DBT Style and Best Practices Guide

**DBT**

**Project Structure:**

When referring to our project structure, we will not get into the decision making required about the final data model, i.e. Kimball vs Data Vault, etc. That is a decision that needs to be made during discussions with the client so that it can better fit their needs and their specific use case.

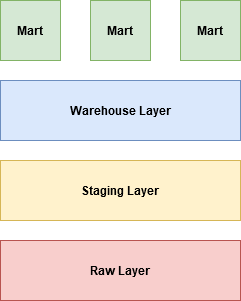
Instead, this section will cover how to organize the dbt project itself. What kind of models to create, where to apply the business logic, how to break out the transformations, and other similar things. A lot of this is derived from the current structure recommended by Fishtown Analytics, as well as community recognized best practices. It is also a living document – subject to change based on releases of new features or discoveries by users that would increase performance or comprehension.

These guidelines are influenced by the assumption that the client is using a data warehouse that is optimized for analytical queries, as well as the possibility that multiple data sources are being loaded by third party tools and not internally developed solutions. For situations where data is being loaded manually or by a homegrown tool that doesn’t neatly account for CDC, see the “snapshot layer” section.

**Data Transformations:**

The data in any project has three distinct checkpoints:

1. **Sources**: Schemas and tables in a source-conformed structure. This means that tables and columns are in a structure based on what the API or source returns. Frequently loaded with a third-party tool. These need to be identified in the sources.yml file in the model’s directory.
   1. **Snapshot Models:** If a third-party tool like FiveTran or Stitch is not being used and the client is interested in retaining history of the data beyond a static representation of the source, it is recommended that you create a snapshot layer that exists before the staging models. This would use the ‘snapshot’ materialization and it auto-creates a table that uses Type 2 SCD logic to retain row history. This would then be used as the “source” by the staging models. At its simplest form, we use this for mutable data, as opposed to immutable data.
2. **Staging Models:** this is the atomic unit of data modeling. Each model bears a one-to-one relationship with the source data table it represents. This means that it has the same granularity (no aggregations have been performed), but the columns may have been renamed, recast, or reconsidered into a consistent and useful format.
3. **(Optional) Warehouse Models:** Warehouse models can be used if the modelers want to create a warehouse layer where the data is stored in a Kimball or Data Vault style. The reasoning for this revolves around access and the flexibility of marts – if all the data is stored in a consistent and standard warehouse layer, it is easier to make changes to the mart models. They won’t be as dependent on atomic staging models and will instead rely on standardized warehouse models.
4. **Mart Models:** Mart models are models that represent business processes and entities. They’ve abstracted the data from the sources that they are based on.
   1. It is important to note the difference between the staging and mart checkpoints. Sources, snapshot, and staging are all source-centric models, whereas mart models are business-centric.



You may be wondering – why aren’t we performing all of our business logic in the staging models like we do in our ETL training? It’s a good question. First, we do this to take advantage of one of the key features of dbt, reducing complexity. Being able to establish relationships between models using jinja allows us to decrease the complexity contained within a single model, which increases the understandability of our solution. Second, it allows us to further ensure data validity. We can perform tests on our staging models to ensure that the data aligns with what we expect to see, especially before we aggregate it for analysis.

**Snapshot Models:**

Snapshot models are particularly helpful when you are dealing with a mutable source data tables. These are tables where records are updated in-place over time, like an orders table where the status of the order changes. This is opposed to immutable tables, where a record is never updated again after it is created.

Mutable sources, while great in the mind of the application developer, are difficult to work with as an analyst. We generally prefer to retain historical changes and account for them in business logic.

Instead of building a history table, or custom designing a snapshot process, we can use the snapshot materialization within dbt to accomplish the same purpose. This will allow us to retain all of the historical data while only persisting the active data forward into the data warehouse. Whenever a source table is mutable, we recommend using a snapshot.

If using Snowflake, we recommend using the following lines within your project yml file. This will ensure that the tables aren’t stored as transient, which will give you up to 90 days of time travel if you are in the enterprise edition. While this isn’t needed (CDC should capture fluctuating values) it is helpful just in case something happens.

snapshots:

  project name:

    transient: false

For more information, see the documentation: <https://docs.getdbt.com/docs/building-a-dbt-project/snapshots/>

**Staging Models:**

The goal of the staging layer is to create staging models, which take raw data and clean/prepare it for further analysis. If anyone were to query the data warehouse or analyze the lineage graph, they will understand that a model with the stg\_ prefix indicates that:

* Fields have been renamed and recast in a consistent way.
* Datatypes, such as time zones, are consistent.
* Light cleansing, removing unwanted characters or replacing empty strings with NULL values, has occurred.
* If useful or needed, flattening of JSON objects has occurred.
* There is a primary key that is both unique and not null. This has been tested.
  + Recommendation: While most databases have a sequential id that is assigned as a business key for each row (actor\_id = 1,2,etc.) this is not actually a helpful key because it doesn’t represent anything. Some individuals, such as the author of this documentation, believe it is better to create a business key that serves as the new primary key. It must be created from a column, or series of columns, that are always unique – this should be the subject of a dbt test! One method to do this is through the use of the *surrogate\_key* macro in dbt\_utils, which can be installed with the packages.yml file and running dbt deps.
  + Individuals may be asking – how can I track history if I’m not keeping the same primary key? You don’t need to because of the recommended use of A8

Staging models can have joins in them if it is required to add additional columns for context or enrichment. Rows can be added through unions or removed through filters. Natural keys can be deduplicated or hashed together to create a surrogate key. This is all to represent that staging models need to follow the guidelines in the Data Transformation section but they do not need to be a static match of a single table.

As for how this needs to be laid out in the project structure, it is considered best practice to create a new directory per source within the staging directory in the model directory, such as shown below:

└── models  
 ├── marts

├── staging  
 | ├── source a  
 | ├── source b

Each staging directory needs to contain, at a minimum:

* One staging model for each object that is useful for analytics:
  + Named (stg\_<source>\_<object>). The source distinction is only needed if there are multiple sources.
  + The recommended materialization is a view unless performance requires a table.
* A src\_<source>.yml file which contains:
  + Definitions, tests, and documentation around each model.
* A stg\_<source>.yml file which contains:
  + Tests and documentation for models in the same directory.

***Base Models:***

Sometimes a source will contain rough data that requires significant cleaning, correcting, and categorizing before it can be used in a staging model. If this is the case, we recommend using base models. Effectively, this is a series of layer of models that are placed before the staging models that are used to rename and standardize the data into a format that can be used. Each model should follow the same standard of the “Model Design” section.

These models should be stored in a nested ‘base’ subdirectory within the specific source directory inside of the staging directory. For example:

└── models  
 ├── marts

├── staging

| ├── marketo

| ├── base

| ├── model

We recommend that these models use the ephemeral materialization so that they are not exposed to any end users who may be querying the warehouse. We also recommend that they are testing in a base.yml file within the same directory, following the testing guidelines.

**Warehouse Models:**

Warehouse models should be used when the modelers want to integrate a Kimball or Data Vault style of data modeling before the creation of the Mart Models. This is recommended for especially complex data models, or situations where there needs to be a secondary ‘area of truth’ that analysts will have access to. Instead of deconstructing the mart models to answer their questions, they can see the overarching data model and derive their answers from their own SQL queries.

└── models  
 ├── marts

├── staging

├── warehouse

If using Data Vault, there exists an open source project called dbtvault. It has pre-defined macros and packages that will standardize the creation of hubs, links, and satellites. Given that it is open source, it is frequently changing, with added functionality and improved performance. Versions can be locked for production systems so that there aren’t dependency issues. The link is: <https://dbtvault.readthedocs.io/en/latest/>

**Mart Models:**

Marts are stores of models that describe business entities and processes. They are often grouped by business unit – marketing, finance, product, etc. However, there are also frequently models that are shared across an entire business. These should be grouped in a core directory. Following this example, our directory would look like below:

└── models  
 ├── marts  
 | ├── core  
 | ├── finance  
 | ├── marketing  
 | └── product

While the format of the mart models are entirely dependent on many different factors, the goal remains the same; build fact and dimension models that help us abstract and make sense of the source data. Here are possible descriptions of the model types (these are not representative of all facts or dimensions):

* fact\_<verb>: A tall, narrow table representing real-world processes that have occurred or are occurring. The foundation of these models are an immutable event stream, be that sessions, transactions, orders, stories, votes, etc.
* dim\_<noun>: A wide, short table where each row is a person, place, or thing. They are the ultimate source of truth for identifying or describing entities related to the organization. The sources are mutable, possibly slowly changing.

The work of staging models is focused on cleaning and preparing the data. Fact tables are the product of substantive data transformation: choosing (and reducing) dimension, date spinning, executing business logic, and making informed decisions with the subject matter experts.

This modeling layer is significantly more complex than creating staging models. The models we design must be highly tailored to the analytical needs of the organization. This means that there are very few principles that can be applied across every industry and every customer. Below are a few common recommendations:

* Fact and dimension models should be materialized as tables within a warehouse in order to improve query performance. Where performance requires it, consider using the incremental materialization.
* Intermediate transformations may be required to get to the needed level of detail. These should be placed in a nested directory within the mart specific directory. In order to prevent end users from referencing these as a source of truth, consider using the ephemeral materialization. (*There may be less need for intermediate transformations if using a warehouse layer. This is dependent, however, on the specific transformation of the model.)*
* Models should be tested and documented in the <dir\_name>.yml file within the same directory as the models.
* Any extra documentation that is needed can be placed in a <dir\_name>.md file within the same directory, using doc blocks to reference it in the yml file.

**Model Design:**

Based on the Fishtown Analytics recommendations for best practices, we follow their SQL Style Guide, which heavily uses CTE’s in developing models. Any worry about CTE performance optimization doesn’t really apply in the development of DBT models, especially if used in the recommended way. So you get all the benefits of CTEs – easier to read and debug – and none of the downsides.

The first stylistic choice that will be considered A8 standard is to begin each model by “importing” each upstream relation that is going to be used. Think of this as the same as importing python packages – we need to lay out our dependencies early on. It makes it easy for future developers to see what the dependencies of that specific model are and it gives us an easy method to alias all dependency tables. See the “CTE Standards” subsection for added detail on CTE usage.

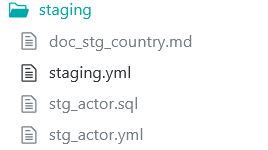
Example of “importing” the data from an events staging model:

*With events as (*

*select \* from {{ ref(‘stg\_events’) }}*

*)*

The second is that every subdirectory should contain a ‘model’ yml file that is named after the model. For example, a subdirectory called base with the base\_actor.sql file should have the base\_actor.yml file. This file will contain all of the documentation and testing for that model. To clarify, **we believe that there should be a yml schema file for each and every model.** This ensures readability. Below is an example of the file format. Notice that stg\_actor has both the sql model file and the yml file for tests and descriptions.



As a baseline, unique and not\_null tests need to be applied to the primary key of each model. This is the absolute minimum testing required for each model.

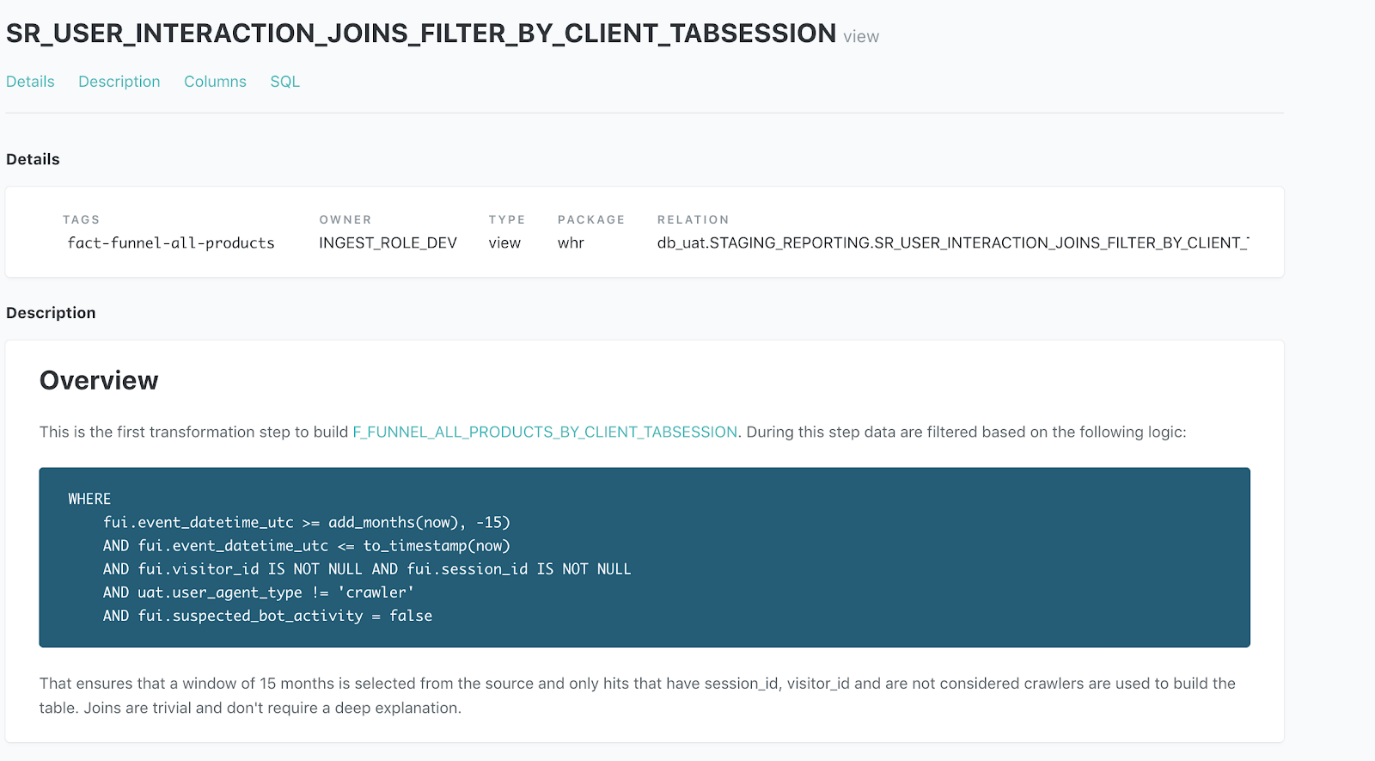
Finally, we will only reference our sources in the first layer of models, which will be stg\_ models unless base models are required by the project. All other models should only select from other models, never pulling from the source.

**Documentation:**

Documenting the process is incredibly important – it adds value by increasing the speed by which a new user can get up to speed and allowing analysts to walk through every step in order to understand the output. With those in mind, the following guidelines are considered best practice for A8 documentation:

* **Every table that is materialized (not ephemeral) and accessible to analysts or end users must have every field described**. The table description can be described in-line (using the yml file) or in a markdown document.
* **Every** **intermediate, warehouse, or mart model that performs some sort of transformation must contain an inline description, or markdown document, of the step**. If the step is non-trivial or important to a KPI, a markdown document must be used and must contain significant detail so that the end user can understand.

Below is an example of what a correct description would look like for a model that is performing a transformation. The logic of what data it is being applied to should be included in the description, as well as a description of the transformation,



**Tests:**

To debug a failing test, find the SQL that dbt ran by:

dbt Cloud:

Within the test output, click on the failed test, and then select "Details"

dbt CLI:

Open the file path returned as part of the error message.

Navigate to the target/compiled/schema\_tests directory for all compiled test queries

Copy the SQL into a query editor (in dbt Cloud, you can paste it into a new Statement), and run the query to find the records that failed.

**Best Practices:**

Below is a list of best practices that have been taken from Fishtown Analytics and development by Analytics8 consultants. Please keep them in mind on any project that you may be staffed on.

**Always use the ref function**

The ref functions is what makes dbt such an interesting and powerful tool. It allows the tool to infer dependencies and ensures that models are built in the correct order. Always use the ref function when selecting from another model – never use the direct relation reference to the database object itself.

**Limit references to raw data – only use source jinja functionality**

Following the guidelines in the ‘Staging Models’ section, your models should limit the reference to the raw data. This data, if loaded by third parties, can change over time – tables and columns may be added, removed or renamed. When this happens, it is significantly easier to debug and update if using the source jinja function.

**Break complex models up into smaller pieces**

It is considered A8 dbt best practice to use CTEs. As such, a complex model can include multiple CTEs. In dbt you can separate these CTEs into separate models that build on top of each other. It is recommended to do this when any of the following are true:

* A CTE is duplicated across two or more models.
* A CTE changes the gran of the data it selects from. It can be helpful to test the transformations in a model independent of the larger model.
* The SQL query contains too many lines. Breaking CTEs into separate models increases reduces the complexity of the code and allows other users to more easily understand your code.

**Group your models in the recommended directory structure**

A good amount is written in the Project Structure section about the proper structure of directories. It may not seem important to a new user, but it becomes increasingly important as the project size increases. Proper use of directories allows developers to run specific subsections of the DAG, communicate modeling steps to other developers, and even create project specific conventions around upstream dependencies.

**Consider the best materialization for your model**

Materializations determine the way that models are built. We recommend generally applying the same type for all models in the same directory, as it helps keep things consistent. This can change, however, depending on the needs of the model. Here are guidelines for which materializations to use:

* Use views by default. They are faster to build, but slower to query when compared to tables.
* Use ephemeral models for lightweight transformations that shouldn’t be exposed to end-users.
* Use tables for models that are queried by BI tools. Also use tables for models that have multiple descendants, as it increases performance.
* Use incremental models when build time for table models exceeds the acceptable time. Be careful, however, as they introduce added complexity into a project in that they persist through runs and are harder to update for new fields.

**Use hooks to manage privileges on objects created by dbt**

Hooks are SQL code that can run before or after a model has run. Use these in conjunction with grant statements to ensure that permissions are applied to the objects are created by dbt. By codifying these grant statements, we can more easily version control and apply permissions in the same area that we perform development.

**Style**

One of the hardest parts of any project is deciding what to name things. Surprisingly, it is also one of the most important parts of development. Standard names allow non-developers, new analysts, and business users to easily understand what the specific piece of information is representing. As such, A8 will follow the recommendations of Fishtown Analytics, the creators of dbt, when it comes to naming models and columns.

**Naming and Field Conventions:**

* Schemas, table, and columns should be named in snake\_case
* Names should be based on *business* terminology as opposed to source terminology. For example, SCF001 in the source would be renamed to stmt\_cash\_flow
* Table, and model, names should be plurals, e.g. accounts as opposed to account.
* Each model should have a primary key. If one does not exist, create a primary key as a combination of values that represent that unique record.
* The primary key of a model should be named <object>\_id. For example, account\_id or inventory\_id. This makes it easier to reference in downstream models.
* Timestamp columns should be named <event>\_at, e.g. created\_at or updated\_at. If not natively in UTC, they should be converted as soon as possible. If non UTC time zones need to be retained, denote the time zone in the name as a suffix, e.g. created\_at\_est.
* Booleans should be prefixed with an is\_ or has\_.
* Price/revenue fields should be in decimal currency (19.99 for $19.99). If non-decimal currency is used, indicate with a suffix, e.g. price in cents (1999).
* Use the same field names across models wherever possible. Customer\_id should always be customer\_id if it relates to the customers table, never changed to user\_id or anything like that.

**SQL Style Guide:**

Keeping naming and field conventions consistent across all models and projects is important, but that doesn’t mean we can forget about the actual SQL that we are writing. In order to allow for ease of reading and consistency among all developers, the following are the commandments of the A8 SQL Style Guide:

* Lines of SQL should be no longer than 80 characters.
* Field names and function names should all be lowercase. Certain databases will store these as all capitalized, which is fine. Querying for field and function names are not case sensitive.
* The ‘AS’ keyword should be used when aliasing a field or table.
* All relevant fields should be stated before any aggregate or window functions.
* Specify join keys – do not use the ‘using’ language. Certain warehouses, specifically Snowflake, have inconsistencies with this language.
* Use Union all over union \*
* Use lines to create visual breaks for separate calculations. Think of this as creating blocks for calculations like case statements or aggregations
* Have lines in between joins to allow for ease of understanding
* Always alias table with descriptive names, not initials like c for customers.
* Avoid table aliases in join conditions.
* If joining two or more tables, ***always*** prefix your column names with the table alias.
* Indents should be four spaces

**CTE Standards:**

* Names should be as verbose as needed to convey what it is that they do. Clarification is superior to brevity.
* If there is confusing logic, or particularly complicated logic in the CTE, it should be commented to explain the purpose.
* If a CTE is being duplicated across multiple models, it should be pulled out and added into its own model that can be referenced.

**Required Documentation:**

Getting Started: <https://docs.getdbt.com/tutorial/setting-up>

* This is great documentation for getting started with dbt\_cloud or the dbt CLI. I recommend going through these videos and reading the steps to see how Claire creates a new dbt project and handles the basic steps of a dbt project

Models: <https://docs.getdbt.com/docs/building-a-dbt-project/building-models>

* Read the dbt models section and the section on materializations to better understand what it is that the models are doing and how they are being created.

Documentation: <https://docs.getdbt.com/docs/building-a-dbt-project/documentation>

* Read the dbt documentation section to understand how to create dbt documentation for your project

Tests: <https://docs.getdbt.com/docs/building-a-dbt-project/tests>

* Read the dbt documentation to understand how to apply tests in your .yml files or even create your own tests

Snapshots: <https://docs.getdbt.com/docs/building-a-dbt-project/snapshots>

* Read the dbt documentation to understand how to create snapshot models, which are separate from models.

Sources: <https://docs.getdbt.com/docs/building-a-dbt-project/using-sources>

* Read the dbt documentation to understand how to define and reference sources.

**Recommended Reading:**

Packages: <https://docs.getdbt.com/docs/building-a-dbt-project/package-management>

Why NOT To Use Auto-Incrementing ID: <https://discourse.getdbt.com/t/generating-an-auto-incrementing-id-in-dbt/579>

Understanding Idempotent Transformation: <https://discourse.getdbt.com/t/understanding-idempotent-data-transformations/518>

Fishtown Style Explanation: <https://discourse.getdbt.com/t/why-the-fishtown-sql-style-guide-uses-so-many-ctes/1091>