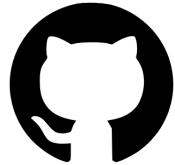


Repository & Slides



codecentric/dbt-workshop

- have python installed
- have dbt installed
- have duckdb installed
- + your favorite IDE/editor

A graphical database-client might be helpful



@codecentric

Data pipelines, Documentation and Lineage with SQL & dbt

Schedule

13:30 1. Block

- Short introduction dbt
- Connect to a database
- dbt sources & models
- Documentation & lineage
- Plugins

15:30 Break

16:00 2. Block

- Data Testing
- Unit Testing
- Macros
- Deployment, CI/CD, Production

Stretch Goals:

- Model Contracts & Versions
- Incremental Models
- Linting, Formatter

18:00 End





Google
BigQuery



Step 2: Create database connection

```
[9] ⌂ ▶ 0.1s  
import os
```

```
connection = mysql.connector.connect(  
    host=os.environ  
    user=os.environ  
    passwd=os.environ  
    database=os.env
```

def connect(*args: Any,
 **kwargs: Any) -> None

Create or get a MySQL connection object
In its simplest form, Connect() will open a connection to a MySQL server and return a MySQLConnection object.
When any connection pooling arguments are given, for example pool_name or pool_size, a pool is created or a previously one is used to return a PooledMySQLConnection.
Returns MySQLConnection or PooledMySQLConnection.

Step 3: Run Pandas

```
[14]
```

```
df = pd.read_sql_query("select * from datasources.gpu_model_data", con = connection)  
df
```

Table Visualize



untitled1.py

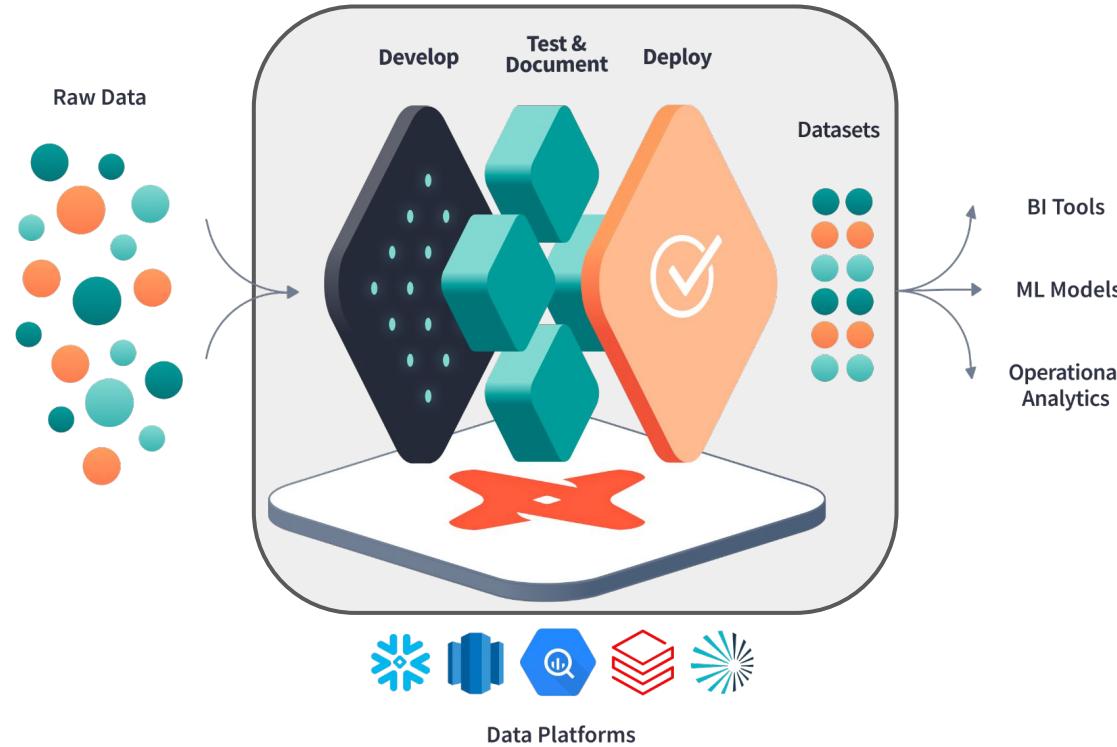
```
1 query = """  
2 SELECT *  
3 FROM employee e1  
4 WHERE salary >  
5     (SELECT AVG(salary)  
6      FROM employee e2  
7      WHERE e1.department_id = e2.department_id)  
9 """  
10 result = pd.read_sql(query)  
11 result[["first_name", "last_name", "salary"]]
```

Why is it bad?

- high degree of flexibility in the solution
- tendency to do “solved” work
- no standards
- complex to integrate
- complex to extend
- complex to maintain



“SQL Data Transformations with Software Engineering Best Practices”





- The T in ELT - running on the data in the warehouse
 - A dataset centric approach for dashboards, reports and analysis
 - Code & Configuration
 - Not a query engine!
-
- A cli tool (OpenSource)
 - A SaaS Offering (by dbt Labs, Series D with 4,2 Billion validation)
 - Usual “Enterprise” Features
 - Web IDE (also low/no-code)
 - Semantic/Metric Layer
 - dbt Fusion: same API, different engine

Let's get started

Repository & Slides



codecentric/dbt-workshop

Branch: exercises - if you want to get your hands dirty and write code

Branch: main - if you just want to follow along and run some commands

Database Connection



dbt_project.yml

```
1 name: 'dbt_demo'  
2 version: '1.0.0'  
3 config-version: 2  
4  
5 profile: 'dbt_demo'
```



profiles.yml

```
1 dbt_demo: # profile name  
2   target: postgres_dwh # this is the default target  
3   outputs:  
4     postgres_dwh:  
5       type: postgres  
6       host: localhost  
7       user: develop  
8       password: develop  
9       port: 5432  
10      dbname: postgres  
11      schema: analytics
```



exact configuration depends on target system

profiles.yml

- in current working directory
- default to the `~/.dbt/`
- example in Repo-Root: `profiles_template.yml`

Exercise: Connect to duckdb

- Create a profiles.yml in your repository root (there is a template as well)
- Check the config for duckdb
- Check with `dbt debug`
- Install dependencies with `dbt deps`

dbt_project.yml

- the main config for the project
- defines the profile
- general model, seed, snapshot config
- override default directory structure



dbt_project.yml

```
1 name: 'dbt_workshop'~  
2 version: '1.0.0'~  
3 ~  
4 profile: 'dbt_workshop'~  
5 ~  
6 models:~  
7 ...dbt_workshop:~  
8 ....+materialized: table~  
9 ....staging:~  
10 .....schema: staging~  
11 ~  
12 seeds:~  
13 ...+schema: seeds
```

Sources

- the data loaded into the warehouse, without any further transformation
- references *physical assets* (e.g. tables)
- allows to do
 - test the data
 - test the freshness of the data
- can be generated with dbt-codegen package

```
dbt run-operation generate_source
```



The screenshot shows a terminal window with a dark theme. At the top, there are three colored dots (red, yellow, green) followed by the file name "sources.yml". The file content is displayed below:

```
1 sources:-  
2   ...- name:: raw-  
3     ...- tables:-  
4       ...- name:: raw_customers-  
5         ...- columns:-  
6           ...- name:: id-  
7           ...- name:: first_name-  
8           ...- name:: last_name-  
9  
10      ...- name:: raw_orders-  
11        ...- columns:-  
12          ...- name:: id-  
13          ...- name:: user_id-  
14          ...- name:: order_data-  
15          ...- name:: status-  
16          ...-  
17  ...
```

Exercise: Generate Source yml

- Use this documentation: <https://github.com/dbt-labs/dbt-codegen#usage>
- `dbt run-operation generate_source --args '{"schema_name": "raw","generate_columns": true}'`
- Generate sources for `raw_payments` and add it to `models/sources/sources.yml`

Snapshots

Build a history of regularly provided data representing the current state

```
orders_snapshot.sql

1 {%- snapshot.orders_snapshot %}-
2 -
3 {{-
4   config(
5     target_schema='snapshots',
6     unique_key='id',
7     strategy='check',
8     check_cols=["status"]
9   )-
10 }}-
11 -
12 select * from {{ source('raw', 'raw_orders') }}-
13 -
14 {%- endsnapshot %}
```

Exercise: Execute Snapshot

- Review the snapshot for the orders in `snapshots/orders_snapshot.sql`
- Create the snapshot with `dbt snapshot`
- Watch the generated snapshot table in the `snapshots` schema
- Change some state in the `raw_order` table
- Run `dbt snapshot` again
- Watch the changes in the generated snapshot table

Seeds

- (Nearly) static data, e.g country codes
- Seeds are versioned controlled
- Not a mechanism to regularly import data!



country_codes.csv

```
1 country_code, country_name
2 US, United States
3 CA, Canada
4 GB, United Kingdom
5 DE, Germany
6 AT, Austria
7 CH, Switzerland
```

Exercise: Explore & add seeds

- Create the seed with `dbt seed`
- Watch the generated seed table in the `seeds` schema

Models

- defined as sql file
 - with sql code
 - with optional yml config
- ⇒ same for seeds/models/snapshots

```
stg_customers.sql
1 with source as (
2     select * from {{ source('raw', 'raw_customers') }} as
3     ),
4     renamed as (
5         select
6             id as customer_id,
7             first_name,
8             last_name
9         from source
10    )
11   )
12   )
13 select * from renamed
```

```
schema.yml
```

```
version: 2
models:
  - name: stg_customers
    description: The raw customer data with some column renaming.
    columns:
      - name: customer_id
      - tests:
          - unique
          - not_null
```

References

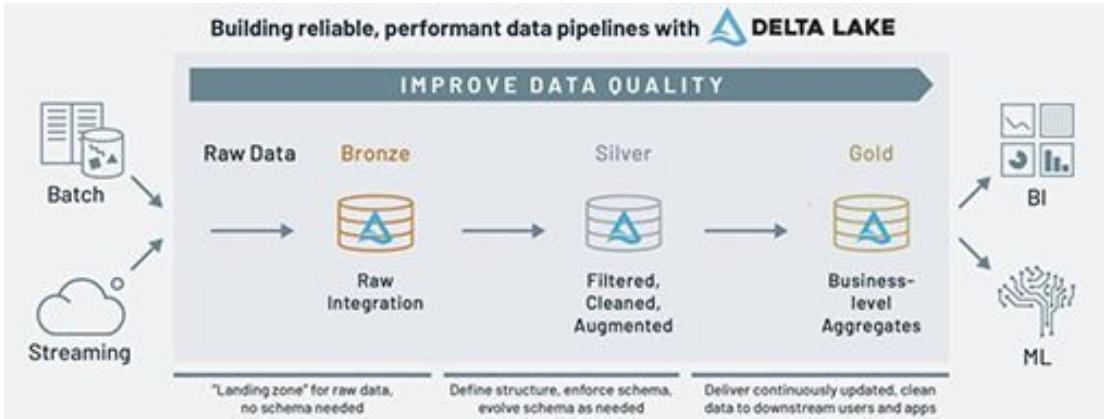
Reference **models**, **snapshots** and **seeds**



schema.yml

```
1 with orders as ( -  
2   -  
3     ....select * from {{ ref('stg_orders') }} -  
4   -  
5 ), -  
6   -  
7 payments as ( -  
8   -  
9     ....select * from {{ ref('stg_payments') }} -  
10  -  
11 ), -  
12  -  
13 ...
```

Stages/Zones in the DWH



Exercise: Create Models

- Create the staging model for `payments`
 - Use Common Table expression
 - rename the id column to `payment_id`
-
- Finish the `customers` analysis model
 - add the payment information
 - use the data in the `payment` staging table
 - join the data with the `order` information to get the `customer_id`
 - the `customer_payments` should contain the total payment amount per customer
 - add this value to the final output as `customer_lifetime_value`

Do not forget to define the yml!

Use `dbt run --select staging.stg_payments / dbt run --select +analysis.customers`

Exposure

- Usages of dbt models in external applications, processes..

Available properties

Required:

- **name**: a unique exposure name written in [snake case](#)
- **type**: one of `dashboard`, `notebook`, `analysis`, `ml`, `application` (used to organize in docs site)
- **owner**: `name` or `email` required; additional properties allowed

Expected:

- **depends_on**: list of refable nodes, including `ref`, `source`, and `metric` (While possible, it is highly unlikely you will ever need an `exposure` to depend on a `source` directly)

Optional:

- **label**: may contain spaces, capital letters, or special characters.
- **url**: enables the link to [View this exposure](#) in the upper right corner of the generated documentation site
- **maturity**: one of `high`, `medium`, `low`

General properties (optional)

```
exposures.yml
1 version: 2
2
3 exposures:
4
5   - name: weekly_jaffle_metrics
6     label: Jaffles by the Week
7     type: dashboard
8     maturity: high
9     url: https://bi.tool/dashboards/1
10    description: >
11      Did someone say "exponential growth"?
12
13 depends_on:
14   - ref('fct_orders')
15   - ref('dim_customers')
16   - source('gsheets', 'goals')
17   - metric('count_orders')
18
19 owner:
20   name: Callum McData
21   email: data@jaffleshop.com
```

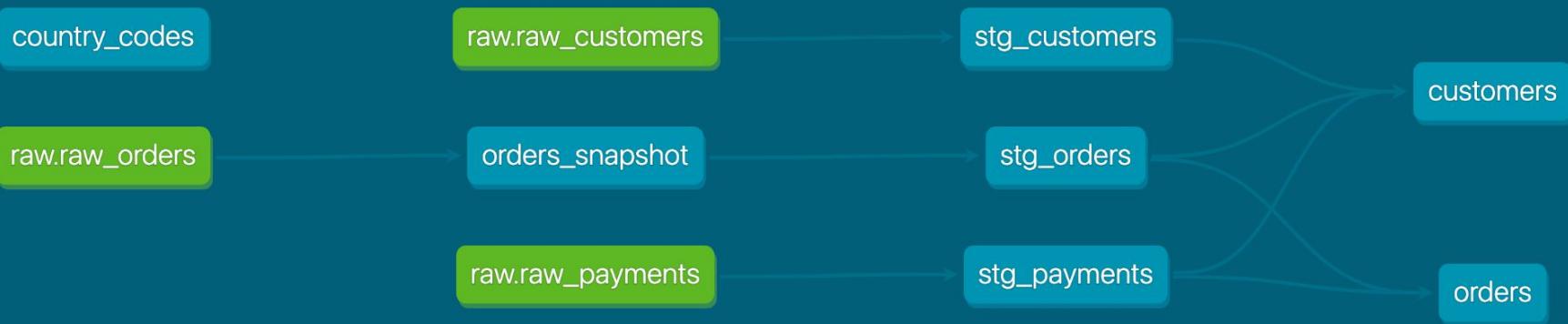
Exercise: Create an Exposure

- “Business” is using a dashboard on the customers table, to see the valuable customers. The upper management relies heavily on this for decision making.
- The “Data Science Team” uses all the staging data for training a demand forecast model. The project is still experimental.

Add the missing information to the exposures.yml

Lineage

Built when running dbt run, based on the DAG of references



Documentation

Build from all the information available

orders table

[Details](#) [Description](#) [Columns](#) [Referenced By](#) [Depends On](#) [Code](#)

Details

TAGS	OWNER	TYPE	PACKAGE	LANGUAGE	RELATION	ACCESS	VERSION
untagged	develop	table	dbt_demo	sql	postgres.analytics.orders	protected	

Description

This table has basic information about orders, as well as some derived facts based on payments

Columns

COLUMN	TYPE	DESCRIPTION	TESTS	MORE?
order_id	integer	This is a unique identifier for an order	U N	>
customer_id	integer	Foreign key to the customers table	N F	>
order_date	text	Date (UTC) that the order was placed		>

Documentation

Can also be provided as separate markdown files

status	text	^
Details		
Description		
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 been shipped to the customer and is currently in transit	
completed	The order has been received by the customer	
return_pending	The customer has indicated that they would like to return the order, but it has not yet been received at the warehouse	
returned	The order has been returned by the customer and received at the warehouse	
Generic Tests		
Accepted Values: placed, shipped, completed, return_pending, returned		

Exercise: Generate & Visit Documentation

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

Data Tests

Generic

```
staging.yaml

1 models:-
2   ... name::stg_customers
3   ... description: The raw customer data with some column renaming.
4   ... columns:-
5     ... - name::customer_id
6     ... - tests:-
7       ... - unique
8       ... - not_null
```

Single

```
tests/assert_customers_id_not_null.sql

1 select *-
2 from {{ ref("stg_customers") }}-
3 where customer_id is null-
4
```

Test-Driven model development (Test first, then yml + sql)

Develop against data contracts, verify them

Exercise: Add dbt provided Generic Tests

- Add generic tests to Source `payments`
 - `id`: unique, not null
 - `order_id`: reference to source `raw_orders`, field `id`
- Staging `stg_payments`
 - `id`: unique, not null
 - `payment_method`: erwartet `credit_card`, `coupon`, `bank_transfer` oder `gift_card`
- Analysis `customers`
 - `id`: unique, not null

Relevante Dokumentation:

<https://docs.getdbt.com/docs/build/tests#generic-tests>

Exercise: Run tests & view failures

- Run tests with `dbt test`
- Remove some of the accepted payment methods
- Run tests with `dbt test --store-failures`
- Have a look at the failures

Unit Tests

- When your SQL contains complex logic:
 - Regex
 - Date math
 - Window functions
 - case when statements when there are many whens
- When you're writing custom logic to process input data, similar to creating a function
- Logic for which you had bugs reported before.
- Edge cases not yet seen in your actual data that you want to handle.
- Prior to refactoring the transformation logic (especially if the refactor is significant).

Run during development & in CI/CD -

no need to run in production (`--exclude-resource-type unittest`)

Unit Tests



customers.sql

```
1 with customers as (
2     select * from {{ ref('stg_customers') }}
3 ),
4
5 accepted_email_domains as (
6     select * from {{ ref('top_level_email_domains') }}
7 ),
8
9 check_valid_emails as (
10    select
11        [ ... ]
12        customers.email,
13        coalesce (regexp_like(
14            customers.email, '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.\.[A-Za-z]{2,}$'
15        )
16        = true
17        and accepted_email_domains.tld is not null,
18        false) as is_valid_email_address
19    from customers
20        left join accepted_email_domains
21        on customers.email_top_level_domain = lower(accepted_email_domains.tld)
22
23 )
24 select * from check_valid_emails
```

Unit Tests

```
customers.yml

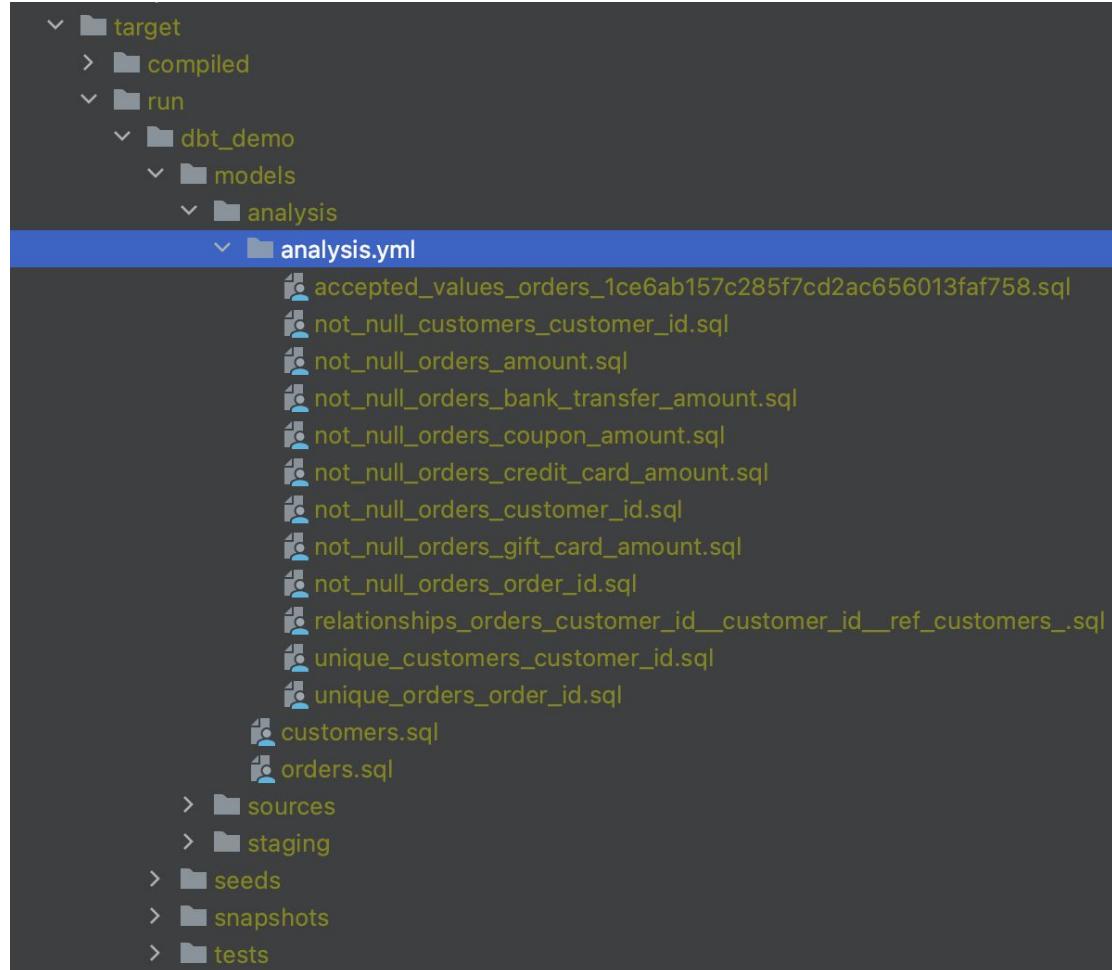
1 unit_tests:
2   - name: test_is_valid_email_address
3     description: "Check my is_valid_email_address logic captures all known edge cases"
4     model: customers
5     given:
6       - input: ref('stg_customers')
7         rows:
8           - {email: cool@example.com, email_top_level_domain: example.com}
9           - {email: cool@unknown.com, email_top_level_domain: unknown.com}
10          - {email: badgmail.com, email_top_level_domain: gmail.com}
11          - {email: missingdot@gmailcom, email_top_level_domain: gmail.com}
12        - input: ref('top_level_email_domains')
13          rows:
14            - {tld: example.com}
15            - {tld: gmail.com}
16        expect:
17          rows:
18            - {email: cool@example.com, is_valid_email_address: true}
19            - {email: cool@unknown.com, is_valid_email_address: false}
20            - {email: badgmail.com, is_valid_email_address: false}
21            - {email: missingdot@gmailcom, is_valid_email_address: false}
```

Exercise: dbt-unit-testing

- Locate the unit test for the customer model
- Remove the comments
- Run it
- Fix the expected data

The target folder

- contains the compiled and ran sql files
- the compiled graph
- the run results
- the docs (the catalog)



Packages

- dbt-expectations, dbt-date, dbt-privacy, dbt-constraints, elementary, common data

Table shape

- expect_column_to_exist
- expect_row_values_to_have_recent_data
- expect_grouped_row_values_to_have_recent_data
- expect_table_aggregation_to_equal_other_table
- expect_table_column_count_to_be_between
- expect_table_column_count_to_equal_other_table
- expect_table_column_count_to_equal
- expect_table_columns_to_not_contain_set
- expect_table_columns_to_contain_set
- expect_table_columns_to_match_ordered_list
- expect_table_columns_to_match_set
- expect_table_row_count_to_be_between
- expect_table_row_count_to_equal_other_table
- expect_table_row_count_to_equal_other_table_times_factor
- expect_table_row_count_to_equal

Distributional functions

- expect_column_values_to_be_within_n_moving_stdevs
- expect_column_values_to_be_within_n_stdevs
- expect_row_values_to_have_data_for_every_n_datepart



packages.yml

```
1 packages:-  
2   ... package: calogica/dbt_expectations  
3   ... version: ["≥0.8.0", "<0.9.0"]
```

Install with `dbt deps`

Exercise: Install Packages

- Review the `packages.yml`
- Add `metaplane/dbt_expectations` in version 0.8.0 oder einer Bugfix version (0.10.1)
- Run `dbt deps`

Exercise: Install Packages (elementary only with postgres!)

- Review the `packages.yml`
- Add `elementary-data/elementary` in version 0.10.0 oder einer Bugfix version (0.10.1)
- Add `metaplane/dbt_expectations` in version 0.8.0 oder einer Bugfix version (0.10.1)
- Run `dbt deps`
- Configure elementary models in `dbt_project.yml`

```
models:
```

```
    elementary:
```

```
        +schema: elementary
```

Exercise: Write dbt-expectations tests

- Add one or two dbt-expactation tests
- Ideas:
 - Shape (number of rows, columns)
 - Aggregations (Expect Max/Min Values)
- Available tests: <https://github.com/calogica/dbt-expectations#available-tests>

Exercise: Use Elementary for test-visualization (only with postgres)

- make sure to have the `elementary-data` Python Package installed
- run models and tests (`dbt run + dbt test` or `dbt build`)
- create the report with `edr report`
- Bonus:
 - Explore the anomaly tests of elementary data:
<https://docs.elementary-data.com/guides/add-elementary-tests>
 - Add an anomaly detection test, e.g. a volume test on the staging order model
 - Run the test and visualize it again

Jinja & Macros

```
orders.sql
```

```
1 select ~
2   order_id, ~
3 ~
4   {% for payment_method in payment_methods -%} ~
5     sum(case when payment_method = '{{ payment_method }}' then amount else 0 end) as {{ payment_method }}_amount, ~
6   {% endfor -%} ~
7 ~
8   sum(amount) as total_amount ~
9 ~
10 from payments ~
11 ~
12 group by order_id
```



```
cents_to_dollars.sql
```

```
1 {% macro cents_to_dollars(column_name, precision=2) %} ~
2   ({{ column_name }} / 100)::numeric(16, {{ precision }})) ~
3 {% endmacro %}
```

having different behaviors depending on the target

Exercise: Write a macro

- Write and use a Macro:
 - in `stg_payments` the `amount` is stored in cents
 - convert it do dollar
 - write a macro `cents_to_dollars` so it can be reused

Exercise: Jinja2 + Model

Finish the `orders` analysis model

- add the payment information
 - use the data in the `payment` staging table
 - for each order in the payment table
 - have a column for each payment method and the amount paid with the method
 - and a `total_amount` column
- The final table should contain the separated payment information and the order information

At the top of the `orders.sql` the model is deactivated. Active it when you work on this!

Exercise: Incremental models

Incremental models are very useful.

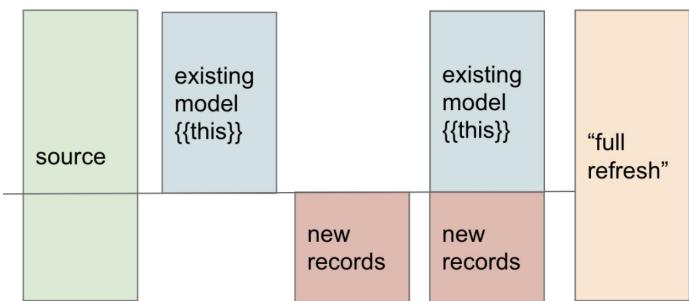
Using an incremental model limits the amount of data that needs to be transformed, vastly reducing the runtime of your transformations. This improves warehouse performance and reduces compute costs.

This is a great source for explanation

<https://docs.getdbt.com/guides/best-practices/materializations/4-incremental-models>

Task:

- enable the models `stg_orders_incr.sql`, `orders_per_year`
- do `dbt run --select stg_orders_incr.sql`
- do `dbt run --select orders_per_year`
- in `raw_orders_copy`, set the config `append_data=true`
- do `dbt run --select orders_per_year` and see how it increments



Model Contracts

- A defined API for public models
- Includes
 - Column Names
 - Column Types
 - Constraint (depending on platform)
- Just the shape, not the content of the model!
- Checks ***at build-time*** that query returns expected columns
- Will include defined columns in the DDL statement
- Will detect breaking changes in the schema



model.yml

```
1 models:
2   - name: dim_customers
3     config:
4       contract:
5         enforced: true
6     columns:
7       - name: customer_id
8         data_type: int
9         constraints:
10          - type: not_null
11        - name: customer_name
12          data_type: string
```

Model Versions

- Same as for an API - schemas and (model contracts) change
- New version for breaking changes - not only in schema
- Version modeled can be referenced explicitly, or just with latest

v	version	ref syntax	File name	Database relation
3	"prerelease"	<code>ref('dim_customers', v=3)</code>	<code>dim_customers_v3.sql</code>	<code>analytics.dim_customers_v3</code>
2	"latest"	<code>ref('dim_customers', v=2)</code> and <code>ref('dim_customers')</code>	<code>dim_customers_v2.sql</code> or <code>dim_customers.sql</code>	<code>analytics.dim_customers_v2</code> and <code>analytics.dim_customers</code> (recommended)
1	"old"	<code>ref('dim_customers', v=1)</code>	<code>dim_customers_v1.sql</code>	<code>analytics.dim_customers_v1</code>

Model Versions

```
model.yml

1 models:
2   - name: dim_customers
3     latest_version: 1
4     config:
5       materialized: table
6       contract: {enforced: true}
7     columns:
8       - name: customer_id
9         description: This is the primary key
10        data_type: int
11       - name: country_name
12         description: Where this customer lives
13         data_type: varchar
14
15     # Declare the versions, and highlight the diffs
16     versions:
17       - v: 1
18         # Matches what's above -- nothing more needed
19       - v: 2
20         # Removed a column -- this is the breaking change!
21         columns:
22           - include: all
23             exclude: [country_name]
```

Plus a sql file for each model version

Dev/Prod Separation



profiles.yml

```
1 dbt_demo: # profile name
2   target: postgres_dwh_ci
3   outputs:
4     postgres_dwh_ci:
5       type: postgres
6       host: localhost
7       user: develop
8       password: develop
9       port: 5432
10      dbname: postgres
11      schema: ci
```



profiles.yml

```
1 dbt_demo: # profile name
2   target: postgres_dev_mniehoff
3   outputs:
4     postgres_dev_mniehoff:
5       type: postgres
6       host: localhost
7       user: develop
8       password: develop
9       port: 5432
10      dbname: postgres
11      schema: dev_mniehoff
```

beware: snapshots are not namespaced

dbt + Formatting: sqlfluff

```
sqlfluff
1 [sqlfluff]
2 templater = dbt
3 dialect = postgres
4 max_line_length = 120
5
6 [sqlfluff:templater:jinja]
7 apply_dbt_builtins = true
8
```

sqlfluff

```
.pre-commit-config.yaml
1 repos:
2   - repo: https://github.com/pre-commit/pre-commit-hooks
3     rev: v4.4.0
4     hooks:
5       - id: check-yaml
6       - id: end-of-file-fixer
7       - id: trailing-whitespace
8       - id: mixed-line-ending
9       - id: check-merge-conflict
10      - id: check-added-large-files
11        args: ["--maxkb=200"]
12        exclude: ^seeds/
13      - repo: https://github.com/psf/black
14        rev: 23.3.0
15        hooks:
16          - id: black
17      - repo: https://github.com/charliermarsh/ruff-pre-commit
18        rev: "v0.0.262"
19        hooks:
20          - id: ruff
21            args: [--fix, --exit-non-zero-on-fix]
```

pre-commit hooks

Exercise: Do a commit & use pre-commit

- Make sure the `pre-commit`, `sqlfluff` and `sqlfluff-templater-dbt` Python Packages are installed
- Install hooks with `pre-commit install`
- Do a commit and watch the checks, adjust the code until the commit succeeds
- Checks can also be executed by `pre-commit run --all-files`

I could go on ..

- materialization
- ad-hoc analysis
- hooks
- authorization

```
exposures.yml

1 exposures:
2
3   - name: weekly_jaffle_report
4     type: dashboard
5     maturity: high
6     url: https://bi.tool/dashboards/1
7     description: >
8       Did someone say "exponential growth"?
9
10    depends_on:
11      - ref('orders')
12      - ref('customers')
13
14    owner:
15      name: Callum McData
16      email: data@jaffleshop.com
```

Ecosystem + Integration

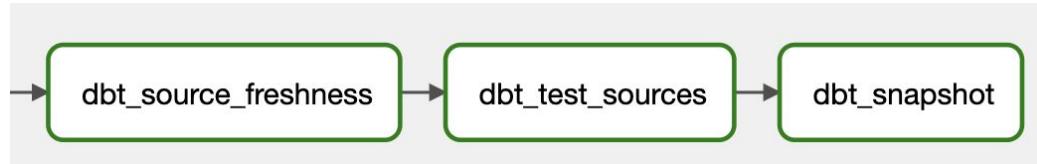


<https://github.com/Hiflylabs/awesome-dbt>

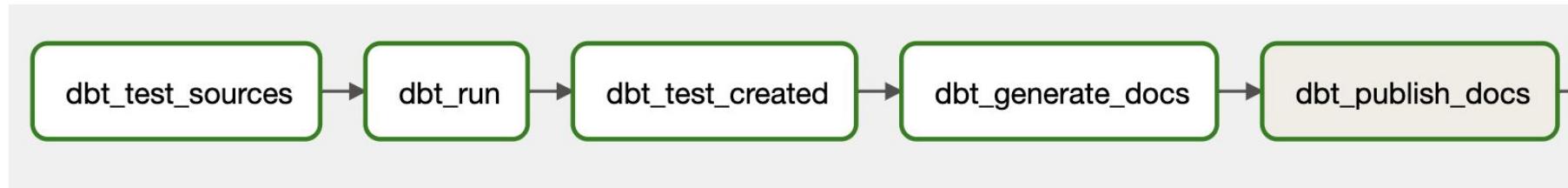
Orchestration

- Airflow, prefect, dagster, ...
- Best experiences:
 - Create a docker container based on the project
 - Use this container to run commands

data import DAG:

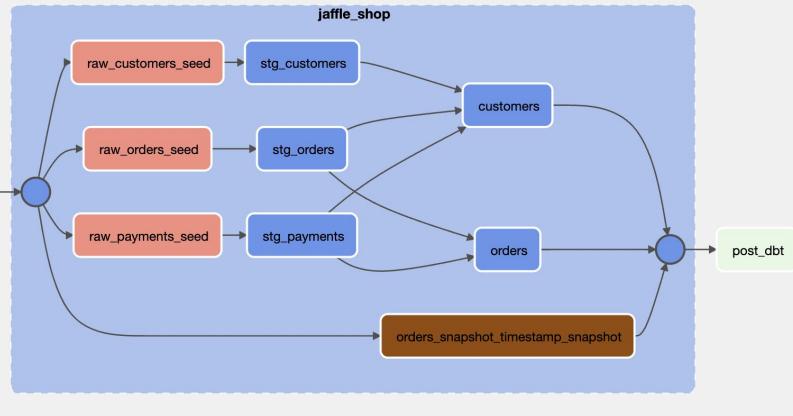


dbt DAG:



Orchestration

- At least dagster & airflow support “included dbt” - dbt models & tests als nodes in a job/DAG
 - Airflow: <https://astronomer.github.io/astronomer-cosmos/index.html>
 - dagster: nativ



Vizualization + Query

- Everything that can connect to the storage
- Bonus: integration with dbt, like Lightdash

```
version: 2

models:
  - name: projects
    columns:
      - name: dashboards_created_num_total
      - name: days_since_project_created
```

The screenshot shows the Lightdash interface. On the left, there's a dark sidebar with the text "Projects". To its right is a search bar with the placeholder "Search metrics + dimensions". Below the search bar, under the heading "Dimensions", are two items: "123 Dashboards created num total" and "123 Days since project created". A purple arrow points from the "dashboards_created_num_total" dimension in the Lightdash interface to the "dashboards_created_num_total" column in the dbt configuration file.

dbt connection ?

Your dbt project must be compatible with [dbt version 1.4.1](#)

Type *
Github

Personal access token

Repository *
mniehoff/dbt-demo

Branch *
main

Project directory path *
/

Host domain (for Github Enterprise)
github.com

Target name
prod

Schema *
analytics

miro

duckDB + dbt

- Community Plugin
- DuckDB: 
- sqlite but for OLAP
- fast SQL layer for
Files/Blobs/...
- ⇒ dbt on Blob Storage/Files



```
● ● ● profiles.yml

1 your_profile_name:
2   target: dev
3   outputs:
4     dev:
5       type: duckdb
6       path: 'file_path/database_name.duckdb'
7       extensions:
8         - httpfs
9         - parquet
10      settings:
11        s3_region: my-aws-region
12        s3_access_key_id: "{{ env_var('S3_ACCESS_KEY_ID') }}"
13        s3_secret_access_key: "{{ env_var('S3_SECRET_ACCESS_KEY') }}"
```

EL_T



Why is it good?

Simplicity

- good starting point for documentation
- data lineage within the DWH can be easily understood and visualized
- supports dataset-centric visualization (opposed to query- or semantic-driven)

Engineering

- logic and config treated as code and stored in source control
- repeatable, automated test, even test driven would be possible
- easy automation and CI/CD

Ecosystem

- huge ecosystem with great integrations
- works with every major storage
- ready to adapt with new trends and technologies

What is it not?



Import



Data Catalog



Visualization



Low-/No-Code

Alternatives

- sqllmesh
 - compatibility with dbt
 - improves some shortcomings of dbt (virtual data environments)
 - same owner (fivetran) by now
- bruin (<https://github.com/bruin-data/bruin>)
 - fairly new
 - combines EL + T

T₁ H₄ A₁ N₁ K₅
Y₄ O₁ U₁

Innovative - Trustful - Competent - Pragmatic

We support you with
your data projects

Feel free to contact us!

📍 codecentric AG
Hochstraße 11
42697 Solingen

👤 Matthias Niehoff
Head of Data
matthias.niehoff@codecentric.de
www.codecentric.de

@codecentric

