YETL

Yet Another ETL Framework

Yet (another) ETL Framework

Expressive, agile, fun for Data Engineers using Python!

Yetl is a configuration API for Databricks datalake house pipelines. It allows you to easily define the configuration and metadata of pipelines that can be accessed using a python to build modular pyspark pipelines with good software engineering practices.

```
pip install yetl-framework
```

What does it do?

Using yetl a data engineer can define configuration to take data from a source dataset to a destination dataset and just code the transform in between. It takes care of the mundane allowing the engineer to focus only on the value end of data flow in a fun and expressive way.

Feaures:

- Define table metadata, properties and dependencies easily that can be used programmaticaly with pipelines
- Define table metadata in Excel and covert to a configuration yaml file using the CLI
- Creates Delta Lake databases and tables during the pipeline:
  - Delclare SQL files with create table statements in the SQL dir
  - Dynamically infer the table SQL from the config and the schema of a data frame
  - Initialise a Delta Table with no schema and the load it with merge schema true.
- Create spark schema for schema of read
- Once you have a schema or data frame sometime it's handy to have the DDL e.g. for schema hints, the API will work this out for you
- API provides a table collection index and table mappings API with property acceessors for each table to use as you wish when building the pipeline
- Supports jinja variables for expressive configuration idioms
- Provides a timeslice object for parameterising pipelines with wildcard paths
- Provides timeslice transform to parse the datetime from a path or filename into the dataset
- Can be used to create checkpoints in a consistent ways for your project for complex streaming patterns

Once you have re-usable modular pipeline code and configuration... you can get really creative:

- Parameterise whether you want to run your pipelines as batch or streaming
- Bulk load a migration and then switch to batch streaming autoloader for incrementals
- Generate databricks worflows
- Generate databricks DLT pipelines
- Parameterise bulk reload or incremental pipelines
- Test driven development

- Integrate with data expectations framework

What is it really?

The best way to see what it is, is to look at a simple example.

Define your tables:

```yaml
version: 3.0.0

audit_control:
  delta_lake:
    raw_dbx_patterns_control:
      catalog: hub
      header_footer:
        sql: ../sql/{{database}}/{{table}}.sql
        depends_on:
          - raw.raw_dbx_patterns.*
      raw_audit:
        # if declaring your tables using SQL you can declare that SQL in a linked file.
        # Jinja templating is supported for catalog, database and table
        sql: ../sql/{{database}}/{{table}}.sql
        depends_on:
          # when depending on all the tables in a db you can use the wild card
          # note these layers supported audit_control, landing, raw, base, curated
          # layer.db.*
          - raw.raw_dbx_patterns.*
          # or to declare specifically
          # layer.db.*
          - audit_control.raw_dbx_patterns_control.header_footer

# landing is just a reference to source files
# there are no properties of interest but the database
# name and tables themselves
landing:
  # read declares these as tables that read
  # using spark.read api and are not deltalake table
  # these are typically source files of a specific format
  # landed in blob storage.
  read:
    landing_dbx_patterns:
      catalog: hub
      customer_details_1: null
      customer_details_2: null

# raw is the 1st ingest layer of delta lake tables
# raw is optional you may not need it if you data feed provided is very clean.
raw:
  delta_lake:
    raw_dbx_patterns:
      catalog: hub
      customers:
        id: id
        depends_on:
          - landing.landing_dbx_patterns.customer_details_1
          - landing.landing_dbx_patterns.customer_details_2
        # we can define tables level expected thresholds for
        # schema on read errors from landing.
        # There are warning and exception thresholds
        # how what behaviour they drive in the pipeline
        # is left to data engineer.
        warning_thresholds:
          invalid_ratio: 0.1
          invalid_rows: 0
          max_rows: 100
          min_rows: 5
        exception_thresholds:
          invalid_ratio: 0.2
          invalid_rows: 2
          max_rows: 1000
          min_rows: 0
```

```
        # you can define any custome properties that you want        # properties provided can be used a filter when looking up        # tables in the API collection        custom_properties:          process_group: 1        # there are may other properties supported for delta lake tables        # e.g. zorder, partition by, cluster by,         # delta table properties, vacuum, auto increment id column        # see reference docsß for this yaml spec        z_order_by:          - _load_date_1          - _load_date_2        vacuum: 30base:  delta_lake:    # delta table properties can be set at stage level or table level    delta_properties:      delta.appendOnly: true      delta.autoOptimize.autoCompact: true          delta.autoOptimize.optimizeWrite: true      delta.enableChangeDataFeed: false    base_dbx_patterns:      catalog: hub      customer_details_1:        id: id        depends_on:          - raw.raw_dbx_patterns.customers        # delta table properties can be set at stage level or table level        # table level properties will overwride stage level properties        delta_properties:          delta.enableChangeDataFeed: true      customer_details_2:        id: id        depends_on:          - raw.raw_dbx_patterns.customers
```

Define you load configuration:

```yaml
version: 3.0.0
tables: ./tables.yaml

audit_control:
  delta_lake:
    # delta table properties can be set at stage level or table level
    delta_properties:
        delta.appendOnly: true
        delta.autoOptimize.autoCompact: true
        delta.autoOptimize.optimizeWrite: true
    managed: false
    container: datalake
    location: /mnt/{{container}}/data/raw
    path: "{{database}}/{{table}}"
    options:
      checkpointLocation: "/mnt/{{container}}/checkpoint/{{project}}/{{checkpoint}}"

landing:
  read:
    trigger: customerdetailscomplete-{{filename_date_format}}*.flg
    trigger_type: file
    container: datalake
    location: "/mnt/{{container}}/data/landing/dbx_patterns/{{table}}/{{path_date_format}}"
    filename: "{{table}}-{{filename_date_format}}*.csv"
    filename_date_format: "%Y%m%d"
    path_date_format: "%Y%m%d"
    # injects the time period column into the dataset
    # using either the path_date_format or the filename_date_format
    # as you specify
    slice_date: filename_date_format
    slice_date_column_name: _slice_date
    format: cloudFiles
    spark_schema: ../schema/{{table.lower()}}.yaml
    options:
      # autoloader
      cloudFiles.format: csv
      cloudFiles.schemaLocation:  /mnt/{{container}}/checkpoint/{{project}}/{{checkpoint}}
      cloudFiles.useIncrementalListing: auto
      # schema
      inferSchema: false
      enforceSchema: true
      columnNameOfCorruptRecord: _corrupt_record
      # csv
      header: false
      mode: PERMISSIVE
      encoding: windows-1252
      delimiter: ","
      escape: '"'
      nullValue: ""
      quote: '"'
      emptyValue: ""

raw:
  delta_lake:
    # delta table properties can be set at stage level or table level
    delta_properties:
      delta.appendOnly: true
      delta.autoOptimize.autoCompact: true
      delta.autoOptimize.optimizeWrite: true
      delta.enableChangeDataFeed: false
    managed: false
    container: datalake
    location: /mnt/{{container}}/data/raw
```

```
    path: "{{database}}/{{table}}"    options:      mergeSchema: true      checkpointLocation: "/mnt/{{container}}/checkpoint/{{project}}/{{checkpoint}}"base:  delta_lake:    container: datalake    location: /mnt/{{container}}/data/base    path: "{{database}}/{{table}}"    options: null
```

Import the config objects into you pipeline:

```python
from yetl import Config, Timeslice, StageType, Read, DeltaLake

pipeline = "autoloader"
config_path = "./test/config"
project = "test_project"
timeslice = Timeslice(day="*", month="*", year="*")
config = Config(
    project=project, pipeline=pipeline, config_path=config_path, timeslice=timeslice
)
table_mapping = config.get_table_mapping(
    stage=StageType.raw, table="customers"
)

source: Read = table_mapping.source["customer_details_1"]
destination: DeltaLake = table_mapping.destination
config.set_checkpoint(source=source, destination=destination)

print(table_mapping)
```

Use even less code and use the decorator pattern:

```python
@yetl_flow(
        project="test_project",
        stage=StageType.raw,
        config_path="./test/config"
)
def autoloader(table_mapping:TableMapping):
    # << ADD YOUR PIPELINE LOGIC HERE - USING TABLE MAPPING CONFIG >>
    return table_mapping # return whatever you want here.

result = autoloader(table="customers")
```

Example project

databricks-patterns

This example projects has 4 projects loading data landed from:

- adventure works
- adventure works lt
- adventure works dw
- header_footer - a small demo of files with semic structured headers and footers that stripped into an audit table on the when loaded.
- header_footer_uc - same as header footer but using databricks Unity Catalog.

Yet Another ETL Framework