**Review**

**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.

**Learning Objectives**

* Load training data into your data platform
* Set up an empty repository and connect your GitHub account to dbt Cloud.
* Set up your warehouse and repository connections.
* Navigate the dbt Cloud IDE.
* Complete a simple development workflow in the dbt Cloud IDE.

**Learning Objectives**

* Explain what models are in a dbt project.
* Build your first dbt model.
* Explain how to apply modularity to analytics with dbt.
* Modularize your project with the ref function.
* Review a brief history of modeling paradigms.
* Identify common naming conventions for tables.
* Reorganize your project with subfolders.

**Practice**

Using the resources in this module, complete the following in your dbt project:

**Quick Project Polishing**

* In your dbt\_project.yml file, change the name of your project from my\_new\_project to jaffle\_shop (line 5 AND 35)

**Staging Models**

* Create a staging/jaffle\_shop directory in your models folder.
* Create a stg\_customers.sql model for raw.jaffle\_shop.customers

select

id as customer\_id,

first\_name,

last\_name

from raw.jaffle\_shop.customers

* Create a stg\_orders.sql model for raw.jaffle\_shop.orders

select

id as order\_id,

user\_id as customer\_id,

order\_date,

status

from raw.jaffle\_shop.orders

**Mart Models**

* Create a marts/core directory in your models folder.
* Create a dim\_customers.sql model

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

**Configure your materializations**

* In your dbt\_project.yml file, configure the staging directory to be materialized as views.

models:

jaffle\_shop:

staging:

+materialized: view

* In your dbt\_project.yml file, configure the marts directory to be materialized as tables.

models:

jaffle\_shop:

...

marts:

+materialized: table

**Building a fct\_orders Model**

***This part is designed to be an open ended exercise - see the exemplar on the next page to check your work.***

* Use a statement tab or Snowflake to inspect raw.stripe.payment
* Create a stg\_payments.sql model in models/staging/stripe
* Create a fct\_orders.sql (not stg\_orders) model with the following fields.  Place this in the marts/core directory.
  + order\_id
  + customer\_id
  + amount (hint: this has to come from payments)

**Refactor your dim\_customers Model**

* Add a new field called lifetime\_value to the dim\_customers model:
  + lifetime\_value: the total amount a customer has spent at jaffle\_shop
  + Hint: The sum of lifetime\_value is $1,672

# Exemplar

## Self-check stg\_payments, orders, customers

Use this page to check your work on these three models.

**staging/stripe/stg\_payments.sql**

select

id as payment\_id,

orderid as order\_id,

paymentmethod as payment\_method,

status,

-- amount is stored in cents, convert it to dollars

amount / 100 as amount,

created as created\_at

from raw.stripe.payment

**marts/core/fct\_orders.sql**

with orders as (

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

),

payments as (

select \* from {{ ref('stg\_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

**marts/core/dim\_customers.sql**

\*Note: This is different from the original dim\_customers.sql - you may refactor fct\_orders in the process.

with customers as (

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

),

orders as (

select \* from {{ ref('fct\_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

**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'

) }}

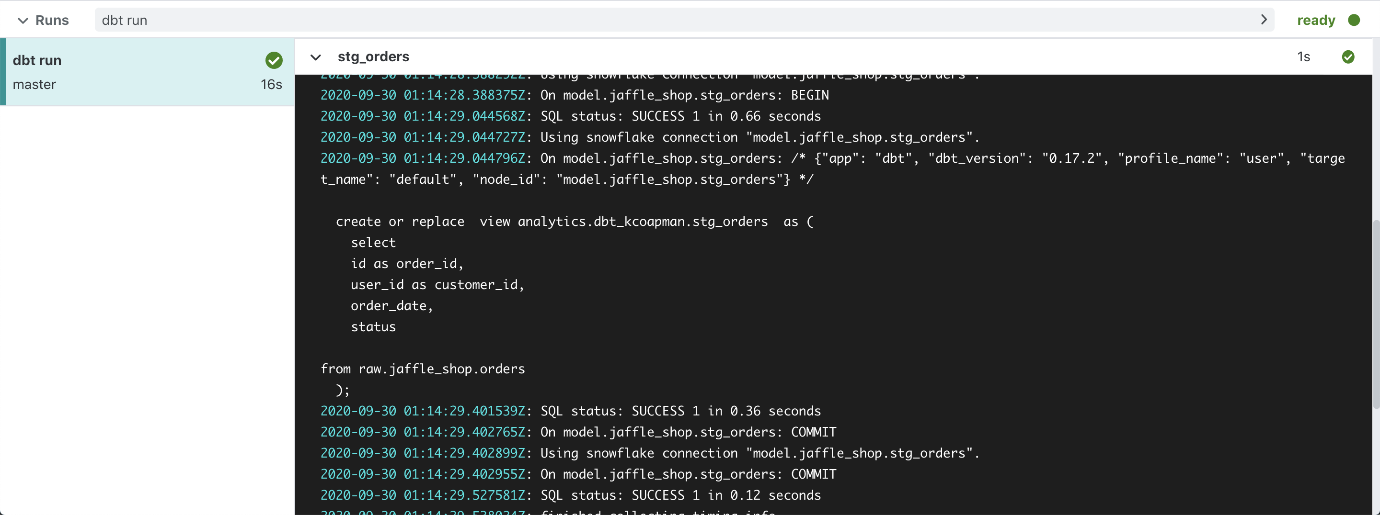
* The same applies for configuring a model as a view:

{{ 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.
* 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.

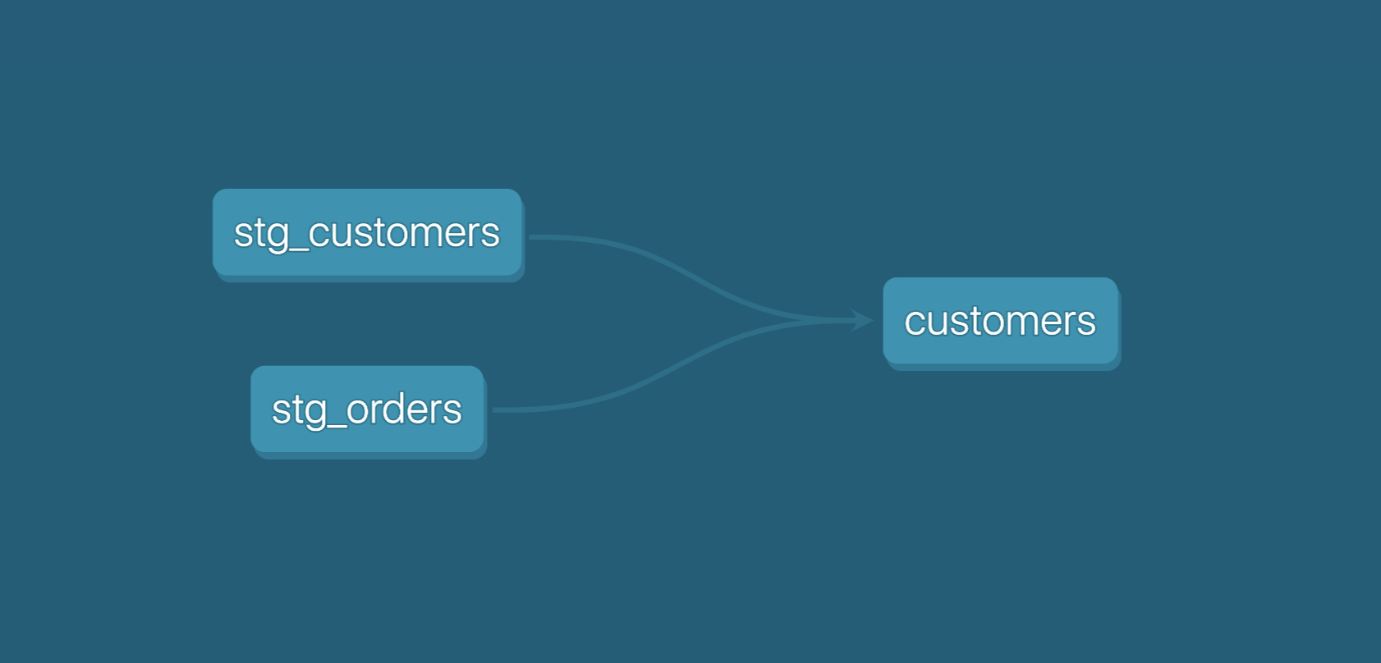


**Modularity**

* 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.

**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\_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\_customers') )} compiles to analytics.dbt\_jsmith.stg\_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.



**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.

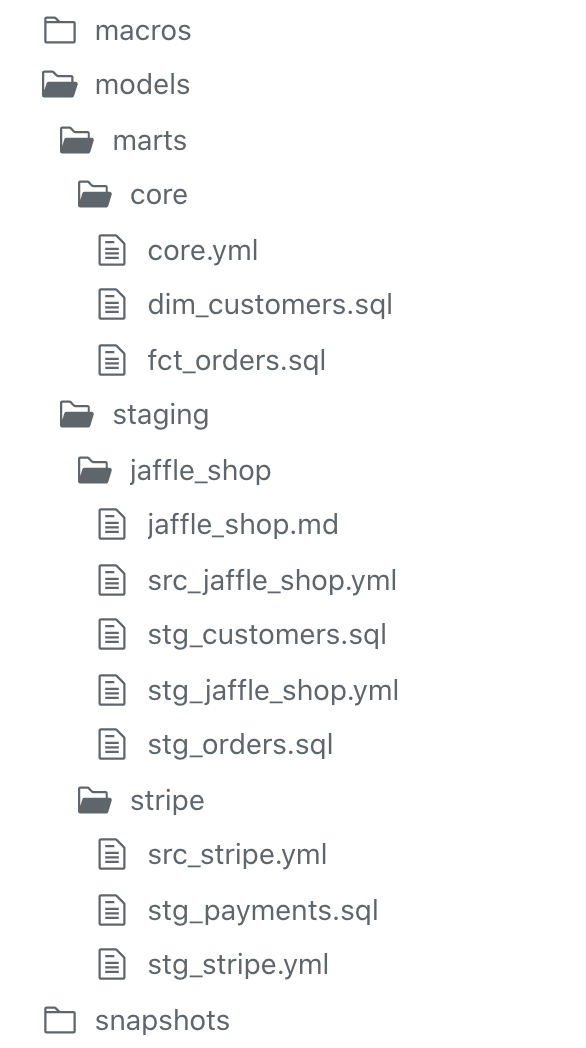
**Naming Conventions**

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.

**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)



**Learning Objectives**

* Explain the purpose of sources in dbt.
* Configure and select from sources in your data warehouse.
* View sources in the lineage graph.
* Check the last time sources were updated and raise warnings if stale.

**Practice**

Using the resources in this module, complete the following in your dbt project.

**Configure sources**

* Configure a source for the tables raw.jaffle\_shop.customers and raw.jaffle\_shop.orders in a file called src\_jaffle\_shop.yml.

**models/staging/jaffle\_shop/src\_jaffle\_shop.yml**

version: 2

sources:

- name: jaffle\_shop

database: raw

  schema: jaffle\_shop

tables:

- name: customers

- name: orders

* Extra credit: Configure a source for the table raw.stripe.payment in a file called src\_stripe.yml.

**Refactor staging models**

* Refactor stg\_customers.sql using the source function.

**models/staging/jaffle\_shop/stg\_customers.sql**

select

    id as customer\_id,

first\_name,

last\_name

from {{ source('jaffle\_shop', 'customers') }}

* Refactor stg\_orders.sql using the source function.

**models/staging/jaffle\_shop/stg\_orders.sql**

select

id as order\_id,

user\_id as customer\_id,

order\_date,

status

from {{ source('jaffle\_shop', 'orders') }}

* Extra credit: Refactor stg\_payments.sql using the source function.

**Extra credit**

* Configure your Stripe payments data to check for source freshness.
* Run dbt source freshness.

**Note:** If you are working on BigQuery, the loaded\_at\_field is called '\_batched\_at' (not '\_etl\_loaded\_at')

Table

Description automatically generated

You can configure your sources.yml file as below:

version: 2

sources:

  - name: stripe

    database: dbt-tutorial

    schema: stripe

    tables:

      - name: payment

        loaded\_at\_field: \_batched\_at

        freshness:

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

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

# Exemplar

## Self-check src\_stripe and stg\_payments

Use this page to check your work.

**models/staging/stripe/src\_stripe.yml**

version: 2

sources:

- name: stripe

database: raw

    schema: stripe

tables:

- name: payment

**models/staging/stripe/stg\_payments.sql**

select

id as payment\_id,

orderid as order\_id,

paymentmethod as payment\_method,

status,

-- amount is stored in cents, convert it to dollars

amount / 100 as amount,

created as created\_at

from {{ source('stripe', 'payment') }}

**Review**

**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 YML 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](https://docs.getdbt.com/reference/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.

Graphical user interface

Description automatically generated

**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.

**Learning Objectives**

* Explain why testing is crucial for analytics.
* Explain the role of testing in analytics engineering.
* Configure and run generic tests in dbt.
* Write, configure, and run specific tests in dbt.

**Practice**

Using the resources in this module, complete the following exercises in your dbt project:

**Generic Tests**

* Add tests to your jaffle\_shop staging tables:
  + Create a file called stg\_jaffle\_shop.yml for configuring your tests.
  + Add unique and not\_null tests to the keys for each of your staging tables.
  + Add an accepted\_values test to your stg\_orders model for status.
  + Run your tests.

**models/staging/jaffle\_shop/stg\_jaffle\_shop.yml**

version: 2

models:

  - name: stg\_customers

    columns:

      - name: customer\_id

        tests:

          - unique

          - not\_null

  - name: stg\_orders

    columns:

- name: order\_id

      tests:

          - unique

          - not\_null

      - name: status

tests:

- accepted\_values:

values:

- completed

- shipped

- returned

- return\_pending

- placed

**Singular Tests**

* Add the test tests/assert\_positive\_value\_for\_total\_amount.sql to be run on your stg\_payments model.
* Run your tests.

**tests/assert\_positive\_value\_for\_total\_amount.sql**

-- Refunds have a negative amount, so the total amount should always be >= 0.

-- Therefore return records where this isn't true to make the test fail.

select

order\_id,

sum(amount) as total\_amount

from {{ ref('stg\_payments') }}

group by 1

having not(total\_amount >= 0)

**Extra Credit**

* Add a relationships test to your stg\_orders model for the customer\_id in stg\_customers.
* Add tests throughout the rest of your models.
* Write your own specific tests.

# Exemplar

Add a relationships test to your stg\_orders model for the customer\_id in stg\_customers.

**models/staging/jaffle\_shop/stg\_jaffle\_shop.yml**

version: 2

models:

- name: stg\_customers

columns:

- name: customer\_id

tests:

- unique

- not\_null

- name: stg\_orders

columns:

- name: order\_id

tests:

          - unique

- not\_null

- name: status

tests:

- accepted\_values:

values:

- completed

- shipped

- returned

- placed

- return\_pending

- name: customer\_id

tests:

- relationships:

to: ref('stg\_customers')

field: customer\_id

**Review**

**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 - schema tests and data tests:
  + **Generic tests** are written in YAML and return the number of records that do not meet your assertions. These are run on specific columns in a model.
  + **Specific tests** are specific queries that you run against your models. These are run on the entire model.
* 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](https://en.wikipedia.org/wiki/Referential_integrity))
* Generic tests are configured in a YAML file, whereas specific tests are stored as select statements in the tests folder.
* 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](https://docs.getdbt.com/reference/node-selection/test-selection-examples).
* 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.

Graphical user interface, text, application, email

Description automatically generated

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

Graphical user interface, application, Teams

Description automatically generated

**Learning Objectives**

* Explain why documentation is crucial for analytics.
* Understand the documentation features of dbt.
* Write documentation for models, sources, and columns in .yml files.
* Write documentation in markdown using doc blocks.
* Generate and view documentation in development.
* View and navigate the lineage graph to understand the dependencies between models.

**Practice**

Using the resources in this module, complete the following in your dbt project:

**Write documentation**

* Add documentation to the file models/staging/jaffle\_shop/stg\_jaffle\_shop.yml.
* Add a description for your stg\_customers model and the column customer\_id.
* Add a description for your stg\_orders model and the column order\_id.

**Create a reference to a doc block**

* Create a doc block for your stg\_orders model to document the status column.
* Reference this doc block in the description of status in stg\_orders.

**models/staging/jaffle\_shop/stg\_jaffle\_shop.yml**

version: 2

models:

- name: stg\_customers

description: Staged customer data from our jaffle shop app.

columns:

- name: customer\_id

description: The primary key for customers.

tests:

- unique

- not\_null

- name: stg\_orders

description: Staged order data from our jaffle shop app.

columns:

- name: order\_id

description: Primary key for orders.

tests:

- unique

- not\_null

- name: status

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

tests:

- accepted\_values:

values:

- completed

- shipped

- returned

- placed

- return\_pending

- name: customer\_id

description: Foreign key to stg\_customers.customer\_id.

tests:

- relationships:

to: ref('stg\_customers')

field: customer\_id

**models/staging/jaffle\_shop/jaffle\_shop.md**

{% docs order\_status %}

One of the following values:

| status         | definition                                       |

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

| placed         | Order placed, not yet shipped                    |

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

| completed      | Order has been received by customers             |

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

| returned       | Item has been returned                           |

{% enddocs %}

**Generate and view documentation**

* Generate the documentation by running dbt docs generate.
* View the documentation that you wrote for the stg\_orders model.
* View the Lineage Graph for your project.

**Extra Credit**

* Add documentation to the other columns in stg\_customers and stg\_orders.
* Add documentation to the stg\_payments model.
* Create a doc block for another place in your project and generate this in your documentation.

**Review**

**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...

**Learning Objectives**

* Understand why it's necessary to deploy your project.
* Explain the purpose of creating a deployment environment.
* Schedule a job in dbt Cloud.
* Review the results and artifacts of a scheduled job in dbt Cloud.

**Review**

**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.