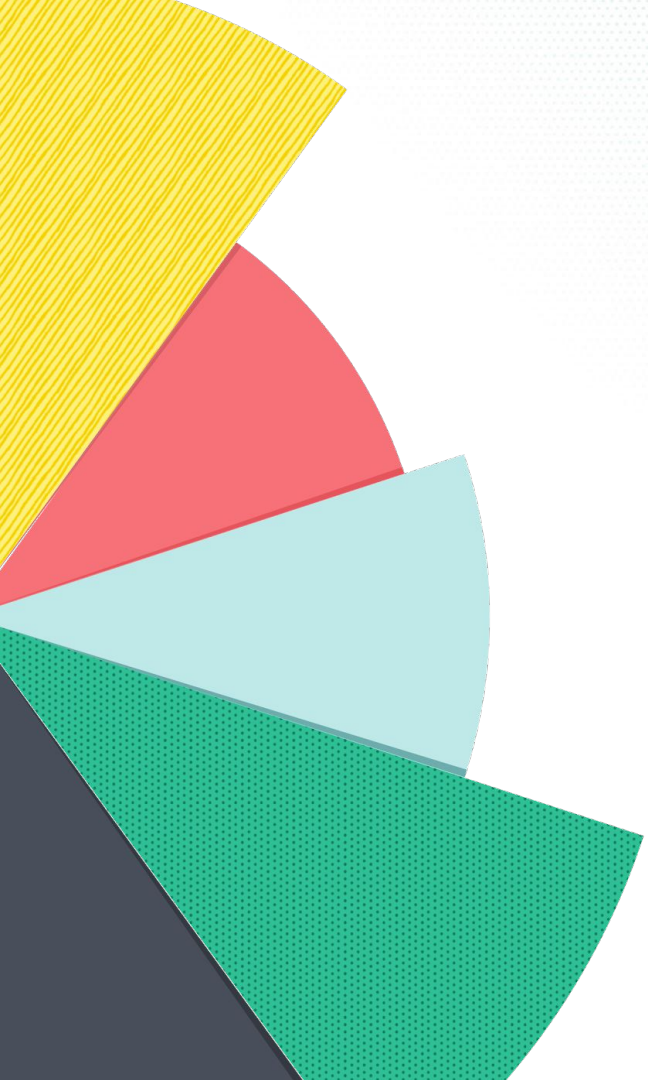
preset

Airflow Summit

Operating Contexts

Patterns around defining how a DAG should behave in dev, staging, prod & beyond



Agenda

- Operating Contexts? - WTF
- Implementing OCs
- Typical Patterns
- Creative Patterns
- The case for OC semantics in Airflow's IA

Maxime Beauchemin
mistercrunch

creator of Apache Airflow and Apache Superset - founder at Preset

Edit profile

1k followers · **11** following · ☆ **139**

preset-io
San Mateo, CA
maximebeauchemin@gmail.com
mistercrunch.blogspot.com

Organizations

- Passionate about building data tools!
- Started Apache **Airflow** at Airbnb in 2014
- Started Apache **Superset** at Airbnb in 2015
- Started **Preset** - The Apache Superset company in 2019

Apache Superset™

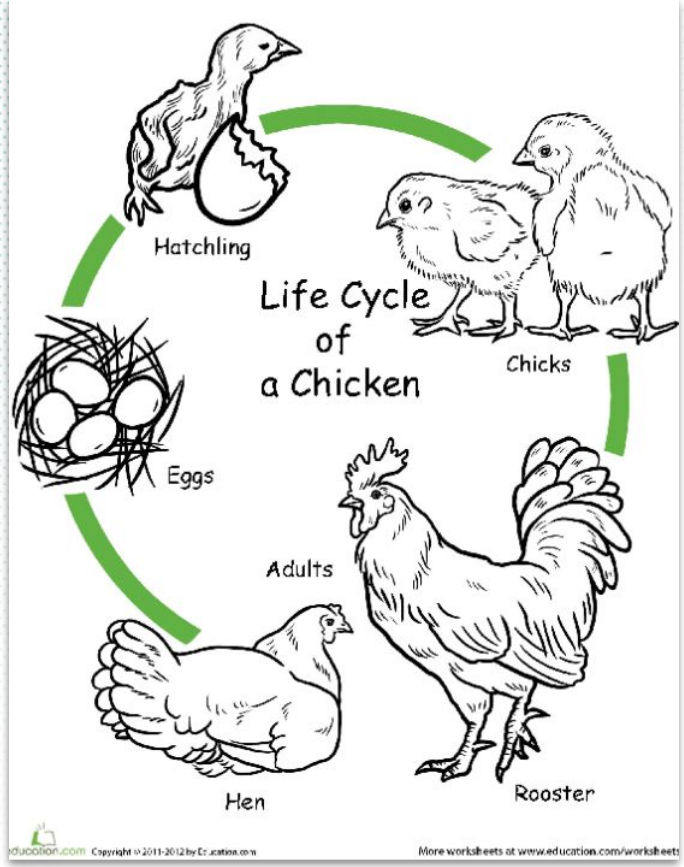
Apache Airflow

preset™

Operating Contexts? - WTF

- **[DISCLAIMER]** THIS IS NOT AN AIRFLOW CONCEPT!
- Synonyms/related ideas: mode, DAG modifiers, parameterizable DAG, target
- A common pattern emerging from the flexibility of the DSL
- **Definition:** *a **declared** **mode of operation** for a DAG that **alters its shape or behavior** in a **deterministic** way*
- Example OCs:
  - development, staging & production
  - Verbose / high test
  - Backfill
  - Fast/cheap mode
  - ...

DAG Life Cycle



High level mechanics

How?

- Through environment variables!
- A function that returns a DAG object - can expose a clear function signature

What?

- Alter `DAG.default_args`
- Alter `DAG.params`
- Alter `conn_id`s
- Alter datasets names / pointers

Simplest Example

```
import os
from datetime import datetime

from airflow import DAG


OPERATING_CONTEXT = os.environ.get('AIRFLOW__OPERATING_CONTEXT', 'dev')

default_args = {
    'owner': 'airflow',
    'retries': 1,
}

schema_suffix = '_' + OPERATING_CONTEXT if OPERATING_CONTEXT else ''
if OPERATING_CONTEXT == 'dev':
    default_args.update({
        'retries': 0,
    })
```

```
$ AIRFLOW__OPERATING_CONTEXT=prod airflow {subcommand} {...}
```

DAG function signature

```
def get_dag(
        operating_context: Literal["prod", "dev", 'stg'] = "prod",
        fast_mode: bool = False,
) -> DAG:
```

Pattern #1 - Connections / Env mapping

- Create different connections for different env (ie: “snowflake”, “snowflake_staging”, “snowflake_dev”)
- Alter your default_args base on an envvar

```
OPERATING_CONTEXT = os.environ.get('AIRFLOW__OPERATING_CONTEXT')

default_args = {
    'snowflake_conn_id': 'snowflake_prod',
}

if OPERATING_CONTEXT == 'staging':
    default_args.update({
        'snowflake_conn_id': 'snowflake_staging',
    })
```

preset

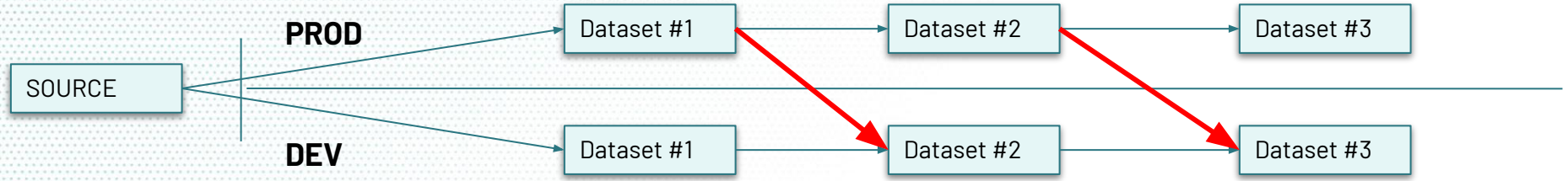
Pattern #2 - Schema / Env mapping

- Assuming clear mapping of environment to database schema mapping
- Apply suffix / naming scheme
- It's common to source from a production schema upstream

```
schema_suffix = '_' + OPERATING_CONTEXT if OPERATING_CONTEXT else ''
params = {
    'core_data_schema': 'core_data' + schema_suffix
    'core_cx_schema': 'core_cx' + schema_suffix
```

Pattern #3 - Crossover!

Source from prod [trusted, up to date] and load into a another env



Pattern #4 - Limit dataset(s)

- Apply one or many “LIMIT” clauses upstream
- Non deterministic, useful as a not-so-”dry run”
- Some specific assertions can be done with CheckOperators

```
FROM {{ source_schema }}.super_large_table
{% if params.fast_mode %}
LIMIT {{ params.fast_mode_row_limit }}
{% endif %}
```

Pattern #4 - Fixtures as source dataset

- Use a “fixture” as sources (static dataset)
- Perform specific checks & assertions

The Case for incorporating this in Airflow

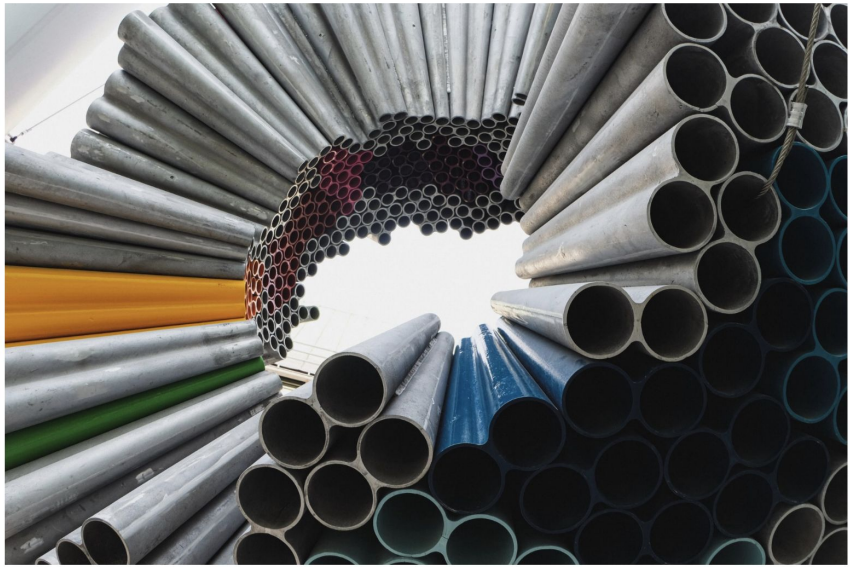
- Clear semantics (env var name, global variables)
- “Grepable”!
- An anchor point for best practices to be defined
- Coupling defaults around things like logging levels, ...

Conclusion

- No one can afford a full-scale staging DW
- Find the right patterns that are right for your DAG / team / org
- Get creative!
- Careful with the DAG-factory patterns
- No rules! If there were, you should break them

Upcoming on Medium

Defining “Operating Contexts” in Airflow DAGs



This post defines more formal semantics around the idea of defining “operating contexts” while authoring Airflow DAGs. To be clear an operating

OpenLineage

Data Lineage with Apache Airflow using OpenLineage

Julien Le Dem and Willy Lulciuc, Datakin | July 2021


That's all Folks!