

Analytics Engineering

Traditional Data Teams

- Data engineers are responsible for maintaining data infrastructure and the ETL process for creating tables and views.
- Data analysts focus on querying tables and views to drive business insights for stakeholders.

ETL and ELT

- ETL (extract transform load) is the process of creating new database objects by extracting data from multiple data sources, transforming it on a local or third party machine, and loading the transformed data into a data warehouse.
- ELT (extract load transform) is a more recent process of creating new database objects by first extracting and loading raw data into a data warehouse and then transforming that data directly in the warehouse.
- The new ELT process is made possible by the introduction of cloud-based data warehouse technologies.

Analytics Engineering

- Analytics engineers focus on the transformation of raw data into transformed data that is ready for analysis. This new role on the data team changes the responsibilities of data engineers and data analysts.
- Data engineers can focus on larger data architecture and the EL in ELT.
- Data analysts can focus on insight and dashboard work using the transformed data.
- Note: At a small company, a data team of one may own all three of these roles and responsibilities. As your team grows, the lines between these roles will remain blurry.

dbt

- dbt empowers data teams to leverage software engineering principles for transforming data.
- The focus of this course is to build your analytics engineering mindset and dbt skills to give you more leverage in your work.

Models in dbt

dbt Models

- A model in dbt is a query. Queries can become complex and be inter-dependent, for example the one depicted to the right.
- Here, we find CTEs. A Common Table Expression (CTE), is a temporary result set that can be used in a SQL query. You can use CTEs to break up complex queries into simpler blocks of code that can connect and build on each other. In a less formal, more human-sense, you can think of a CTE as a separate, smaller query within the larger query you're building up. Creating a CTE is essentially like making a temporary view that you can access throughout the rest of the query you are writing.

```
1  with customers as (  
2  
3      select  
4          id as customer_id,  
5          first_name,  
6          last_name  
7  
8      from `dbt-tutorial`.jaffle_shop.customers  
9  
10 ),  
11  
12 orders as (  
13  
14     select  
15         id as order_id,  
16         user_id as customer_id,  
17         order_date,  
18         status  
19  
20     from `dbt-tutorial`.jaffle_shop.orders  
21  
22 ),  
23  
24 customer_orders as (  
25  
26     select  
27         customer_id,  
28  
29         min(order_date) as first_order_date,  
30         max(order_date) as most_recent_order_date,  
31         count(order_id) as number_of_orders  
32  
33     from orders  
34  
35     group by 1  
36  
37 ),  
38  
39 final as (  
40  
41     select  
42         customers.customer_id,  
43         customers.first_name,  
44         customers.last_name,  
45         customer_orders.first_order_date,  
46         customer_orders.most_recent_order_date,  
47         coalesce(customer_orders.number_of_orders, 0) as number_of_orders  
48  
49     from customers  
50  
51     left join customer_orders using (customer_id)  
52  
53 )  
54  
55 select * from final
```

Modularity and the Ref Function

- Instead of having one big model with several CTEs, we break the Query/Model down into several queries/models which can still be inter-dependent:

```
1 with customers as (  
2   select  
3     id as customer_id,  
4     first_name,  
5     last_name  
6   from 'dbt-tutorial'.jaffle_shop.customers  
7 )  
8  
9  
10  
11  
12 orders as (  
13   select  
14     id as order_id,  
15     user_id as customer_id,  
16     order_date,  
17     status  
18   from 'dbt-tutorial'.jaffle_shop.orders  
19 )  
20  
21  
22  
23  
24 customer_orders as (  
25  
26   select  
27     customer_id,  
28  
29     min(order_date) as first_order_date,  
30     max(order_date) as most_recent_order_date,  
31     count(order_id) as number_of_orders  
32  
33   from orders  
34  
35   group by 1  
36 )  
37  
38  
39  
40 final as (  
41   select  
42     customers.customer_id,  
43     customers.first_name,  
44     customers.last_name,  
45     customer_orders.first_order_date,  
46     customer_orders.most_recent_order_date,  
47     coalesce(customer_orders.number_of_orders, 0) as number_of_orders  
48  
49   from customers  
50  
51   left join customer_orders using (customer_id)  
52  
53 )  
54  
55 select * from final
```



```
models/stg_customers.sql  
  
select  
  id as customer_id,  
  first_name,  
  last_name  
  
from 'dbt-tutorial'.jaffle_shop.customers
```

```
models/stg_orders.sql  
  
select  
  id as order_id,  
  user_id as customer_id,  
  order_date,  
  status  
  
from 'dbt-tutorial'.jaffle_shop.orders
```

```
models/customers.sql  
  
with customers as (  
  select * from {{ ref('stg_customers') }}  
)  
,  
orders as (  
  select * from {{ ref('stg_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  
  
  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  
  
  from customers  
  
  left join customer_orders using (customer_id)  
)  
  
select * from final
```



dbt run --full-refresh success

add-customers-model

less than a minute ago

> System Logs

All 3 Pass 3 Warn 0 Error 0

> ✓ stg_customers

> ✓ stg_orders

> ✓ customers

Modularity and the Ref Function

- The model putting everything together (customers.sql) uses the ref function from dbt to create the CTEs
- The ref function allows to pull information from a model that has already been materialized in our Warehouse (BigQuery)

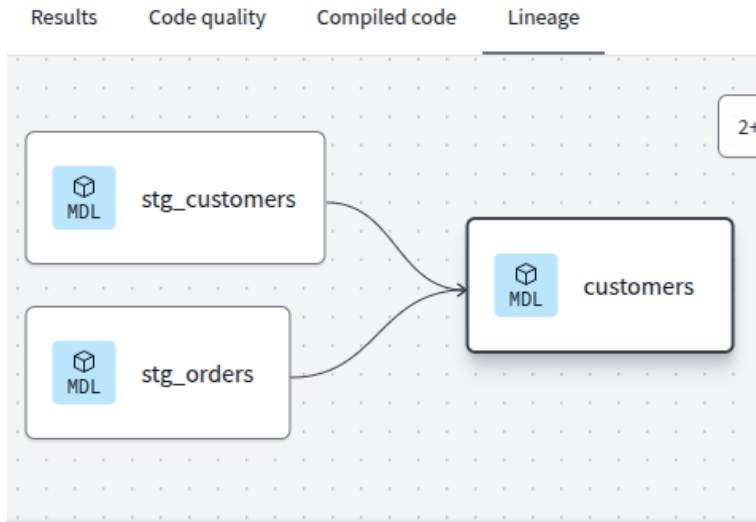
```
with customers as (  
  select * from {{ ref('stg_customers') }}  
)  
orders as (  
  select * from {{ ref('stg_orders') }}  
)  
customer_orders as (  
  select  
    customer_id,  
    min(order_date) as first_order_date,  
    max(order_date) as most_recent_order_date,  
    count(order_id) as number_of_orders  
  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  
  from customers  
  left join customer_orders using (customer_id)  
)  
select * from final
```

CTE

CTE

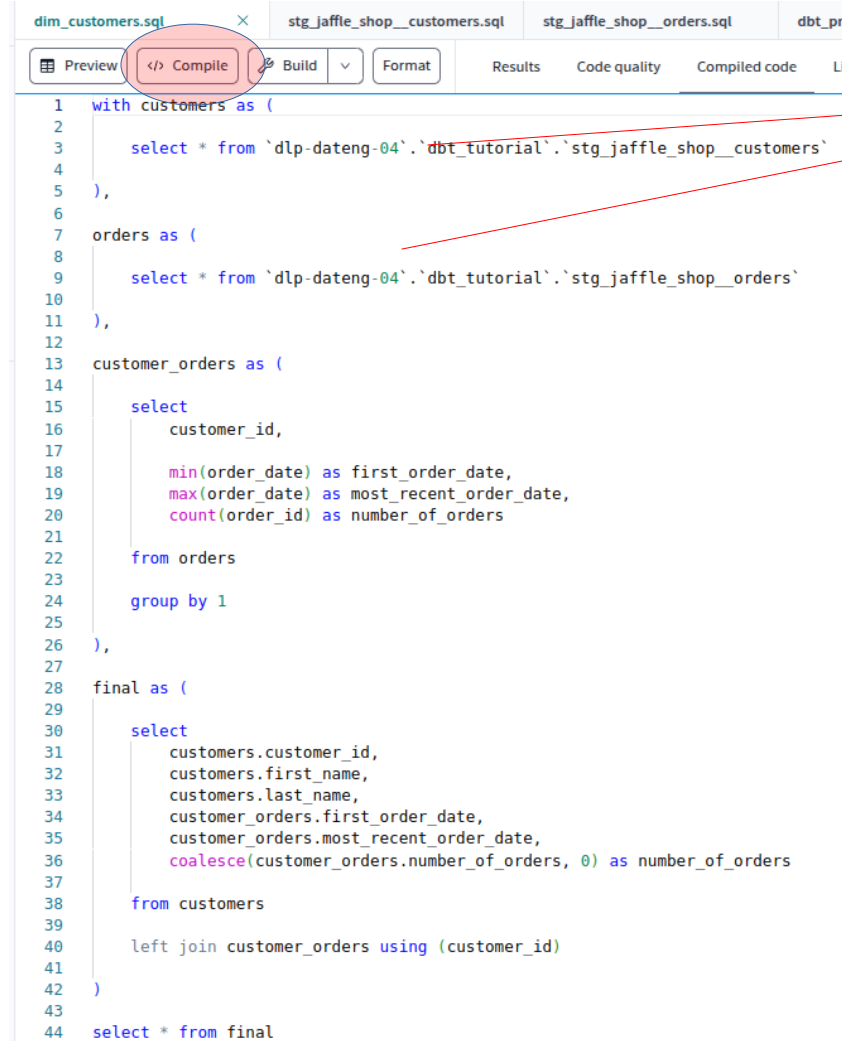
CTE

CTE



Modularity and the Ref Function

- The advantage of using the ref function instead of raw SQL code, is that it manages the dependencies automatically.
- The ref function constructs the inter-dependent queries based on the environmental variables from each developer/user, so sharing the dbt Models between developers becomes easier.
- It can be thought of as working with relative filepaths instead of absolute filepaths.
- To see what the ref function actually translates to in SQL, we can use the compile button in the cloud IDE:



```
1 with customers as (  
2  
3     select * from `dlp-dateng-04`.`dbt_tutorial`.`stg_jaffle_shop_customers`  
4  
5 ),  
6  
7 orders as (  
8  
9     select * from `dlp-dateng-04`.`dbt_tutorial`.`stg_jaffle_shop_orders`  
10  
11 ),  
12  
13 customer_orders as (  
14  
15     select  
16         customer_id,  
17  
18         min(order_date) as first_order_date,  
19         max(order_date) as most_recent_order_date,  
20         count(order_id) as number_of_orders  
21  
22     from orders  
23  
24     group by 1  
25  
26 ),  
27  
28 final as (  
29  
30     select  
31         customers.customer_id,  
32         customers.first_name,  
33         customers.last_name,  
34         customer_orders.first_order_date,  
35         customer_orders.most_recent_order_date,  
36         coalesce(customer_orders.number_of_orders, 0) as number_of_orders  
37  
38     from customers  
39  
40     left join customer_orders using (customer_id)  
41  
42 )  
43  
44 select * from final
```

This paths are constructed by the ref function based on the dbt user's environmental variables, hence making the code easily shareable between users.

In summary: the path to the source data is absolute but the path to the models needed for a given analysis are relative to the user's environment.

Naming Conventions

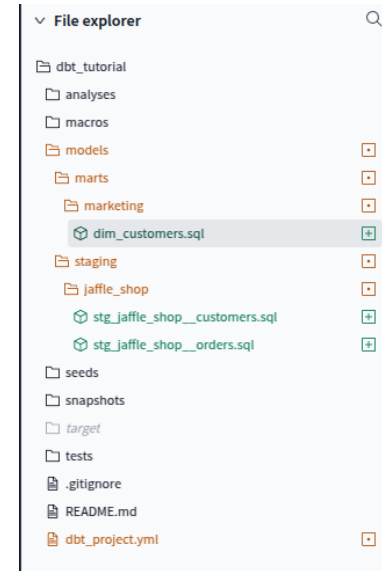
There are 5 naming conventions:

- Sources:
 - References to the raw data in the Data Lake or the Datawarehouse
 - Staging models: light computations (edit column name, change data type, lowercase, etc.)
- Intermediate:
 - Anything between staging and the final table
 - Usually hidden from the final user
 - Always reference the staging models. Should not directly reference the source tables
- Fact:
 - Skinny tables that are very long
 - Represent things that are occurring or have occurred (e.g. incoming events, orders)
- Dimension
 - Things that exist (e.g. customers, zones, things that don't change that often)

Organizing a dbt project

- dbt Models should be organized in folders
- This allows for better structure but also to apply a given configuration to all the models within that folder
- This folder-wide configurations can be defined in the .yml file
 - For example to specify if the models should generate views or materialized tables:

```
models:
  # Project name
  jaffle_shop:
    # Applies to all files under models/marts/
    marts:
      materialized: table
    # Applies to all files under models/staging/
    staging:
      materialized: view
```



Review

Models

- Models are .sql files that live in the models folder.
- Models are simply written as select statements - there is no DDL/DML that needs to be written around this. This allows the developer to focus on the logic.
- In the Cloud IDE, the Preview button will run this select statement against your data warehouse. The results shown here are equivalent to what this model will return once it is materialized.
- After constructing a model, `dbt run` in the command line will actually materialize the models into the data warehouse. The default materialization is a view.
- The materialization can be configured as a table with the following configuration block at the top of the model file:

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

- The same applies for configuring a model as a view:

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

- When `dbt run` is executing, dbt is wrapping the select statement in the correct DDL/DML to build that model as a table/view. If that model already exists in the data warehouse, dbt will automatically drop that table or view before building the new database object.

*Note: If you are on BigQuery, you may need to run `dbt run --full-refresh` for this to take effect.

Review

- The DDL/DML that is being run to build each model can be viewed in the logs through the cloud interface or the target folder.

The screenshot displays the dbt cloud interface. At the top, a navigation bar shows 'Runs' and 'dbt run' with a 'ready' status indicator. Below this, a sidebar on the left lists 'dbt run' and 'master' with a green checkmark and a '16s' duration. The main panel shows the logs for the 'stg_orders' model. The logs include timestamps, status messages, and the SQL code being executed. The SQL code is a 'create or replace view' statement for 'analytics.dbt_kcoapman.stg_orders' that selects columns from 'raw.jaffle_shop.orders'.

```
2020-09-30 01:14:28.388375Z: On model.jaffle_shop.stg_orders: BEGIN
2020-09-30 01:14:29.044568Z: SQL status: SUCCESS 1 in 0.66 seconds
2020-09-30 01:14:29.044727Z: Using snowflake connection "model.jaffle_shop.stg_orders".
2020-09-30 01:14:29.044796Z: On model.jaffle_shop.stg_orders: /* {"app": "dbt", "dbt_version": "0.17.2", "profile_name": "user", "target_name": "default", "node_id": "model.jaffle_shop.stg_orders"} */

create or replace view analytics.dbt_kcoapman.stg_orders as (
  select
    id as order_id,
    user_id as customer_id,
    order_date,
    status

from raw.jaffle_shop.orders
);
2020-09-30 01:14:29.401539Z: SQL status: SUCCESS 1 in 0.36 seconds
2020-09-30 01:14:29.402765Z: On model.jaffle_shop.stg_orders: COMMIT
2020-09-30 01:14:29.402899Z: Using snowflake connection "model.jaffle_shop.stg_orders".
2020-09-30 01:14:29.402955Z: On model.jaffle_shop.stg_orders: COMMIT
2020-09-30 01:14:29.527581Z: SQL status: SUCCESS 1 in 0.12 seconds
2020-09-30 01:14:29.528047Z: finished collecting timing info
```

Review

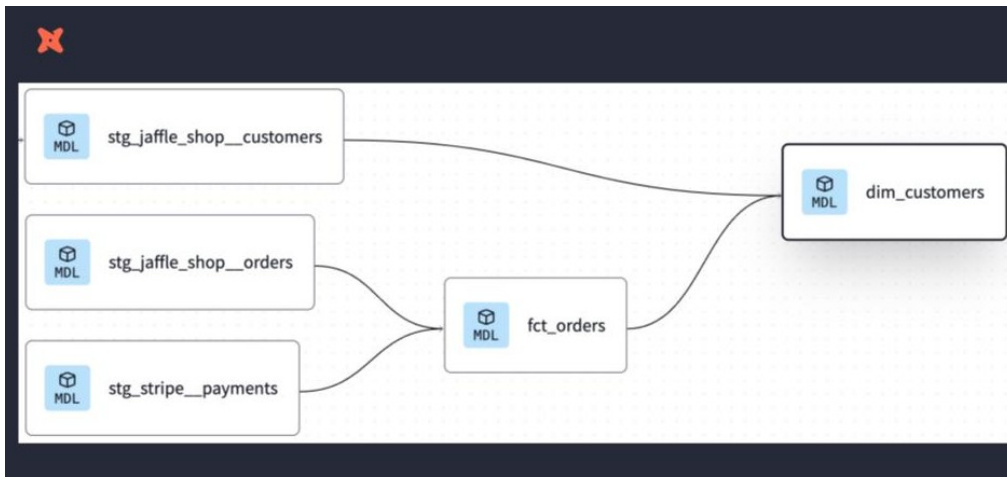
Modularity

- We could build each of our final models in a single model as we did with `dim_customers`, however with dbt we can create our final data products using modularity.
- **Modularity** is the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use.
- This allows us to build data artifacts in logical steps.
- For example, we can stage the raw customers and orders data to shape it into what we want it to look like. Then we can build a model that references both of these to build the final `dim_customers` model.
- Thinking modularly is how software engineers build applications. Models can be leveraged to apply this modular thinking to analytics engineering.

Review

ref Macro

- Models *can* be written to reference the underlying tables and views that were building the data warehouse (e.g. `analytics.dbt_jsmith.stg_jaffle_shop_customers`). This hard codes the table names and makes it difficult to share code between developers.
- The `ref` function allows us to build dependencies between models in a flexible way that can be shared in a common code base. The `ref` function compiles to the name of the database object as it has been created on the most recent execution of `dbt run` in the particular development environment. This is determined by the environment configuration that was set up when the project was created.
- Example: `{{ ref('stg_jaffle_shop_customers') }}` compiles to `analytics.dbt_jsmith.stg_jaffle_shop_customers`.
- The `ref` function also builds a lineage graph like the one shown below. dbt is able to determine dependencies between models and takes those into account to build models in the correct order.



Review

Modeling History

- There have been multiple modeling paradigms since the advent of database technology. Many of these are classified as normalized modeling.
- Normalized modeling techniques were designed when storage was expensive and computational power was not as affordable as it is today.
- With a modern cloud-based data warehouse, we can approach analytics differently in an *agile* or *ad hoc* modeling technique. This is often referred to as denormalized modeling.
- dbt can build your data warehouse into any of these schemas. dbt is a tool for *how* to build these rather than enforcing *what* to build.

Review

Naming Conventions

In working on this project, we established some conventions for naming our models.

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

Review

Reorganize Project

- When `dbt run` is executed, dbt will automatically run every model in the models directory.
- The subfolder structure within the models directory can be leveraged for organizing the project as the data team sees fit.
- This can then be leveraged to select certain folders with `dbt run` and the model selector.
- Example: If `dbt run -s staging` will run all models that exist in `models/staging`. (Note: This can also be applied for `dbt test` as well which will be covered later.)
- The following framework can be a starting part for designing your own model organization:
- **Marts** folder: All intermediate, fact, and dimension models can be stored here. Further subfolders can be used to separate data by business function (e.g. marketing, finance)
- **Staging** folder: All staging models and source configurations can be stored here. Further subfolders can be used to separate data by data source (e.g. Stripe, Segment, Salesforce). (We will cover configuring Sources in the Sources module)

```
graph TD; macros[macros]; models[models]; marts[marts]; finance[finance]; marketing[marketing]; staging[staging]; jaffle_shop[jaffle_shop]; stripe[stripe]; seeds[seeds]; fct_orders[fct_orders.sql]; dim_customers[dim_customers.sql]; stg_customers[stg_jaffle_shop__customers.sql]; stg_orders[stg_jaffle_shop__orders.sql]; stg_payments[stg_stripe__payments.sql]; models --- marts; models --- finance; models --- marketing; models --- staging; models --- stripe; marts --- fct_orders; marts --- dim_customers; staging --- jaffle_shop; jaffle_shop --- stg_customers; jaffle_shop --- stg_orders; stripe --- stg_payments;
```

Sources in dbt

Sources

- Sources represent the raw data that is loaded into the data warehouse.
- We *can* reference tables in our models with an explicit table name (`raw.jaffle_shop.customers`).
- However, setting up Sources in dbt and referring to them with the `source` function enables a few important tools.
 - Multiple tables from a single source can be configured in one place.
 - Sources are easily identified as green nodes in the Lineage Graph.
 - You can use `dbt source freshness` to check the freshness of raw tables.

Configuring sources

- Sources are configured in YAML files in the models directory.
- The following code block configures the table `raw.jaffle_shop.customers` and `raw.jaffle_shop.orders`:

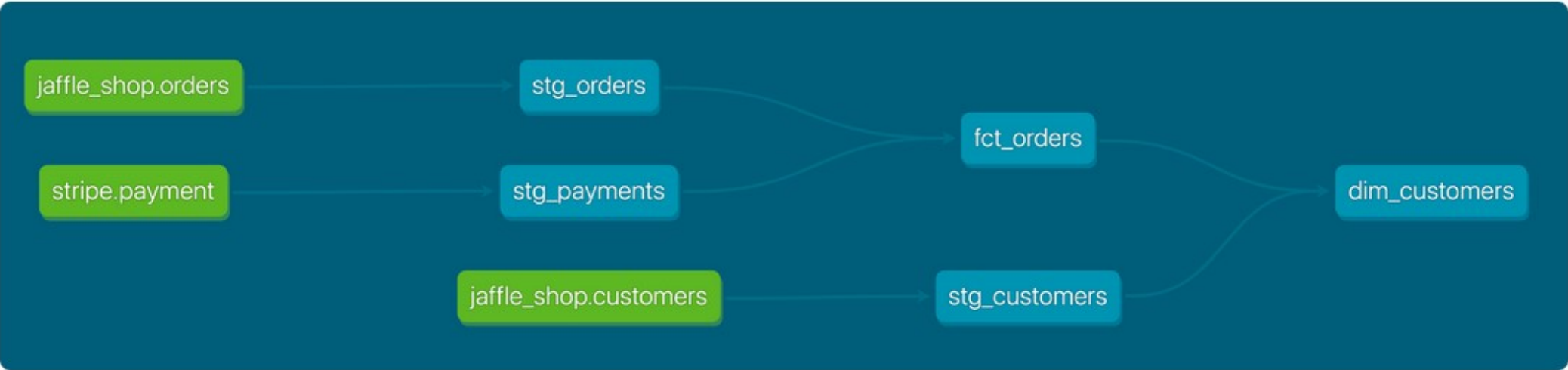
```
version: 2

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

- View the full documentation for configuring sources on the [source properties](#) page of the docs.

Source function

- The `ref` function is used to build dependencies between models.
- Similarly, the `source` function is used to build the dependency of one model to a source.
- Given the source configuration above, the snippet `{{ source('jaffle_shop', 'customers') }}` in a model file will compile to `raw.jaffle_shop.customers`.
- The Lineage Graph will represent the sources in green.



Source freshness

- 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.

```
version: 2

sources:
  - name: jaffle_shop
    database: raw
    schema: jaffle_shop
    tables:
      - name: orders
        loaded_at_field: _etl_loaded_at
        freshness:
          warn_after: {count: 12, period: hour}
          error_after: {count: 24, period: hour}
```

- 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`.

Testing in dbt

Testing

- **Testing** is used in software engineering to make sure that the code does what we expect it to.
- In Analytics Engineering, testing allows us to make sure that the SQL transformations we write produce a model that meets our assertions.
- In dbt, tests are written as select statements. These select statements are run against your materialized models to ensure they meet your assertions.

Tests in dbt

- In dbt, there are two types of tests - generic tests and singular tests:
 - **Generic tests** are a way to validate your data models and ensure data quality. These tests are predefined and can be applied to any column of your data models to check for common data issues. They are written in YAML files.
 - **Singular tests** are data tests defined by writing specific SQL queries that return records which fail the test conditions. These tests are referred to as "singular" because they are one-off assertions that are uniquely designed for a single purpose or specific scenario within the data models.
- dbt ships with four built in tests: unique, not null, accepted values, relationships.
 - **Unique** tests to see if every value in a column is unique
 - **Not_null** tests to see if every value in a column is not null
 - **Accepted_values** tests to make sure every value in a column is equal to a value in a provided list
 - **Relationships** tests to ensure that every value in a column exists in a column in another model (see: [referential integrity](#))
- 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 test_type:generic`
 - `dbt test --select test_type:singular`
 - `dbt test --select one_specific_model`
- Read more here in [testing documentation](#).

- In development, dbt Cloud will provide a visual for your test results. Each test produces a log that you can view to investigate the test results further.

dbt test

🔗 adding_doc_block

↓ Logs

Passed	25	0	0	0	0	16:39:55	24 seconds
RUN STATUS	PASS	WARN	FAIL	SKIPPED	QUEUED	START	DURATION

SYSTEM LOGS

> view logs

DETAILS

> accepted_values_fct_orders_status__placed__shipped__completed__return_pending__returned	1s	●
> accepted_values_stg_orders_status__placed__shipped__completed__return_pending__returned	771ms	●
> accepted_values_stg_payments_payment_method_bank_transfer_coupon_credit_card_gift_card	693ms	●
> not_null_dim_customers_customer_id	971ms	●
> not_null_fct_orders_order_id	1s	●

In production, dbt Cloud can be scheduled to run `dbt test`. The 'Run History' tab provides a similar interface for viewing the test results.

Details

Timing		Artifacts	
1 minute, 7 seconds ago RUN TRIGGERED	48 seconds TIME IN QUEUE	3 seconds ago COMPLETED	1 minute, 4 seconds COMPLETED AFTER

Run Steps

✓ SUCCESS - 00:00:00	Clone Git Repository	SHOW LOGS +
✓ SUCCESS - 00:00:00	Create Profile from Connection Snowflake	SHOW LOGS +
✓ SUCCESS - 00:00:00	Invoke dbt with `dbt deps`	SHOW LOGS +
✗ ERROR - 00:00:12	Invoke dbt with `dbt test`	SHOW LOGS +

You've learned about the 4 built-in generic tests, but there are so many more tests in packages you could be using! Learn about them in our free online course on [Jinja, Macros, and Packages](#).

Do you want to take your testing knowledge to the next level? Check out our free online course on [Advanced Testing](#)!

Documentation in dbt

Documentation

- 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 often lags behind the code it is meant to describe. This can happen because documentation is a separate process from the coding itself that lives in another tool.
- Therefore, 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.

Writing documentation and doc blocks

- Documentation of models occurs in the YML files (where generic tests also live) inside the models directory. It is helpful to store the YML file in the same subfolder as the models you are documenting.
- For models, descriptions can happen at the model, source, or column level.
- 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.

Generating and viewing documentation

- In the command line section, an updated version of documentation can be generated through the command `dbt docs generate`. This will refresh the `view docs` link in the top left corner of the Cloud IDE.
- The generated documentation includes the following:
 - Lineage Graph
 - Model, source, and column descriptions
 - Generic tests added to a column
 - The underlying SQL code for each model
 - and more...

Deployment in dbt

Development vs. Deployment

- Development in dbt is the process of building, refactoring, and organizing different files in your dbt project. This is done in a development environment using a development schema (`dbt_jsmith`) and typically on a *non-default* branch (i.e. feature/customers-model, fix/date-spine-issue). After making the appropriate changes, the development branch is merged to main/master so that those changes can be used in deployment.
- Deployment in dbt (or running dbt in production) is the process of running dbt on a schedule in a deployment environment. The deployment environment will typically run from the *default* branch (i.e., main, master) and use a dedicated deployment schema (e.g., `dbt_prod`). The models built in deployment are then used to power dashboards, reporting, and other key business decision-making processes.
- The use of development environments and branches makes it possible to continue to build your dbt project *without* affecting the models, tests, and documentation that are running in production.

Creating your Deployment Environment

- A deployment environment can be configured in dbt Cloud on the 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 here. For example, you may choose to use a dedicated warehouse for your production runs in Snowflake.
- **Deployment Credentials:** Here is where you enter the credentials dbt will use to access your data warehouse:
 - IMPORTANT: When deploying a real dbt Project, you should set up a **separate data warehouse account** for this run. This should not be the same account that you personally use in development.
 - IMPORTANT: The schema used in production should be **different** from anyone's development schema.

Scheduling a job in dbt Cloud

- Scheduling of future jobs can be configured in dbt Cloud on the Jobs page.
- You can select the deployment environment that you created before or a different environment if needed.
- **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.
- **Triggers:** This section is where the schedule can be set for the particular job.
- After a job has been created, you can manually start the job by selecting `Run Now`

Reviewing Cloud 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.

Curious to know more about deploying with dbt Cloud? Check out our free online [Advanced Deployment course](#), where you'll learn how to deploy your dbt Cloud project with advanced functionality including continuous integration, orchestrating conflicting jobs, and customizing behavior by environment!

Want to know how to automate and accelerate your dbt workflow? Learn how with our free online course on [Webhooks!](#)