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.

dbt, data platforms, and version control

If you are new to dbt, the underlying architecture may be new to you. First, read this quick overview and then follow the steps for creating the necessary accounts. There are effectively two ways in which to use dbt: dbt CLI and dbt Cloud.

- **dbt Cloud** is a hosted version that streamlines development with an online Integrated Development Environment (IDE) and an interface to run dbt on a schedule.
- **dbt Core** is a command line tool that can be run locally.

This course will assume you are using dbt Cloud, but the concepts and practices can easily be extended to dbt CLI.

Data platform

dbt is designed to handle the transformation layer of the 'extract-load-transform' framework for data platforms. dbt creates a connection to a data platform and runs SQL code against the warehouse to transform data.

In our demos, we will be using Snowflake as our data warehouse. You will need access to your own data platform to complete the practice exercises. Read more in our documentation on dbt's supported databases.

Version control

dbt also enables developers to leverage a version control system to manage their code base. A popular version control system is git. If you are unfamiliar with git, don't worry, dbt Cloud provides a UI that makes it simple to use a git workflow.

In this course, we will be demoing with **GitHub** to host the code base that we build. If you would prefer to use another service for version control with dbt Cloud, check out our documentation for other version control options.

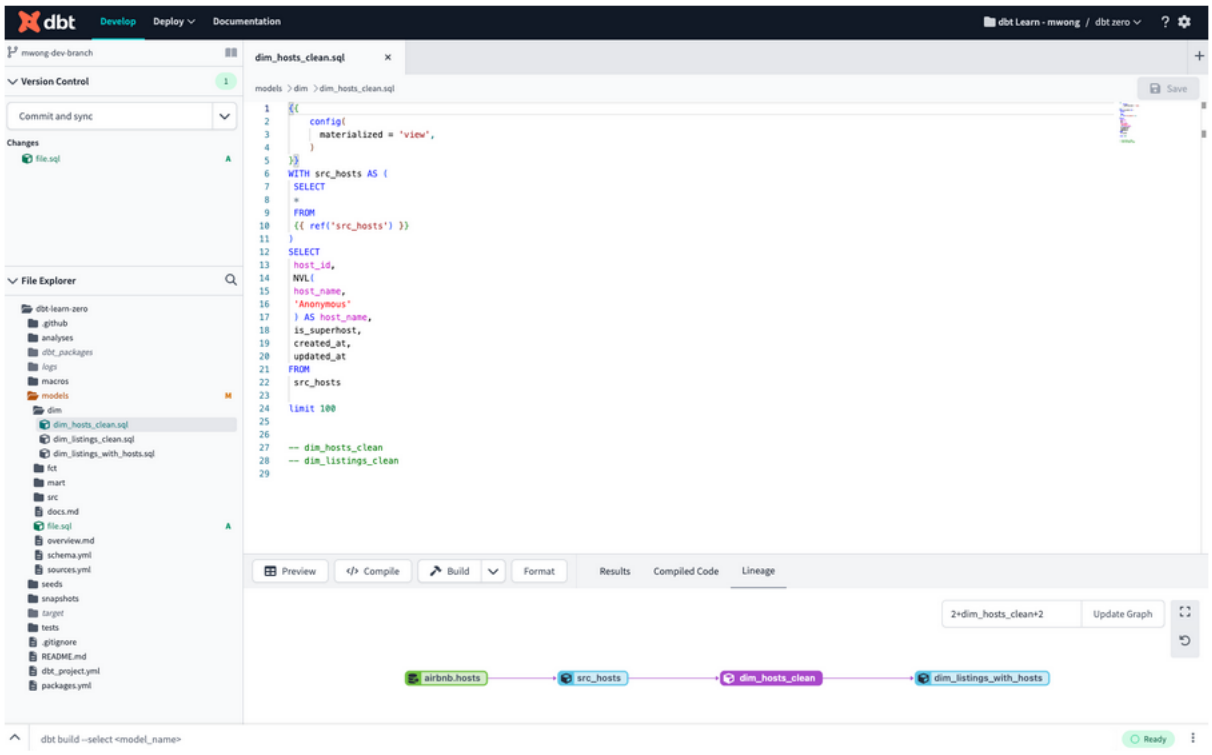
All in all, dbt is going to be the transformation interface between the code we write (stored and managed in a git repository) and the sample data we have to work with (stored and transformed in your data platform).

Now let's start setting up your individual accounts. In the following lessons, we will work to ensure all of these are connected appropriately.

Review

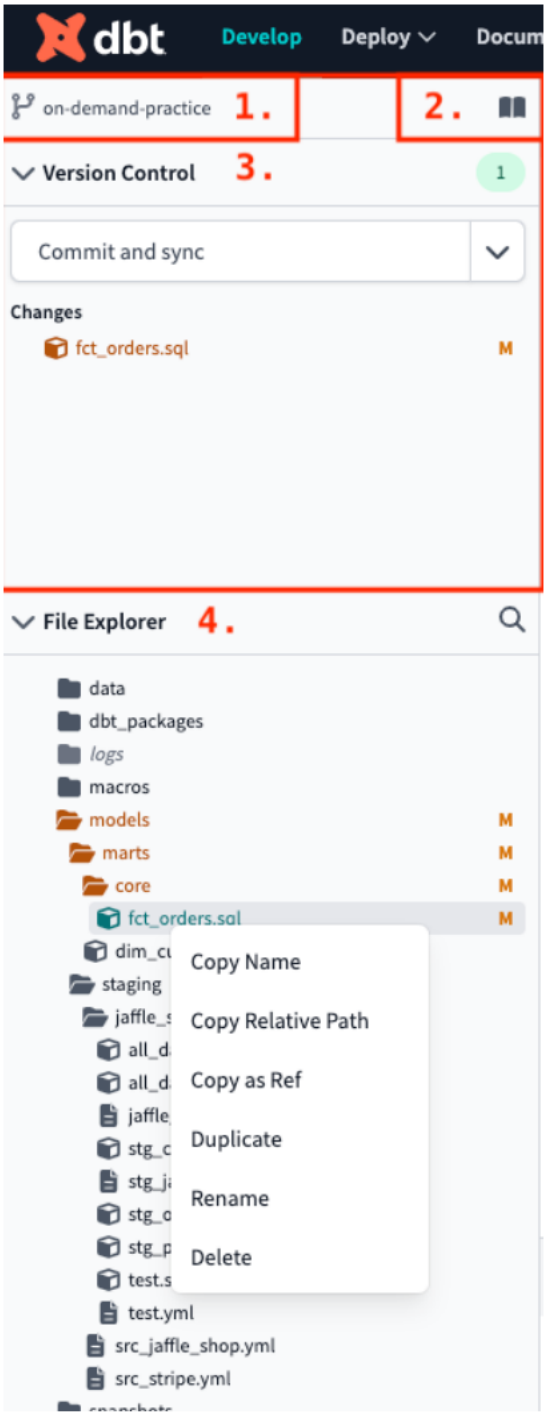
dbt Cloud IDE

The dbt Cloud integrated development environment (IDE) is a single interface for building, testing, running, and version-controlling dbt projects from your browser. With the Cloud IDE, you can compile dbt code into SQL and run it against your database directly



Basic layout

The IDE streamlines your workflow, and features a popular user interface layout with files and folders on the left, editor on the right, and command and console information at the bottom.



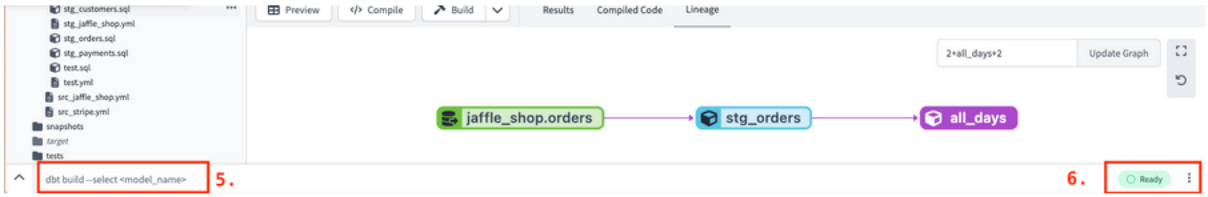
1. Git repository link — Clicking the Git repository link, located on the upper left of the IDE, takes you to your repository on the same active branch.

2. Documentation site button — Clicking the Documentation site book icon, located next to the Git repository link, leads to the dbt Documentation site. The site is powered by the latest dbt artifacts generated in the IDE using the dbt docs generate command from the Command bar.

3. Version Control — The IDE's powerful Version Control section contains all git-related elements, including the Git actions button and the Changes section.

4. File Explorer — The File Explorer shows the filetree of your repository. You can:

- Click on any file in the filetree to open the file in the File Editor.
- Click and drag files between directories to move files.
- Right click a file to access the sub-menu options like duplicate file, copy file name, copy as ref, rename, delete.
    - **Note:** To perform these actions, the user must not be in read-only mode, which generally happens when the user is viewing the default branch.
- Use file indicators, located to the right of your files or folder name, to see when changes or actions were made:
    - Unsaved (•) — The IDE detects unsaved changes to your file/folder
    - Modification (M) — The IDE detects a modification of existing files/folders
    - Added (A) — The IDE detects added files
    - Deleted (D) — The IDE detects deleted files



5.Command bar — The Command bar, located in the lower left of the IDE, is used to invoke dbt commands. When a command is invoked, the associated logs are shown in the Invocation History Drawer.

6. IDE Status button — The IDE Status button, located on the lower right of the IDE, displays the current IDE status. If there is an error in the status or in the dbt code that stops the project from parsing, the button will turn red and display "Error". If there aren't any errors, the button will display a green "Ready" status. To access the IDE Status modal, simply click on this button.

Explore Editing Features in the IDE

Review the Steps

Configure BigQuery for dbt Cloud

- Setting up BigQuery for dbt Cloud involves creating roles, adding principles, and working with projects
- Before you begin, make sure you have the BigQuery Admin or Owner role so you can complete these tasks
- To develop in dbt Cloud, it is recommended developers have two BigQuery projects, one for raw data and one for transformed data
- Developers will use a BigQuery service account to connect to dbt Cloud
- You will create and the service account in the BigQuery project where the transformed data will be stored
- The service account will need BigQuery Editor and BigQuery Job User roles
- The service account should also be added as a principle to the BigQuery project with the raw data
- The role to assign in the raw data project is BigQuery Data Viewer

Configure Github for dbt Cloud

- Once you have a Github account make sure you're invited to your team's Github organization
- Create a new repo using a name like `internal-analytics` or `dbt-project'
- Provide a quick description for the repo like "A dbt project for managing data transformations"
- Set the repo to private (you can make it public later)
- It is critical that you do not add a README file at this stage. This will be covered when you initialize your project later
- Leave other settings blank for now

Set Up a New dbt Cloud Project

- Login to BigQuery and Github in separate browser tabs before you begin
- In a third tab login to dbt Cloud
- Most people will login to dbt Cloud at cloud.getdbt.com. If you are on a virtual private cloud or based outside of the U.S., this will be a different link provided by your dbt account team
- In dbt Cloud, follow the **Complete project setup** flow
- Name the project 'Analytics' and select BigQuery as the warehouse
- Obtain a key from the BigQuery service account and upload the service account JSON file in dbt Cloud
- dbt Cloud will autopopulate settings for the environment
- It's important to note that each user will enter a schema for the dataset value. The standard recommended is dbt_firstinitiallastname
- Test your data platform connection, and follow the flow to connect Github and select the specific repository
- Click initialize dbt project to create a simple project structure to start building on
- Click commit and push and execute dbt run to see sample models

Review

Set Up a Production Environment

- Deployment environments are used for running code on a schedule and development environments are used for developing code
- A production environment is a deployment environment where developers can run jobs
- A production environment will let the team create a production deployment pipeline for testing and deploying to production
- dbt recommends creating a service account with a role that has access to read from raw data but not write
- The connection detail values for a production environment should always differ from the values in your development environment

Schedule a Job

- dbt Cloud Administrators can configure when and how jobs run in a production environment
- Administrators can set up daily jobs and select the days, and intervals for those jobs
- Running a simple daily job is one way to quickly test your data platform connection
- The audit log will maintain a record of who scheduled a job, when it was scheduled, and the status of the job

Set Up Folders by Data Maturity

- dbt Cloud Administrators can create folder structures to optimize dbt Cloud projects
- The maturity model is one structure used by some team
- You must create a new branch each time you create a new folder structur
- Staging folders are used to organize source conformed logic and store data before it's transformed
- Marts folders are used to organize business confirmed logic and combine staging models to roll up into key business concept
- Use a file named .gitkeep for each folder to help Git recognize an empty director
- Update the dbt_project.yml file and README file to align with the maturity model
- Commit changes to the folder structure with a descriptive message and a pull reques
- Once the pull request has been reviewed, merge the pull request and make sure the changes are reflected in the code and in the dbt Cloud IDE

Set Up Folders by Domain

- dbt Cloud Administrators can set folders up by domain
- This structure lays the groundwork for different teams to build future models
- Create a new branch each time you create a new folder structure
- Create team folders using folder names that apply to your business
- Add a .gitkeep file to each folder
- Update the dbt_project.yml file and README file to align with the model
- Commit changes to the folder structure with a descriptive message and a pull request
- Once the pull request has been reviewed, merge the pull request and make sure the changes are reflected in the code and in the dbt Cloud IDE

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

```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
```

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:

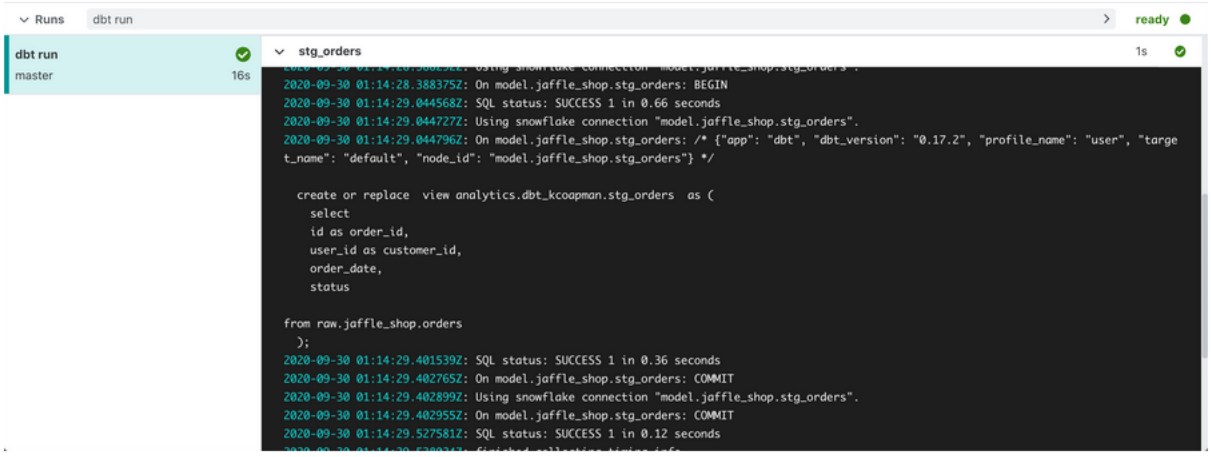
```
{{ config(
materialized='table'
) }}
```

- 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.

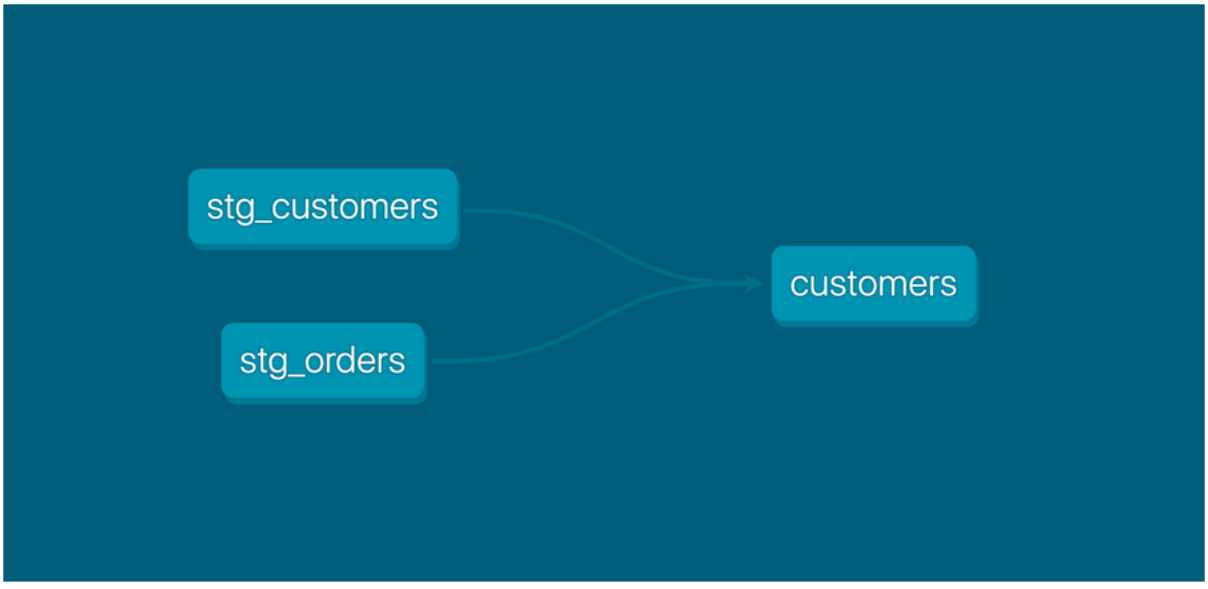


Modularity

- 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.

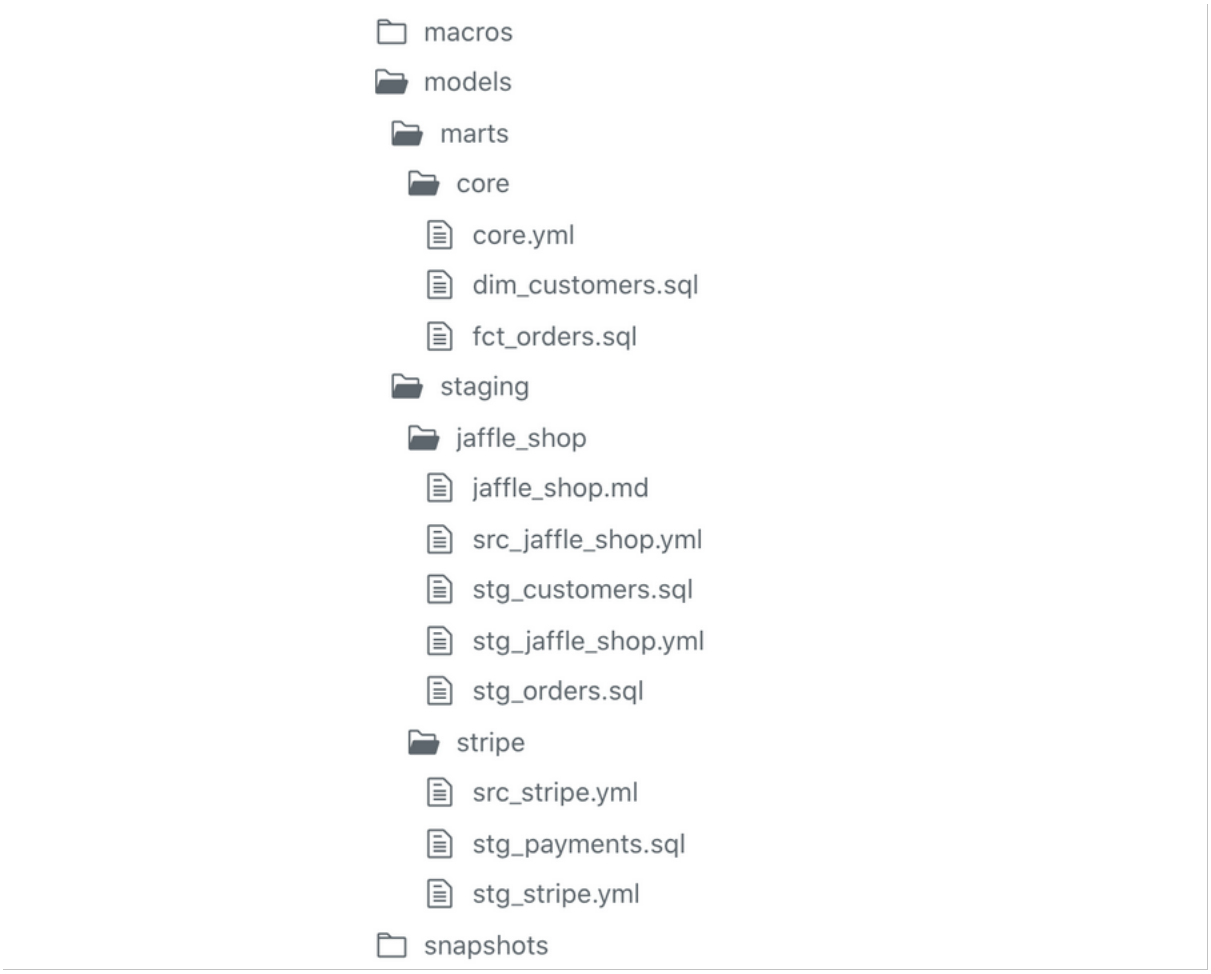
Naming Conventions

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)


macros
models
marts
core
core.yml
dim_customers.sql
fct_orders.sql
staging
jaffle_shop
jaffle_shop.md
src_jaffle_shop.yml
stg_customers.sql
stg_jaffle_shop.yml
stg_orders.sql
stripe
src_stripe.yml
stg_payments.sql
stg_stripe.yml
snapshots

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`.

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`

```yaml
version: 2

sources:
  - name: stripe
    database: raw
    schema: stripe
    tables:
      - name: payment
```

`models/staging/stripe/stg_payments.sql`

```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 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`.

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 singular 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

```yaml
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
```

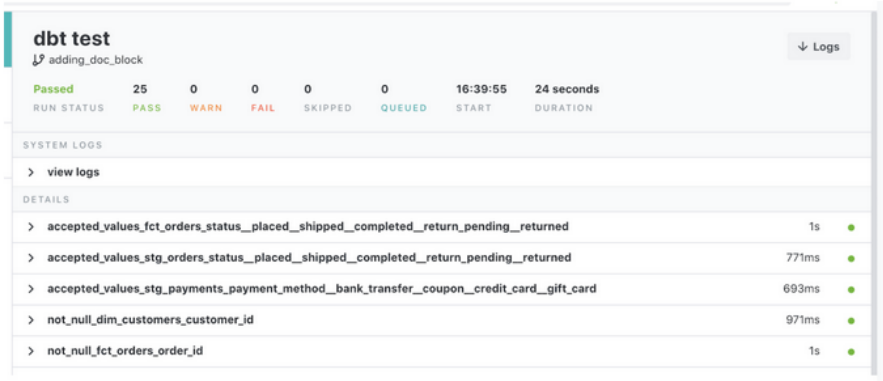
Review

Testing

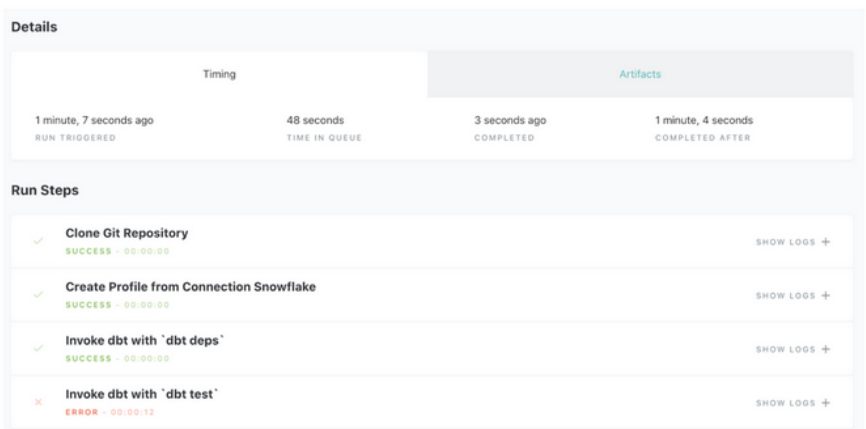
- **Testing** is used in software engineering to make sure that the code does what we expect it to.
- In Analytics Engineering, testing allows us to make sure that the SQL transformations we write produce a model that meets our assertions.
- In dbt, tests are written as select statements. These select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

Tests in dbt

- In dbt, there are two types of tests - generic tests and singular 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.
  - **Singular 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)
- Generic tests are configured in a YAML file, whereas singular 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.
- 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.



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



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...

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.

Congratulations!

via GIPHY

Thank you for joining all of us from the dbt Labs team!!! At this point, you have been empowered with the fundamentals of dbt - models, sources, tests, docs, and deployment.

Make sure you hit complete on each of the lessons (including this one!) and pass all of the Checks for Understanding to mark the course as complete and receive your dbt Fundamentals badge. When you mark all lessons complete, it may take a few seconds for your badge to be created.

Check out the resources below to continue the journey, stay fresh on your skills, and share this with your fellow analytics engineers.

Resources

dbt Docs: There is no shame in referencing the docs as an analytics engineer! Use this to continue your journey, copy YML code into your project, or figure out more advanced features.

Short courses: We have four courses to continue leveling up:

1. **Jinja, Macros, and Packages:** Extend dbt with Jinja/Macros and import packages to speed up modeling and leverage existing macros.
2. **Advanced Materializations:** So far you have learned about tables and views. This course will teach you about ephemeral models, incremental models, and snapshots.
3. **Analyses and Seeds:** Analyses can be used for ad hoc queries in your dbt project and seeds are for importing CSVs into your warehouse with dbt.
4. **Refactoring SQL for Modularity:** Migrating code from a previous tool to dbt? Learn how to migrate legacy code into dbt with modularity in mind.

Contribute

- Support fellow learners and let us know what you thought about the course in **#learn-on-demand**.
- Support other beginners in **#advice-dbt-for-beginners.**

Share

- Share the course with your team, on Twitter or LinkedIn!
- Add the **dbt Learn Fundamentals** badge to your LinkedIn profile.

Feedback

- **Feedback:** Let us know what you thought about the course with positive and constructive feedback. We are transparent about getting better, so don't hesitate to share in **#learn-on-demand**.
- **Bugs:** Help the dbt Labs training team squash bugs in the course by sending them to training@dbtlabs.com and we will triage them from there.

Congratulations and thank you again! See you in dbt Slack!