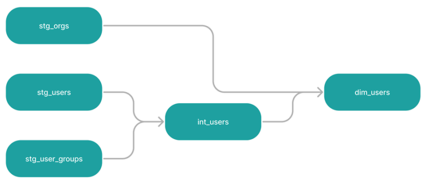
Data Build Tool (dbt) SME

What is dbt?

dbt is a tool for orchestrating multiple levels of transformations across a data warehouse. It includes templating and scripting options that allow for dynamic SQL definitions. There is also a powerful suite of testing options for ensuring data quality.

The DAG

Tables and views are constructed in order, starting with data closest to the raw input sources and moving outwards. The chain of dependence relations is often called the **"DAG"** *(**D**irected **A**cyclic **G**raph)*.



A simple DAG

The Project

By default, dbt projects are organized with these main folders:

- `models` - where you put specifications and SQL code for creating tables and views
- `tests` - where you put custom tests for your dbt models and data sources
- `seeds` - where you can place CSV files that are small enough to be version-controlled and are useful for your data transformations
- `macros` - where you put custom functions and scripts that evaluate to SQL or complete other tasks, callable from your SQL models and tests
- `target` - usually excluded from version control, where dbt places compiled objects at runtime
- `dbt_packages` - usually excluded from version control, where dbt extension packages are installed

Basic Commands

- `dbt compile` - compiles all models in your project, but doesn't run them in your warehouse. The compiled files can be found in the "target" folder.
- `dbt run` - compiles all models in your project and runs them to generate tables and views in your data warehouse.
- `dbt test` - compiles all tests in your project and runs them to generate tables and views in your data warehouse.
- `dbt build` - compiles all models **AND** tests in your project and runs them to generate tables and views in your data warehouse
- `dbt seed` - adds version-controlled CSV files to your data warehouse as tables for downstream use

Configuring dbt

For Paradigm specific instructions for connection to Databricks, please see DBT Databricks Quick Start (as of 9/2024)

Profiles

dbt runs SQL commands using resources from your data warehouse. For this reason, you must set configuration options in a "profiles.yml" file to tell dbt which resources should be used. For Databricks this includes the SQL warehouse, default catalog and schema, and authentication information for your Databricks account.

```yaml
galaxy:
  target: dev  # default output for dbt commands
  outputs:
    prod:
      database: prod
      schema: galaxy
    staging:
      database: staging
      schema: galaxy
    dev:
      database: dev
      schema: galaxy
```

Targets

You can set up multiple profiles for a dbt project. Each profile can have different configuration options. This means you can set up different connections/accounts for different profiles, or have multiple profiles operating in the same account but with different default parameters. This makes it easy to set up environments (dev, staging, prod, etc). When you want to run commands using a specific profile, just add `--target <profile_name>` to your command string.

```
# Examples:
dbt --target dev run   # runs using the "dev" profile
dbt build -t prod   # builds (run+test) using the "prod" profile
```

Data Sources

Before any transformations start happening, you need to set up source configurations for incoming data. This is done in a YAML file, usually placed in the staging sub-folder alongside the models that will use the source.

```yaml
# ./models/staging/jedi/_stg_jedi__sources.yml
version: 2

sources:
  - name: jedi
    database: raw
    schema: jedi
    tables:
      - name: candidates
```

dbt_project.yml

Most configurations of the data-containing outputs in your dbt project will be set here. This YAML file will contain sections for models, tests, and many other kinds of objects. The available configuration options will depend on the type of object, but typically include the catalog and schema where the object will be created. The tree of configurations for the objects in your file should match up with the folder structure of your project. By default, dbt organizes models into three folders:

- Staging
  - First pass to read from raw source tables, as specified in the source YAML files
  - Models should only depend on sources or other staging models
- Intermediate
  - Require more processing and/or incorporate widely re-used business logic.
  - Models should only reference staging models or other intermediate models
- Marts
  - Tables and views that will be used by other entities, such as BI teams and stakeholders.

```yaml
version: 2

models:
  staging:  # Models under directory './models/staging/'
    materialized: ephemeral  # see below for details on what this means

  intermediate:  # Models under directory './models/intermediate/'
    empire:
      schema: empire
    republic:
      schema: republic

  marts:
    materialized: table
    database: prod_analytics
    republic:  # configurations for models under directory './models/marts/republic/'
      schema: republic
      jedi:  # models under directory './models/marts/republic/jedi'
        schema: jedi  # overrides above schema config
    empire:  # models under directory '.models/marts/empire/'
      schema: empire
```

Materialization - tables, views, and more

By setting the `materialized` config option, you can choose how the result of a SQL model compilation will be run. You can set a default materialization in your `profiles.yml` file.

Materialization	Benefits	Drawbacks
table	transformations and calculations results will be saved, making the data quick to access.	will be recreated at each job run, which may waste resources when the underlying data does not change much
view	transformations and calculations will be re-run each time the view is queried, but build time will be relatively quick	resources will only be spent when the view is queried, but can get expensive and/or slow for larger and complex queries
incremental	transformations and calculations will only be run on new data, with changes merged into the table	the data needs to be properly timestamped and each entry needs a unique identifier if you want to update old entries rather than append new rows

ephemeral	References to the model will be replaced with a CTE, reducing clutter while allowing the code to be re-used.	The model will not be materialized anywhere, and if there are intensive calculations it may waste resources.

Designing your project - tables, tests, and more

Models

Models are the basic building blocks of your DAG. These will usually turn into tables and views that can be queried from your data warehouse. Models can be used to encapsulate business logic and publish data for consumption by end-users.

Sources and References

dbt models and tests should never make explicit references to other objects by catalog/schema, instead using the `source` and `ref` macros:

- `source('<source_name>', '<table_name>')` – using the source and table names from a sources YAML file, refers to the table/view in question based on the specified database and schema.
- `ref( '<model_name>' )` – using the filename of the model (minus the extension), refers to the explicit location of the referenced model using the database/schema defined in the project configs.

Bad Example:

```sql
-- ./staging/jedi/padawan_candidates.sql
select * from raw.jedi.candidates where midichlorians>5000
```

Good Example:

```sql
-- ./staging/jedi/padawan_candidates.sql
select * from {{ source('jedi', 'candidates') }} where force_sensitive=true
```

Macros and SQL Compilation

`source` and `ref` are the most common examples of dbt macros. When dbt compiles your code these macros are replaced by lines of SQL before dbt runs the commands in your warehouse.

Model Code

```sql
-- ./models/staging/empire/stg_empire__starships.sql

select
  {{
    dbt_utils.star(
      from=source('empire', 'starships'),
      except=["captain_id"]
    )
  }}
from {{ source( 'empire', 'starships' ) }}
```

Compiled Code

```sql
-- ./target/compiled/staging/empire/stg_empire__starships.sql

select
  id,
  ship_class,
  ship_name,
  ship_status
  -- captain_id excluded
from raw.empire.starships
```

Model Code

```sql
-- ./models/intermediate/jedi/padawan_candidates.sql

select
  id,
  first_name,
  last_name,
  age
from {{ ref( 'stg_jedi__padawans' ) }}
```

- Note that the use of an ephemeral model turns `stg_jedi__padawans` into a CTE instead of an explicit reference to a table/view in the warehouse.

Compiled Code

```sql
-- ./target/compiled/intermediate/jedi/padawan_candidates.sql

with __dbt__cte__stg_jedi__padawans as (
  select * from raw.jedi.candidates -- compiled from: {{source('jedi', 'candidates')}}
  where force_sensitive=true
)
select
  id,
  first_name,
  last_name,
  age
from __dbt__cte__stg_jedi__padawans
```

Tests

Tests are ways to ensure data quality and will be included in your DAG when running `dbt test` or `dbt build`. Tests are compiled and run as SQL in your warehouse, much like models.

Typically, a test will fail when running it returns at least one row. There are configuration options to set custom conditions and thresholds. You can also set a test to give a warning instead of a failure.

Singular Tests

Singular tests are custom-written by you to check a specific data quality issue. The code should be saved in the `./tests/` folder.

Singular tests are tests you write to catch specific data quality issues. Write the test definition the same way you would define a table or view. Tests will be automatically added to the DAG on job runs based on the sources and references used.

```sql
-- ./tests/council_rejects_test.sql
-- This test will fail if any rows are returned from executing the query
select
  *
from {{ ref( 'jedi_masters' ) }}
where full_name = 'Anakin Skywalker'
```

Generic Tests

Generic tests can be re-used on multiple models and even on sources. The most common ones are: `not_null` and `unique`. They are assigned at the column-level in a model YAML file.

```yaml
# ./models/staging/jedi/_stg_jedi__sources.yml
version: 2

sources:
  - name: jedi
    database: raw
    schema: jedi
    tables:
      - name: candidates
        columns:
          - name: id
            tests:
              - unique
              - not_null
```

```yaml
# ./models/intermediate/empire/_int_empire__models.yml
version: 2

tables:
  - name: empire_starships
    columns:
      - name: captain_id
        tests:
          - relationships: # captain_id must be one of following
              to: ref('empire_personnel')
              field: personnel_id

```

```yaml
# ./models/marts/republic/jedi/_marts_republic_jedi__models.yml
version: 2

tables:
  - name: jedi_knights
    columns:
      - name: lightsaber_color
        tests:
          - accepted_values: # might be one of the following
              values: ['green', 'blue', 'yellow', 'purple']

```

Seeds

Sometimes you may want to include a small table or list in your version control. This might be ID mappings or a list of values to be excluded from an output table. Seeds are CSV files that dbt will materialize as tables in your data warehouse, where they can be referenced like other dbt models.

Seed CSV files should be placed under the `./seeds/` directory.

To create the tables in your warehouse, run `dbt seed`. To reference them in a model, just use the `ref` macro with the name of the seed (minus file extension).

```
~ ./seeds/nicknames.csv
real_name, nickname
"Anakin Skywalker", "Ani"
"Obi Wan Kenobi", "Ben Kenobi"

```

```sql
with
jedi as (select * from {{ ref('jedi_knights') }}),
nicknames as ( select * from {{ ref( 'nicknames' ) }})

select
  jedi.*,
  nn.nickname
from jedi
left join nicknames nn on jedi.full_name = nn.real_name
```

Advanced dbt commands

We've already looked at the basic commands, `dbt run`, `dbt test`, `dbt build`, `dbt compile` as well as `dbt seed`. We've also seen that you can choose a target with `--target <target_name>` to switch between defaults profile configurations.

Now that we've gone over models, tests, and seeds, you can also limit which objects are included in the DAG for a given dbt task. This can be done by using the `--select` and `--exclude` options with your dbt commands.

- `--select <criteria>` will only include objects that meet the given criteria
- `--exclude <criteria>` will only exclude objects that meet the given criteria

Selection methods

Criteria for selecting/excluding objects from the DAG can be set using multiple properties that one or more objects may share.

method	example	explanation
fqn	`dbt test --select empire_starships`	The "fully qualified name" is the name of an object in the dbt DAG, usually it is the file name minus extension, the same name used with the 'ref' macro.
path	`dbt run --select "path:models/marts/jedi"`	You can choose all the objects in a specific folder path. Starting with "path:" is not necessary, but can reduce ambiguity.
tag	`dbt build --exclude "tag:incremental_models"`	Tags can be placed on objects in YAML configuration files or in a `{{config(...)}}` block on models and tests.
source	`dbt compile --select "source:jedi"`	This criteria is met by all objects that refer to tables from the named source.
state	`dbt run --select "state:modified"`	When multiple jobs are run on the same machine, dbt can compare against previous runs to determine which models have changed.

Selection operators

You can modify select/exclude criteria with one of several operators. This can simplify commands greatly. Here area a few examples:

example	explanation
`dbt compile --exclude source:jedi+`	This will exclude all models using the named source, as well as their descendants
`dbt run --select "+path:models/marts/republic"`	This will include all the models in the specified path along with all their ancestors.
`dbt build --select @empire_starships`	This will include descendants of the specified model, as well as ancestors of its descendants
`dbt test --select jedi_knights jedi_masters jedi_padawan`	By separating criteria with spaces, you select/exclude objects that meet any one of them. This will run tests for all three of the models.
`dbt build --select "path:models/intermediate/republic+,tag:incremental_models+"`	By separate criteria with a comma, you select/exclude objects that meet all of them. This will run all models and tests that have an ancestor meeting BOTH the path criterion AND the tag criterion.

Extra Fun

Jinja

Jinja is the templating language used by dbt to compile SQL models. Jinja is similar to python and other languages in that it can set/use variables, manage control flow (if-then-else, for-loops, etc), and define/call functions.

Jinja code that doesn't compile directly to SQL is wrapped in curly brackets with % symbols:

`{% set padawan_age_limit=8 %}`

When you want to include the Jinja output in the compiled SQL use doubled curly brackets:

`select * from padawan_candidates where age <= {{ padawan_age_limit }}`

Notice that macros are also called with doubled curly brackets, eg: `{{ref('jedi_masters')}}` because they are replaced with SQL

More command options

`--full-refresh`

Using `--full-refresh` with `run`, `build`, and `seed` commands will force the recreation of objects that aren't normally fully recreated.

For incremental models, it will drop and recreate the table, and is useful when the model's columns change or when logic needs to be updated.

For seeds, it will drop and recreate the table, where the default behavior is usually to truncate and refill.

For some warehouse providers views and models that change from view to table materializations (or vice versa) may need to be fully refreshed before the materialization change can take effect.

`--threads=N`

You can set the default number of threads for a dbt run in the profile configuration, but you can also set them at the command line. This is the maximum number of tasks that dbt will attempt to run concurrently. The actual number of concurrent tasks may be less, depending on your DAG and how long specific models/tests take to run. Increasing has an impact on overall runtime, but don't try to max out your CPU.

`--fail-fast and --no-fail-fast`

When using `--fail-fast`, any error or test failure will result in dbt stopping the entire job as soon as possible. The default behavior is usually for dbt to continue running tasks within the branches of the DAG that remain error-free.

`--warn-error`

This option will convert all test warnings into errors. It can be useful when you need more visibility on potential data quality issues that haven't been configured to give errors by default.