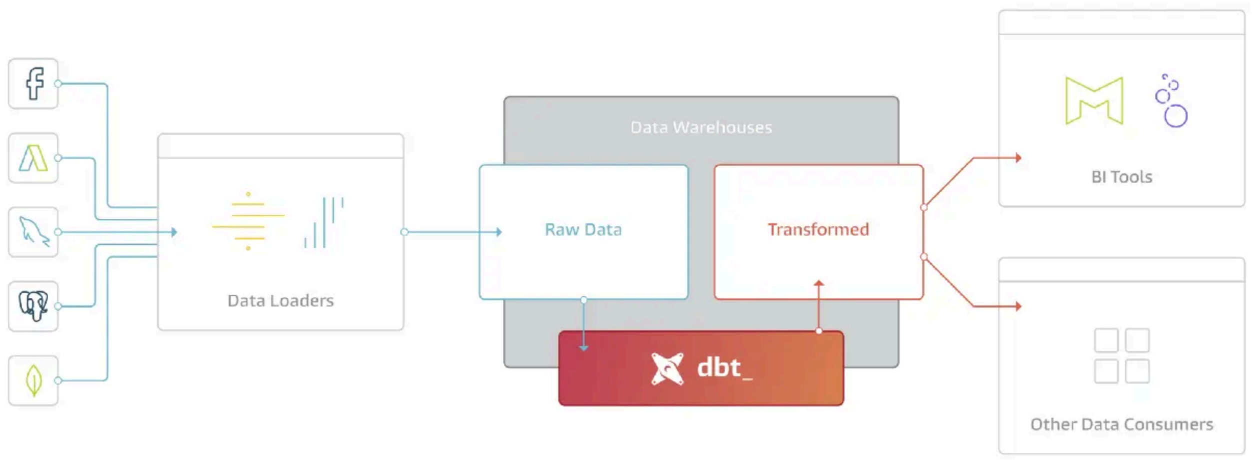
Lab



dbt is a command-line tool that enables data analysts and engineers to transform data in their warehouses more effectively.

dbt is the T in ELT. It doesn't extract or load data, but it's extremely good at transforming data that's already loaded into your warehouse. This "transform after load" architecture is becoming known as ELT (extract, load, transform).

Project

We are a fictional restaurant called the Jaffle Shop that serves jaffles.

We have data of customers, orders, and payments.

It has two main data sources:

- A replica of our transactional database, called `jaffle_shop`, with core entities like orders and customers.
- Synced data from Stripe, which we use for processing payments.

Installation

Use github codespaces with an empty repo that just contains `.devcontainer.json` and `requirements.txt` files.

my_dbt_duckdb_project-main.zip contains those files

`.devcontainer.json` will let the codesapce to install requirements once you open it

```
{
  "name": "My Project",
  "image": "mcr.microsoft.com/devcontainers/python:3.11",
  "postCreateCommand": "pip install -r requirements.txt"
}
```

```
agate==1.9.1
annotated-types==0.7.0
attrs==25.3.0
babel==2.17.0
certifi==2025.1.31
cffi==1.17.1
charset-normalizer==3.4.1
cli_helpers==2.4.0
click==8.1.8
colorama==0.4.6
configobj==5.0.9
daff==1.3.46
dbt-adapters==1.14.3
dbt-common==1.16.0
dbt-core==1.9.3
dbt-duckdb==1.9.2
dbt-extractor==0.5.1
dbt-semantic-interfaces==0.7.4
deepdiff==7.0.1
duckcli==0.2.1
duckdb==1.2.1
idna==3.10
importlib-metadata==6.11.0
isodate==0.6.1
```

```
Jinja2==3.1.6
jsonschema==4.23.0
jsonschema-specifications==2024.10.1
leather==0.4.0
MarkupSafe==3.0.2
mashumaro==3.14
more-itertools==10.6.0
msgpack==1.1.0
networkx==3.4.2
ordered-set==4.1.0
packaging==24.2
parsedatetime==2.6
pathspec==0.12.1
prompt_toolkit==3.0.50
protobuf==5.29.4
pycparser==2.22
pydantic==2.10.6
pydantic_core==2.27.2
Pygments==2.19.1
python-dateutil==2.9.0.post0
python-slugify==8.0.4
pytimeparse==1.1.8
pytz==2025.2
PyYAML==6.0.2
referencing==0.36.2
requests==2.32.3
rpds-py==0.24.0
setuptools==78.1.0
six==1.17.0
snowplow-tracker==1.1.0
sqlparse==0.5.3
tabulate==0.9.0
text-unidecode==1.3
typing_extensions==4.13.0
urllib3==2.3.0
wcwidth==0.2.13
```

```
wheel==0.45.1
zipp==3.21.0
```

Steps:

The whole project can be found in dbt_jaffle_shop-main.zip

1. Initiate a dbt project

```
dbt init jaffle_shop_project
1
```

it will ask you which database adapter you're using and since we installed the dbt-duckdb it will show it if we want to choose and we will choose it.

The `dbt init` command automatically creates a project with some folders and subfolders

2. go to the my_dbt_duckdb_project folder

```
cd jaffle_shop_project
```

3. explain the project structure

https://docs.getdbt.com/docs/build/projects

4. In the seed directory, we will put our CSV data

data can be found in seed_data.zip

1. we will start building our models

In dbt, we work with **models, which is a sql file with a select statement**. These models can depend on other models, have tests defined on them, and can be created as tables or views.

5.1 In the sub-folder models/ delete example and create a new folder called `staging`

```
rm -r example
mkdir staging
```

```
cd staging
```

5.2 in staging we mainly select from the raw data

create `stg_customers.sql` , `stg_orders.sql` and `stg_payments.sql` files

```
touch stg_customers.sql stg_orders.sql stg_payments.sql
```

Write at stg_customers.sql :

```
with source as (

    {#-
    Normally we would select from the table here, but we are using seeds to load
    our data in this project
    #}
    select * from {{ ref('raw_customers') }}

),

renamed as (

    select
        id as customer_id,
        first_name,
        last_name

    from source

)

select * from renamed
```

Write at stg_orders.sql:

```
with source as (

    {#-
    Normally we would select from the table here, but we are using seeds to load
    our data in this project
    #}
    select * from {{ ref('raw_orders') }}

),

renamed as (

    select
        id as order_id,
        user_id as customer_id,
        order_date,
        status

    from source

)

select * from renamed
```

Write at stg_payments.sql:

```
with source as (

    {#-
    Normally we would select from the table here, but we are using seeds to load
    our data in this project
    #}
    select * from {{ ref('raw_payments') }}
),

renamed as (

    select
        id as payment_id,
        order_id,
        payment_method,

        -- `amount` is currently stored in cents, so we convert it to dollars
        amount / 100 as amount

    from source

)

select * from renamed
```

Then add a configuration for those staging SQL models by using the schema.yml file

```
touch schema.yml
```

at schema.yml we configure properties for the models:

```
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: ['placed', 'shipped', 'completed', 'return_pending', 'returned']

  - name: stg_payments
    columns:
      - name: payment_id
        tests:
          - unique
          - not_null
      - name: payment_method
        tests:
          - accepted_values:
              values: ['credit_card', 'coupon', 'bank_transfer', 'gift_card']
```

5.3 at models/ we create 3 files

```
cd ..
touch customers.sql orders.sql schema.yml
```

Write the following into each corresponding file:

```
with customers as (

    select * from {{ ref('stg_customers') }}

```

```
),

orders as (

    select * from {{ ref('stg_orders') }}

),

payments as (

    select * from {{ ref('stg_payments') }}

),

customer_orders as (

        select
        customer_id,

        min(order_date) as first_order,
        max(order_date) as most_recent_order,
        count(order_id) as number_of_orders
    from orders

    group by customer_id

),

customer_payments as (

    select
        orders.customer_id,
        sum(amount) as total_amount

    from payments
    left join orders on
         payments.order_id = orders.order_id

    group by orders.customer_id

),

final as (

    select
        customers.customer_id,
        customers.first_name,
        customers.last_name,
        customer_orders.first_order,
        customer_orders.most_recent_order,
        customer_orders.number_of_orders,
        customer_payments.total_amount as customer_lifetime_value

    from customers

    left join customer_orders
        on customers.customer_id = customer_orders.customer_id

    left join customer_payments
        on  customers.customer_id = customer_payments.customer_id

)

select * from final
```

```
{% set payment_methods = ['credit_card', 'coupon', 'bank_transfer', 'gift_card'] %

with orders as (

    select * from {{ ref('stg_orders') }}
```

```
),

payments as (

    select * from {{ ref('stg_payments') }}

),

order_payments as (

    select
        order_id,

        {% for payment_method in payment_methods -%}
        sum(case when payment_method = '{{ payment_method }}' then amount els
        {% endfor -%}

        sum(amount) as total_amount

    from payments

    group by order_id

),

final as (

    select
        orders.order_id,
        orders.customer_id,
        orders.order_date,
        orders.status,

        {% for payment_method in payment_methods -%}
        order_payments.{{ payment_method }}_amount,

        {% endfor -%}

        order_payments.total_amount as amount

    from orders


    left join order_payments
        on orders.order_id = order_payments.order_id

)

select * from final
```

Add documentation for order status

```
touch models/docs.md
```

```
{% docs orders_status %}

Orders can be one of the following statuses:

| status         | description
|----------------|------------------------------------------------------------------
| placed         | The order has been placed but has not yet left the warehouse
| shipped        | The order has ben shipped to the customer and is currently in tra
| completed      | The order has been received by the customer
| return_pending | The customer has indicated that they would like to return the o
| returned       | The order has been returned by the customer and received at the
```

```
{% enddocs %}
```

```
version: 2

models:
  - name: customers
    description: This table has basic information about a customer, as well as som

    columns:
      - name: customer_id
        description: This is a unique identifier for a customer
        tests:
          - unique
          - not_null

      - name: first_name
        description: Customer's first name. PII.

      - name: last_name
        description: Customer's last name. PII.

      - name: first_order
        description: Date (UTC) of a customer's first order

      - name: most_recent_order
        description: Date (UTC) of a customer's most recent order

      - name: number_of_orders
        description: Count of the number of orders a customer has placed

      - name: total_order_amount
        description: Total value (AUD) of a customer's orders

  - name: orders
```

```
    description: This table has basic information about orders, as well as some der

    columns:
      - name: order_id
        tests:
          - unique
          - not_null
        description: This is a unique identifier for an order

      - name: customer_id
        description: Foreign key to the customers table
        tests:
          - not_null
          - relationships:
              to: ref('customers')
              field: customer_id

      - name: order_date
        description: Date (UTC) that the order was placed

      - name: status
        description: '{{ doc("orders_status") }}'
        tests:
          - accepted_values:
              values: ['placed', 'shipped', 'completed', 'return_pending', 'returned']

      - name: amount
        description: Total amount (AUD) of the order
        tests:
          - not_null

      - name: credit_card_amount
        description: Amount of the order (AUD) paid for by credit card
        tests:
          - not_null
      - name: coupon_amount
        description: Amount of the order (AUD) paid for by coupon
        tests:
          - not_null

      - name: bank_transfer_amount
        description: Amount of the order (AUD) paid for by bank transfer
        tests:
          - not_null

      - name: gift_card_amount
        description: Amount of the order (AUD) paid for by gift card
        tests:
          - not_null
```

6. edit `dbt_project.yml` file

edit the models section to be as follows:

```
models:
  jaffle_shop_project:
    materialized: table
    staging:
      materialized: view
```

- **Materialization** in dbt determines how the SQL query for a model is executed and stored in the database:
  - `table`: The query results are stored as a physical table in the database.
  - `view`: The query is stored as a database view (logical, not physical storage).

7. create profile

   ```
   touch profiles.yml
   ```

If you're using dbt Core, you'll need a `profiles.yml` file that contains the connection details for your data platform. When you run dbt Core from the command line, it reads your `dbt_project.yml` file to find the `profile` name, and then looks for a profile with the same name in your `profiles.yml` file. This profile contains all the information dbt needs to connect to your data platform.

Write at `profiles.yml`

```
jaffle_shop_project:

  target: dev
  outputs:
    dev:
      type: duckdb
      path: 'jaffle_shop_project.duckdb'
      threads: 24
```

Now we have created all the needed files and configs and will need to run some dbt commands to run our project

1.

```
dbt debug
// make sure that Connection test: [OK connection ok]
// profiles.yml file [OK found and valid]
// dbt_project.yml file [OK found and valid]
```

2. Load the CSVs using `dbt seed` command, which materializes the CSVs as tables in your target schema. Note that a typical dbt project does not require this step since dbt assumes your raw data is already in your warehouse.

3. `dbt run`
4. `dbt test`

NOTE You can run `dbt build` right away that will do seed, run and test for you

4. Query from the duck db

```
SELECT table_name, table_type
FROM INFORMATION_SCHEMA.TABLES
```

```
duckcli jaffle_shop_project.duckdb
select * from customers where customer_id = 42;
```

5. Generate and view the documentation for the project:

   ```
   dbt docs generate
   dbt docs serve
   ```