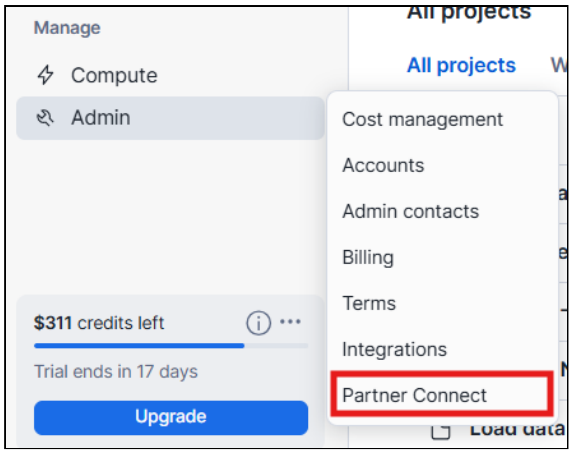
- [ ] Create a dbt trial account. (https://www.getdbt.com/signup)
- [ ] Create a Snowflake trial account (https://signup.snowflake.com/).
- [ ] Load data into snowflake

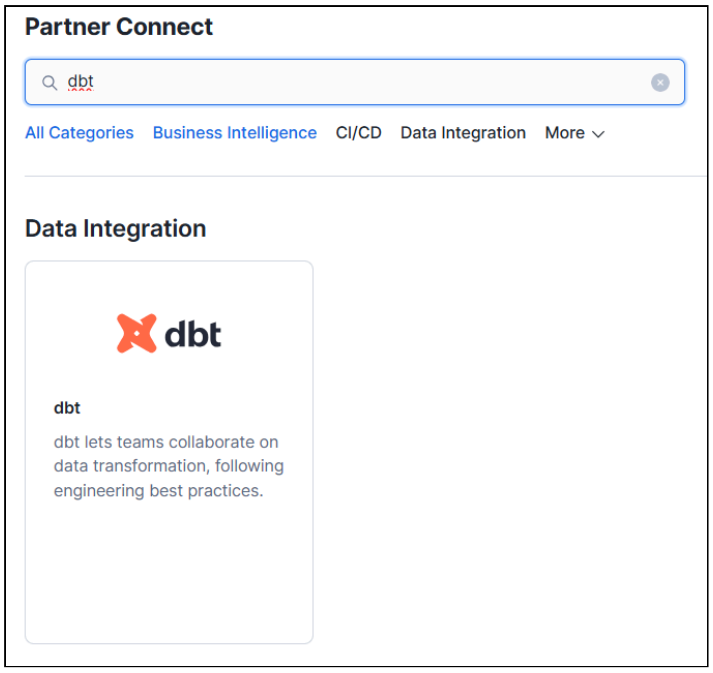
New worksheet:

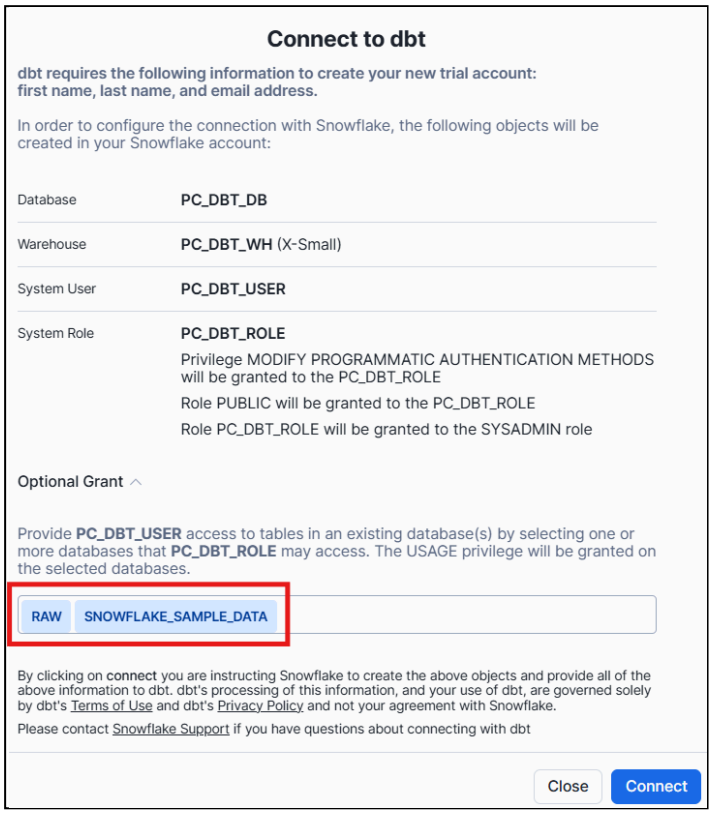
```
-- initial setup
use role sysadmin;
create database dbt_db;
create database raw;
create schema raw.jaffle_shop;
create schema raw.stripe;
create warehouse transforming;

-- customer table --
create table raw.jaffle_shop.customers
( id integer,
first_name varchar,
last_name varchar
);
-- load customer data
copy into raw.jaffle_shop.customers (id, first_name, last_name)
from 's3://dbt-tutorial-public/jaffle_shop_customers.csv'
file_format = (
    type = 'CSV'
    field_delimiter = ','
    skip_header = 1
    );

-- orders table --
create table raw.jaffle_shop.orders
( id integer,
user_id integer,
order_date date,
```

```
status varchar,
_etl_loaded_at timestamp default current_timestamp
);
-- load orders data --
copy into raw.jaffle_shop.orders (id, user_id, order_date, status)
from 's3://dbt-tutorial-public/jaffle_shop_orders.csv'
file_format = (
    type = 'CSV'
    field_delimiter = ','
    skip_header = 1
    );

-- payments table --
create table raw.stripe.payments
( id integer,
orderid integer,
paymentmethod varchar,
status varchar,
amount integer,
created date,
_batched_at timestamp default current_timestamp
);
-- load payments data --
copy into raw.stripe.payments (id, orderid, paymentmethod, status, amount,
from 's3://dbt-tutorial-public/stripe_payments.csv'
file_format = (
    type = 'CSV'
    field_delimiter = ','
    skip_header = 1
    );

-- checkout the tables
select * from raw.jaffle_shop.customers;
select * from raw.jaffle_shop.orders;
select * from raw.stripe.payments;
```

- [ ] Connect Snowflake & dbt.
    - Using **Partner Connect** (Recomended):


Manage
Compute
Admin
All projects
All projects
Cost management
Accounts
Admin contacts
Billing
Terms
Integrations
Partner Connect
$311 credits left
Trial ends in 17 days
Upgrade


Partner Connect
dbt
All Categories
Business Intelligence
CI/CD
Data Integration
More
Data Integration
dbt
dbt
dbt lets teams collaborate on data transformation, following engineering best practices.

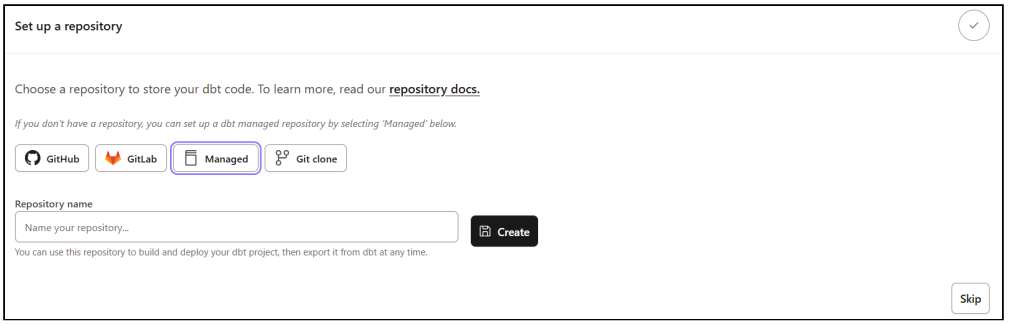

Connect to dbt
dbt requires the following information to create your new trial account: first name, last name, and email address.
In order to configure the connection with Snowflake, the following objects will be created in your Snowflake account:
Database
PC_DBT_DB
Warehouse
PC_DBT_WH (X-Small)
System User
PC_DBT_USER
System Role
PC_DBT_ROLE
Privilege MODIFY PROGRAMMATIC AUTHENTICATION METHODS will be granted to the PC_DBT_ROLE
Role PUBLIC will be granted to the PC_DBT_ROLE
Role PC_DBT_ROLE will be granted to the SYSADMIN role
Optional Grant
Provide PC_DBT_USER access to tables in an existing database(s) by selecting one or more databases that PC_DBT_ROLE may access. The USAGE privilege will be granted on the selected databases.
RAW
SNOWFLAKE_SAMPLE_DATA
By clicking on connect you are instructing Snowflake to create the above objects and provide all of the above information to dbt. dbt's processing of this information, and your use of dbt, are governed solely by dbt's Terms of Use and dbt's Privacy Policy and not your agreement with Snowflake.
Please contact Snowflake Support if you have questions about connecting with dbt
Close
Connect

- **Manual connection**:

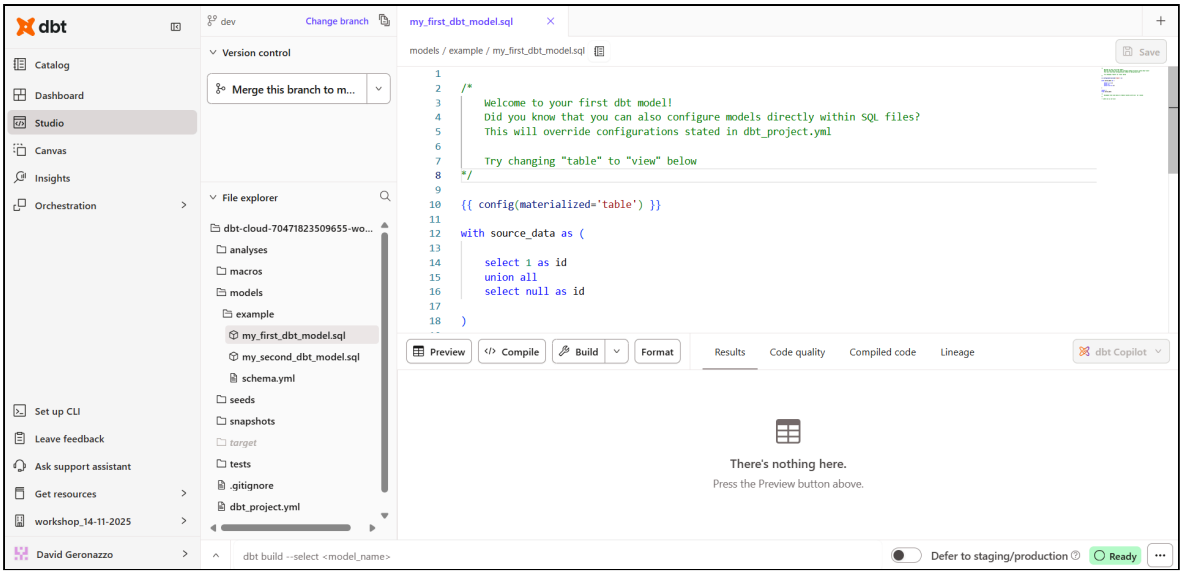
```
use role accountadmin;
create role dbt_transformer;
set name = (select current_user());
grant role dbt_transformer to user identifier($name);

grant all on database raw to role dbt_transformer;
grant all on database dbt_db to role dbt_transformer;

grant all on schema raw.jaffle_shop to role dbt_transformer;
grant all on schema raw.stripe to role dbt_transformer;

grant all on all tables in database raw to role dbt_transformer;
grant all on future tables in database raw to role dbt_transformer;
```

Go to dbt *account settings*, create a **new project**, which require a *development environment* and so a *connection*.

1. create a new project.
2. setup a new connection: chose **Snowflake**
   Enter: ACCOUNT - ROLE (leave black) - DATABASE `dbt_db` - WAREHOUSE `transforming`.
3. setup the development environment:
   Chose **Username and Password**;
   Enter: USERNAME - PASSWORD - ROLE `dbt_transformer` - SCHEMA `dbt_<first-initial><last-name>`
4. test connection and save.
5. Setup a **managed** repository, and give it a name.



Interface



Models

What are Models?

Models are *select statement* (sql files):

- modular steps of logics between our raw layer and the trnasformed layer.
- Each model maps to a table/view in the data platform.

Look at the `models/example` folder:

- 2 models (sql file)
- 1 schema (yml file)

-> We can `compile` the code and `preview` the data.

In order to refer to other models we use the **`ref`** macro: `select * from {{ref('my_model')}}`

Let's create a new model.

- [ ] Create a new file inside: `models/customers.sql`

```
with customers as (
    select
        id as customer_id,
        first_name,
        last_name
    from raw.jaffle_shop.customers

),
```

```
orders as (
    select
        id as order_id,
        user_id as customer_id,
        order_date,
        status
    from raw.jaffle_shop.orders

),

customer_orders as (
    select
        customer_id,
        min(order_date) as first_order_date,
        max(order_date) as most_recent_order_date,
        count(order_id) as number_of_orders,
        sum(amount) as lifetime_value
    from orders
    group by 1

),

final as (
    select
        customers.customer_id,
        customers.first_name,
        customers.last_name,
        customer_orders.first_order_date,
        customer_orders.most_recent_order_date,
        coalesce(customer_orders.number_of_orders, 0) as number_of_orders,
        customer_orders.lifetime_value
    from customers
    left join customer_orders using (customer_id)
)

select * from final;
```

How to create a model in the data platform?

The `dbt run` command.

- create a spcific model: `dbt run --select customers`
- create all models inside a folder: `dbt run --select example`

- create all models: `dbt run`

- [ ] Create the *customers* model.

- [ ] Take a look in Snowflake.

Materialization (view or table?)

dbt, by default, always creates models as views. But we can declare the materialization of a model in different ways: at a **project level**, or at **model level**.
The model level overwrites the project level.

Look at the examples models:

- config blocks in the models
- config in the `example/schema.yml` file
- project config in the `dbt_project.yml` file

- [ ] Materiliaze the *customers* model as a table.

- [ ] Take a look in snowflake

Lineage

Now, lets create our staging models (1-1 with our source tables):

- [ ] Create `stg_jaffle_shop__customers.sql`

```
with source as (
    select
        id as customer_id,
        first_name,
        last_name
    from raw.jaffle_shop.customers
)

select * from final
```

- [ ] create `stg_jaffle_shop__orders.sql`

```
with source as (
    select
        id as order_id,
        user_id as customer_id,
        order_date,
```

```
        status
    from raw.jaffle_shop.orders
)

select * from final
```

- [ ] Use the `ref` macro to refer these in the *customer* model

Sources

In dbt we configure our sources using `.yml` files. This is the structure of dbt models configurations for example.
YML files are human readable configuration files used to structure and document the models/objects inside the project.

- [ ] Let's configure our sources.

- [ ] Create a .yml file `models/sources.yml` to configure our sources.

```
sources:
    - name: jaffle_shop
      database: raw
      schema: jaffle_shop
      tables:
      - name: orders
      - name: customers

    - name: stripe
      tables:
      - name: payments
```

N.B.: we can avoid to add schema/database In dbt, we refer to sources using the `source` macro: `{{ source('source_name', 'object_name') }}`.
Lets implement this in the models.

- [ ] Rafactor `stg_jaffle_shop__orders` and `stg_jaffle_shop__customers` to use sources.

- [ ] Create a staging model for payments source `models/stg_stripe__payments.sql`.
To save time we can use the *codegen* package:

    - go to https://hub.getdbt.com/
    - install codegen package
    - compile in a new file:

```
{{ codegen.generate_base_model(
    source_name='stripe',
```

```
    table_name='payments',
    materialized='table'
) }}
```

- [ ] Look at the lineage now.
- [ ] Do a `dbt run`.

Project structure

How we structure our dbt projects. For example:

```
snowflake_workshop
├── README.md
├── analyses
├── seeds
│   └── employees.csv
├── dbt_project.yml
├── macros
│   └── cents_to_dollars.sql
├── models
│   ├── intermediate *(by area)*
│   │   └── finance
│   ├── marts *(by area)*
│   │   ├── finance
│   │   ├── marketing
│   │   └── etc.
│   ├── staging *(by source)*
│   │   ├── jaffle_shop
│   │   └── etc.
│   └── utilities
├── packages.yml
├── snapshots
└── tests
```

- [ ] Create the `staging`, `marts` and `intermediate` folders.
- [ ] Refactor the *customer model* into `marts/marketing/dim_customers.sql`
- [ ] Refactor the staging models we created into the staging folder
- [ ] Configure to materialize the *marts models* as tables and the *staging models* as views.

```
models:
    snowflake_workshop:
        staging:
            +materialized: view
```

```
        marts:
            +materialized: table
```

- [ ] Create a `fct_orders.sql` into `marts/finance` folder.

```
with orders as  (
    select * from {{ ref ('stg_jaffle_shop__orders' )}}
),

payments as (
    select * from {{ ref ('stg_stripe__payments') }}
),

order_payments as (
    select
        order_id,
        sum (case when status = 'success' then amount end) as amount

    from payments
    group by 1
),

 final as (

    select
        orders.order_id,
        orders.customer_id,
        orders.order_date,
        coalesce (order_payments.amount, 0) as amount

    from orders
    left join order_payments using (order_id)
)

select * from final
```

- [ ] *dim_customer* model should refer to this now.

Source Freshness - link

- Freshness thresholds can be set in the YML file where sources are configured.
  For each table, the keys `loaded_at_field` and freshness must be configured.

- A threshold can be configured for giving a warning and an error with the keys `warn_after` and `error_after`.
- The freshness of sources can then be determined with the command `dbt source freshness`.
  difference between the current time and the last timestamp

- [ ] Configure source freshness for the *orders* table.

```yaml
version: 2

sources:
- name: jaffle_shop
    database: raw
    tables:
    - name: customers
    - name: orders
        config:
        freshness:
            warn_after:
            count: 6
            period: hour
            error_after:
            count: 30
            period: day
        loaded_at_field: _etl_loaded_at

- name: stripe
    database: raw
    tables:
    - name: payments
```

- [ ] Run `dbt source freshness`
- [ ] Look at the query log.

Testing - link

In dbt, there are two types of tests:

- **Generic tests**
  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**
  Singular tests are data tests defined by **writing specific SQL queries** that return records

which fail the test conditions

dbt ships with four built in tests: *unique, not null, accepted values, relationships*.

Tests can be run against your current project using a range of commands:

- `dbt test` runs all tests in the dbt project
- `dbt test --select one_specific_model`

- [ ] Create the yml file to define the tests.
`models/staging/jaffle_shop/_stg_jaffle_shop.yml`

```yaml
models:
- name: stg_jaffle_shop__customers
    columns:
    - name: customer_id
        tests:
        - unique
        - not_null
- name: stg_jaffle_shop__orders
    columns:
    - name: order_id
        tests:
        - unique
        - not_null
    - name: status
        data_tests:
        - accepted_values:
            arguments:
                values:
                - completed
                - shipped
                - returned
                - placed
                - return_pending
    - name: customer_id
        data_tests:
        - relationships:
            arguments:
                to: ref('stg_jaffle_shop__customers')
                field: customer_id
```

- [ ] Create a singular test in `tests\assert_positive_value_for_total_amount.sql`

```sql
-- Refunds have a negative amount, so the total amount should always be >=
-- Therefore return records where this isn't true to make the test fail.
select
    order_id,
    sum(amount) as total_amount
from {{ ref('stg_stripe__payment') }}
group by 1
having (total_amount < 0)
```

- [ ] test the models.
- [ ] Build the models.

Documentation - link

- Documentation is essential for an analytics team to work effectively and efficiently.
- Strong documentation empowers users to self-service questions about data and enables new team members to on-board quickly.

> Documentation should be as automated as possible and happen as close as possible to the coding.

In dbt, models are built in SQL files. These models are documented in YML files that live in the same folder as the models.

- If a longer form, more styled version of text would provide a strong description, **doc blocks** can be used to render markdown in the generated documentation.

- [ ] Add documentation to the file `models/staging/jaffle_shop/_stg_jaffle_shop.yml`.
- [ ] Add a description for your `stg_jaffle_shop__customers` model and the column customer_id.
- [ ] Add a description for your `stg_jaffle_shop__orders` model and the column order_id.
- [ ] Create a reference to a doc block
- [ ] Create a doc block for your `stg_jaffle_shop__orders` model to document the status column. Reference this doc block in the description of status in `stg_jaffle_shop__orders`.

Result: `models/staging/jaffle_shop/_stg_jaffle_shop.yml`

```yaml
models:
  - name: stg_jaffle_shop__customers
    description: Staged customer data from our jaffle shop app.
    columns:
      - name: customer_id
        description: The primary key for customers.
        data_tests:
          - unique
```

```
          - not_null

  - name: stg_jaffle_shop__orders
    description: Staged order data from our jaffle shop app.
    columns:
      - name: order_id
        description: Primary key for orders.
        data_tests:
          - unique
          - not_null
      - name: status
        description: '{{ doc("order_status") }}'
        data_tests:
          - accepted_values:
              arguments:
                values:
                  - completed
                  - shipped
                  - returned
                  - placed
                  - return_pending
      - name: customer_id
        description: Foreign key to stg_customers.customer_id
        data_tests:
          - relationships:
              arguments:
                to: ref('stg_jaffle_shop__customers')
                field: customer_id
```

and `models/staging/jaffle_shop/_jaffle_shop.md` or create a `docs` folder and add `docs-paths: ["docs"]` in the `dbt_project.yml` file.

```
{% 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 %}
```

- [ ] create the project documentation: `dbt docs denerate`.
- [ ] take a look.

Deployment

- A deployment environment can be configured in dbt on the Orchestration > Environments page.
- General Settings: You can configure which dbt version you want to use and you have the option to specify a branch other than the default branch.
- Data Warehouse Connection: You can set data warehouse specific configurations. For example, you may choose to use a dedicated warehouse for your production runs in Snowflake.
- Deployment Credentials: where you enter the credentials dbt will use to access your data warehouse:

Scheduling a job in dbt

- Scheduling of future jobs can be configured in dbt on the Jobs page.
- Commands: A single job can run multiple dbt commands. For example, you can run dbt run and dbt test back to back on a schedule. You don't need to configure these as separate jobs.

Reviewing Jobs
The results of a particular job run can be reviewed as the job completes and over time. The logs for each command can be reviewed. If documentation was generated, this can be viewed. If dbt source freshness was run, the results can also be viewed at the end of a job.

- [ ] setup a deployment environment.
- [ ] setup a *deployment* job that:
  - run marts models at 30th minutes of every hour
  - test the models
  - check data fresheness.
  - updates the documentation.

dbt Catalog

Extras

pivot table with jinja

jinja intercat with variables/models before compile the sql code Syntax:

- `{% %}` operation inside Jinja context
- `{{ x }}` we are pulling something out of the jinja context and printing it in the sql code

```
{% set temperature = 10.0 %}
Se sono in montagna e sono
{% if temperature > 30.0 %}
    {{ temperature }} gradi mi piace mangiare il gelato
{% if temperature < 10.0 %}
    {{ temperature }} gradi mi piace mangiare uno strudel caldo
{% else %}
    {{ temperature }} gradi mi piace mangare un panino
{% endif %}
```

aggiunta dash per sottrarre la linea di codice -

```
{%- set temperature = 30.0 -%}
Se sono in montagna e sono
{%- if temperature > 30.0 -%}
    {{ temperature }} gradi mi piace mangiare il gelato
{%- if temperature < 10.0 -%}
    {{ temperature }} gradi mi piace mangiare uno strudel caldo
{%- else -%}
    {{ temperature }} gradi mi piace mangare un panino
{%- endif -%}
```

```
-- 1.
{% set frase = 'mi piace mangiare molto la pizza'}
{{ frase }}

-- 2.
{% set animals = ['canguro', 'volpe', 'delfino'] %}
{{ animals[0] }}

{% set animals = ['canguro', 'volpe', 'delfino'] %}
{{ animals[0] }}

{% for j in range(10) %}
    select {{ j }} as number union all
{% endfor %}

{% for j in range(10) %}
    select {{ j }} as number {% if not loop.last %} union all {% endif %}
{% endfor %}
```