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:

```
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:

```
@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.

Project

Yetl has the concept of project. A project houses all the assets to configure your pipelines, the pipelines themselves and deploymemnt assets. They are deployable products that can be a data feed or an individual component data feed product of something larger.

Although it's entirely possible to reconfigure the project structure a canonical structure is recommended since the API just works without any additional consideration for all of these environments:

- Local
- Databricks Workspace
- Databricks Repo

Create a new project

Yetl has a built-in cli for common tasks. One of those common tasks is creating a new project.

Create a python virtual environment and install yetl:

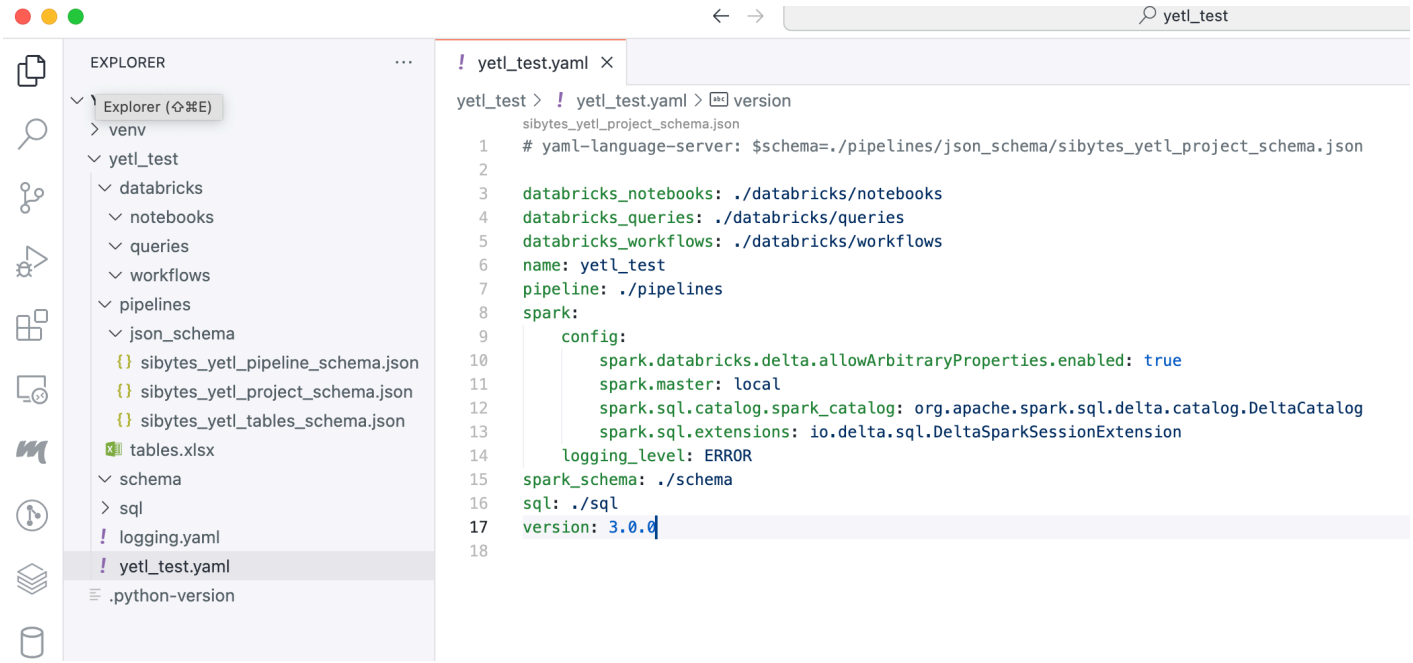
```
mkdir yetl_test
cd yetl_test
python -m venv venv
source venv/bin/activate
pip install --upgrade pip
pip install yetl-framework
```

Create a new yetl project:

```
python -m yetl init test_yetl
```

This will create:

- logging configuration file `logging.yaml`
- yetl project configuration file `yetl_test.yaml` project configuration file.
- Excel template `tables.xlsx` to curate tables names and properties that you want to load, this can be built into a configuration `tables.yaml` with `python -m yetl import-tables`
- `json-schema` contains json schema that you can reference and use redhat's yaml plugin when crafting configuration to get intellisense and validation. There are schemas for:
  - `./pipelines/project.yaml`
  - `./pipelines/tables.yaml`
  - `./pipelines/<my_pipelines>.yaml`
- directory structure to build pipelines



- databricks - will store databricks workflows, notebooks, etc that will be deployed to databricks.
- pipelines - will store yetl configuration for the data pipelines.
- sql - if you choose to define some tables using SQL explicitly and have yetl create those tables then the SQL files will go in here.
- schema - will hold the spark schema's in yaml for the landing data files when we autogenerate the schema's

Project config

The `yetl_test.yaml` project file tells yetl where to find these folders. It's best to leave it on the defaults:

```yaml
databricks_notebooks: ./databricks/notebooks
databricks_queries: ./databricks/queries
databricks_workflows: ./databricks/workflows
name: test_yetl
pipeline: ./pipelines
spark:
    config:
        spark.databricks.delta.allowArbitraryProperties.enabled: true
        spark.master: local
        spark.sql.catalog.spark_catalog: org.apache.spark.sql.delta.catalog.DeltaCatalog
        spark.sql.extensions: io.delta.sql.DeltaSparkSessionExtension
    logging_level: ERROR
spark_schema: ./schema
sql: ./sql
version: 3.0.0
```

It also holds a spark configuration, this is what yetl uses to create the spark session if you choose to run yetl code locally. This can be usefull for local development and testing in deveops pipelines. Since we are just leveraging pyspark and delta lake it's very easy to install a local spark environment however you will need java installed if you want to work this way.

The version attribute is also important this makes sure that you're using the right configuration schema for the specific version of yetl you're using. If these do not agree to the minor version then yetl will raise version

incompatability errors. This ensures it's clear the metadata clearly denotes the version of yetl the it requires. Under the covers yetl uses pydantic to ensure that metadata is correct when deserialized.

Logging config

The `logging.yaml` is the python logging configuration. Configure yetl logging as you please by altering this file.

SQL

Sometimes there are requirements to explicitly create tables using SQL DDL statements.

For example if you're loading a fan in pattern where multiple source tables are loading into the same destination table in parallel you cannot create the schema from the datafeed on the fly, either with or without the merge schema option. This is because delta tables are optimistically isolated and attempting the change the schema in multiple process will mostly likely cause a conflict. This is common pattern for audit tracking tables when loading bronze and silver tables (raw and base stages respectively)

You may also just find it more practical to manage table creation by explicitly declaring create table SQL statements. You can also use this feature to create views.

In order to explicitly define tables or views using SQL:

1. Create a folder in the `sql` directory that is the database name e.g. `my_database`
2. Create a .sql file in the folder called the name of the table e.g. `my_table.sql`
3. Put a deltalake compatible create SQL statement in the sql file

e.g.

```
./sql/my_database/my_table.sql
```

For example, in this project we use this feature to create an audit tracking table for loading a ronze (raw) tables called `yetl_control_header_footer.raw_audit.sql`.

Note that if we want to declare the table unmanaged and provide an explicit location we can do using the jinja variable `{{location}}` which is defined in the pipeline configuration. See the SQL documentation for full details and complete list of jinja variables supported.

Pipeline

The pipline folder holds yaml files that describe the databases, tables, files and pipeline metadata that the yetl API will stitch, render and deserialize into an api that can be used to construct data pipelines using pyspark.

A lot of thought has gone into this design to make configuration creation and management as developer friendly and minimal as possible. There are 2 core types of files:

1. Tables & files - tables.yaml contains **WHAT** we are loading
2. Pipeline - {pipeline_pattern_name}.yaml contains **HOW** we are loading it

Tables & Files - The What!

This configuration file contains the information of **WHAT** the database tables and files are that we want to load. What we mean by that, is that it's at the grain of the table and therefore each table can be configured differently if required. For example each table has it's own name. It doesn't contain any configuration of how the table is loaded.

One table definition file is allowed per project. Each file can hold definitions for any number of tables for databases at the following data stages. Database table type can be `read` (spark read api for various formats) or `delta_lake`:

- `audit_control` - `delta_lake`
- `source` - `read` or `delta-lake`
- `landing` - `read`
- `raw` (bronze) - `delta-lake`
- `base` (silver) - `delta-lake`
- `curated` (gold) - `delta-lake`

Typically `read` table types would be used in `landing` and `delta-lake` tables for everything else. `source` may not normally be required for Databricks pipelines since data will be landed using some other cross domain data orchestration tool e.g. Azure Data Factory. However yetl does support `source` for `read` and `delta-lake` tables types it can be usefull when:

- data doesn't need to be landed and is available from some via a spark source
- migrating tables from an older deltalake into a new architecute e.g. Unity Catalog migration
- landing data directly from api's

It is therefore opinionated about the basic levels that should be implemented in terms of stages and follows the medallion lakehouse arhitecture. It's rigidity starts and ends there however; you can put what ever tables you want in any layer, in any datbase with dependencies entirely of your own choosing.

tables.yaml example

By far the biggest hurdle is defining the table configuration. To rememedy this the yetl CLI tool has command to convert an Excel document into the required yaml format. Curating this information in Excel is far more user friendly than a yaml document for large numbers of tables. See the pipeline documentation for full details.

```
python -m yetl import-tables ./pipelines/tables.xlsx ./pipelines/tables.yaml
```

tables.xlsx -> tables.yaml

Pipeline - The How!

This configuration file contains pipeline configuration properties that describes **HOW** the tables are loaded. For a given feed that is landed and loaded to silver (base) tables a lot of these are the same for every table in the feed. There can be more than one of these files for a given project. In other words you can define more than one way to load your tables and parameterise the pattern in the pipeline.

An example use case of this might be to batch load a large migration dataset; then turn on autoloader and event load incrementals from that position onwards.

batch.yaml pipeline example

Summary

So what does yetl do that provides value, since all we have is a bunch of configuration files:

1. Configuration Stitching
2. Jinja Rendering
3. Validation
4. API

Configuration Stitching

It stitches the table and pipeline data together using a minimal python idioms and provides easy access to the table collections and pipeline properties. Some properties fall on a gray line in the sense that sometimes they are the same for all tables but on specific occaisions you might want them to be different. For example deltalake properties. You can define them in the pipeline doc but override them if you want on specific tables.

Jinja Rendering

There are some jinja templating and features built into the configuration deserialization. This can avoid repetition allowing for easier maintenance; or provide enhanced functions around injecting time slice parameters of a specific format into file paths. See the pipeline documentation for specific details.

Validation

When yetl uses pydantic to validate and desrialize the configuration data making it robust, easy to use and easy to extend and support.

API

The yetl API exposes the validated, stitched and rendered configuration into an easy to use API that can used to implement modular pipelines exactly how you want to. **Yetl does not take over specifically the way engineers want to build and test their pipelines**.

It does however provide an easy to use API to iterate and access the metadata you need. The tables are indexed and can easily be retrieved or iterated. So you can build your custom pyspark pipelines or even just ignore some of the managed table properties and iterate the tables to create DLT pipelines in databricks.

On-Board Datafeed

These are the outline steps to on-board a new datafeed:

1. Load Landing Data
2. Create Yetl Project
3. Create Table Metadata
4. Create Pipeline Metadata
5. Create Spark Schema
6. Develop & Test Pipeline
7. Deploy Product

Load Landing Data

This project is about loading data from cloud blob into databricks deltalake tables. Data would normally be orchestrated into landing using some other tool for example Azure Data Factory or AWS Data Pipeline or some other tool that specialises in securely wholesale copying datasets into the cloud or between cloud services.

You may already have this orchestration in place in which case the data will in your landing blob location already or you mock it by getting a sample of data and manually copying it into the location it will be landed to.

For test driven development you can include a small vanilla hand crafted data set into the project itself that can automatically be copied into place from the workspace files as way of creating repeatable end to end integration tests that can staged, executed and torn down with simple commands.

Create Yetl Project

Create a directory, setup virtual python environment and install yetl.

```
mkdir my_project
cd my_project
python -m venv venv
source venv/bin/activate
pip install yetl-framework
```

Create yetl project scaffolding.

```
python -m yetl init my_project
```

Create Table Metadata

Fill out the spreadsheet template with landing and deltalake architecture that you want to load. The excel file has to be in a specific format other the import will not work. Use this example as a template

NOTE:

- Lists are entered using character return between items in an Excel cell.
- Dicts are entered using a : between key value pairs and character returns betweenitmes in an Excel cell.

merge_column.column	required	type	description
stage	y	[audit_control, landing, raw, base, curated]	The architecural layer of the data lake house you want the DB in
table_type	y	[read, delta_lake]	What type of table to create, read is a spark.read, delta_lake is a delta table.
catalog	n	str	name of the catalog. Although you can set it here in the condif the api allows passing it as a parameter also.
database	y	str	name of the database
table	y	str	name of the table
sql	n	[y, n]	whether or not to include a default link to a SQL ddl file for creating the table.
id	n	str, List[str]	a column name or list of column names that is the primary key of the table.
depends_on	n	List[str]	list of other tables that the table is loaded from thus creating a mapping. It required the yetl index which is `stage.database.table` you can also use `stage.database.*` for exmaple if you want to reference all the tables in a database.
deltalake.delta_properties	n	Dict[str,str]	key value pairs of databricks delta properties
deltalake.identity	n	[y, n]	whether or not to include an indentity on the table when a delta table is created implicitly
deltalake.partition_by	n	str, List[str]	column or list of columns to

			partition the table by
deltalake.delta_constraints	n	Dict[str,str]	key value pairs of delta table constraints
deltalake.z_order_by	n	str, List[str]	column or list of columns to z-order the table by
deltalake.vacuum	n	int	vaccum threshold in days for a delta table
warning_thresholds.invalid_ratio	n	float	ratio of invalid to valid rows threshold that can be used to raise a warning
warning_thresholds.invalid_rows	n	int	number of invalid rows threshold that can be used to raise a warning
warning_thresholds.max_rows	n	int	max number of rows thresholds that can be used to raise a warning
warning_thresholds.mins_rows	n	int	min number of rows thresholds that can be used to raise a warning
error_thresholds.invalid_ratio	n	float	ratio of invalid to valid rows threshold that can be used to raise an exception
error_thresholds.invalid_rows	n	int	number of invalid rows threshold that can be used to raise an exception
error_thresholds.max_rows	n	int	max number of rows thresholds that can be used to raise an exception
error_thresholds.mins_rows	n	int	min number of rows thresholds that can be used to raise an exception
custom_properties.process_group	n	any	customer properties can be what ever you want. Yetl is smart enough to build them into the API
custom_properties.rentention_days	n	any	
custom_properties.anything_you_want	n	any	

Create the tables.yaml file by executing:

```
python -m yetl import-tables ./my_project/pipelines/tables.xlsx
./my_project/pipelines/tables.yaml
```

Create Pipeline Metadata

In the `./my_project/pipelines` folder create a yaml file that contains the metadata specifying how to load the tables defined in `./my_project/pipelines/tables.yaml`. You can call them whatever you want and you can create more than one. Perhaps one that batch loads and another that event stream loads. The yetl api will allow you to parameterise which pipeline metadata you want to use. For the purpose of these docs we will refere to this pipeline as `my_pipeline.yaml`.

The pipeline file `my_pipeline.yaml` has a relative file reference to `tables.yaml` and the therefore yetl knows what files to use to stitch the table metadata together.

Please see the pipeline reference documentation for details. Here is an example.

Create Spark Schema

Once the yetl metadata is in place we can start using the API. The 1st task is to create the landing schema that need to load the data. This can be done using a simple notebook on databricks.

Using databricks repo's you can clone your project into databricks.

This must be in it's own cell:

```
%pip install yetl-framework==3.0.0
```

Executing the following code will load the files and save the spark schema into the `./my_project/schema' directory in yaml format making it easy to review and adjust if you wish. There's no reason to move the files anywhere else once created, yetl uses this location as a schema repo. The files will named after the tables making it intuitive to understand what the schema's are and how the map.

The ad works example project shows this notebook approach working very well creating the schema over a relatively large number of tables.

```
from yetl import (
  Config, Read, DeltaLake, Timeslice
)
import yaml, os

def create_schema(
  source:Read,
  destination:DeltaLake
):

  options = source.options
  options["inferSchema"] = True
  options["enforceSchema"] = False

  df = (
    spark.read
    .format(source.format)
    .options(**options)
    .load(source.path)
  )

  schema = yaml.safe_load(df.schema.json())
  schema = yaml.safe_dump(schema, indent=4)

  with open(source.spark_schema, "w", encoding="utf-8") as f:
    f.write(schema)

project = "my_project"
pipeline = "my_pipeline"

# Timeslice may be required depending how you've configured you landing area.
# here we just using a single period to define the schema
# Timeslice(year="*", month="*", day="*") would use all the data
# you have which could be very inefficient.

# This exmaple uses the data in the landing partition of 2023-01-01
# how that is mapped to file and directories the my_pipeline definition
config = Config(
  project=project,
  pipeline=pipeline,
  timeslice=Timeslice(year=2023, month=1, day=1)
)

tables = config.tables.lookup_table(
  stage=StageType.raw,
  first_match=False
)

for t in tables:
  table_mapping = config.get_table_mapping(
    t.stage, t.table, t.database, create_table=False
  )
  create_schema(table_mapping.source, table_mapping.destination)
```

As you can see using this approach can also be used for creating tables in a pipeline step prior to any load pipeline using the `create_table` parameter. It will either create explicity defined tables using SQL DML if you've configured any or just create register empty delta tables with no schema. This may be required if you have multiple sources flowing into a single table (fan-in) to avoid transaction isolation errors creating the tables the 1st time that the pipeline runs.

Develop & Test Pipeline

TODO

Deploy Product

TODO

Table Configuration

The table configuration defines that is loaded in a data pipeline.

The table metadata is the most difficult and time consuming metadata to curate. Therefore yetl provides a command line tool to convert an excel curated definition of metadata into the required yaml format. Curating this kind of detail for large numbers of tables is much easier to do in an Excel document due to it's excellent features.

Example

A solution lands 3 files:

- `customer_details_1`
- `customer_details_2`
- `customer_preferences`

The files are loaded into raw from landing with a deltalake table for each file.

Those tables are then loaded into base tables. `customer_details_1` and `customer_details_2` are unioned together and loaded into a customer table. So the base tables are:

- `customers`
- `customer_perferences`

Each file has a header and footer with some audit data we load this with some other etl audit data into deltalake audit tables:

- `header_footer`
- `raw_audit`
- `base_audit`

Here is the `tables.yaml` metadata that describes the stages, databases and tables:

```yaml
# yaml-language-server: $schema=./json_schema/sibytes_yetl_tables_schema.json

version: 3.0.0

audit_control:
  delta_lake:
    yetl_control_header_footer_uc:
      catalog: development
      base_audit:
        depends_on:
        - raw.yetl_raw_header_footer_uc.*
        sql: ../sql/{{database}}/{{table}}.sql
        vacuum: 168
      header_footer:
        depends_on:
        - raw.yetl_raw_header_footer_uc.*
        sql: ../sql/{{database}}/{{table}}.sql
        vacuum: 168
      raw_audit:
        depends_on:
        - raw.yetl_raw_header_footer_uc.*
        - audit_control.yetl_control_header_footer_uc.header_footer
        sql: ../sql/{{database}}/{{table}}.sql
        vacuum: 168

landing:
  read:
    yetl_landing_header_footer_uc:
      catalog: development
      customer_details_1: null
      customer_details_2: null
      customer_preferences: null

raw:
  delta_lake:
    yetl_raw_header_footer_uc:
      catalog: development
      customer_details_1:
        custom_properties:
          process_group: 1
          rentention_days: 365
        depends_on:
        - landing.yetl_landing_header_footer_uc.customer_details_1
        exception_thresholds:
          invalid_rows: 2
          min_rows: 1
        id: id
        vacuum: 168
        z_order_by: _load_date
      customer_details_2:
        custom_properties:
          process_group: 1
          rentention_days: 365
        depends_on:
        - landing.yetl_landing_header_footer_uc.customer_details_2
        exception_thresholds:
          invalid_rows: 2
          min_rows: 1
        id: id
        vacuum: 168
        z_order_by: _load_date
      customer_preferences:
        custom_properties:
```

```
          process_group: 1          rentention_days: 365        depends_on:        -
landing.yetl_landing_header_footer_uc.customer_preferences        exception_thresholds:
invalid_rows: 2          min_rows: 1        id: id        vacuum: 168        z_order_by:
_load_datebase:  delta_lake:    yetl_base_header_footer_uc:      catalog: development
customer_details_1:        custom_properties:          process_group: 1          rentention_days:
365        depends_on:        - raw.yetl_raw_header_footer_uc.customer_details_1
exception_thresholds:          invalid_rows: 0          min_rows: 1        id: id        vacuum:
168        z_order_by: _load_date      customer_details_2:        custom_properties:
process_group: 1          rentention_days: 365        depends_on:        -
raw.yetl_raw_header_footer_uc.customer_details_2        exception_thresholds:
invalid_rows: 0          min_rows: 1        id: id        vacuum: 168        z_order_by:
_load_date      customer_preferences:        custom_properties:          process_group: 1
rentention_days: 365        depends_on:        -
raw.yetl_raw_header_footer_uc.customer_preferences        exception_thresholds:
invalid_rows: 0          min_rows: 1        id: id        vacuum: 168        z_order_by:
_load_date      customers:        custom_properties:          process_group: 1
rentention_days: 365        depends_on:        - raw.yetl_raw_header_footer_uc.*
exception_thresholds:          invalid_rows: 0          min_rows: 1        id: id        vacuum:
168        z_order_by: _load_date
```

Specification

If you use the `yetl` cli to create a project using `python -m yetl init <my_project>` then the json validation schema for the config files including table the table config will be created at `./<my_project>/piplines/json-schemas/sibytes_yetl_tables_schema.json`. Using vscode and the RedHat yaml extension you can add the following json schema reference to `./<my_project>/piplines/tables.yml` to provide live validation and intellisense:

```
# yaml-language-server: $schema=./json_schema/sibytes_yetl_tables_schema.json
```

This reference describes the required format of the `tables.yaml` configuration.

```
version: major.minor.patch

<stage:Stage>:
    <table_type:TableType>:
        delta_properties:
          <property_name>: str
        <database_name:string>:
            catalog: str|null
            <table_name:str>:
                id: str|list[str]|null
                depends_on: index|list[index]|null
                delta_properties:
                  <property_name>: str
                delta_constraints:
                  <constraint_name>: str
                custom_properties:
                  <property_names>: str
                z_order_by: str|list[str]|null
                partition_by: str|list[str]|null
                cluster_by: str|list[str]|null
                vacuum: int|null
                sql: path|null
                warning_thresholds:
                    invalid_ratio: float
                    invalid_rows: int
                    max_rows: int
                    min_rows: int
                exception_thresholds:
                    invalid_ratio: float
                    invalid_rows: int
                    max_rows: int
                    min_rows: int

            <table_name:str>:
              # table details
              ...
            <table_name:str>:
              # table details
              ...
```

version

Version is the version number of yetl that the metadata is compatible with. If the major and minor version are not the same as the yetl python libary that you're using to load the metadata then an error will be raised. This is to ensure the metadata is compatible with the version of yetl that you're using.

Example:

```
version: 3.0.0
```

Stage

The stage of the datalake house architecture. Yetl supports the following `stage`s:

- `audit-control` - define tables for holding etl data and audit logs

- `landing` - define landing object store where files are copied into you your cloud storage before they uploaded into the delta lakehouse
- `raw` - define databases and tables for the bronze layer of the datalake. These will typically be deltalake tables loading with landing data with nothing more than schema validation applied
- `base` - define databases and deltalake tables for the silver layer of the datalake. These tables will hold data loaded from raw with data quality and cleansing applied.
- `curated` - define databases and deltalake tables for the gold layer of the datalake. These tables will hold the results of heavy transforms that integrate and aggregate data using complex business transformations specifically for business requirements.

At least 2 stages must defined:

- `landing`
- `raw`

These stages are optional:

- `audit_control`
- `base`
- `curated`

Example:

```
audit_control:
  delta_lake:
...
landing:
  read:
...
raw:
  delta_lake:
```

delta_properties

Deltalake properties is an object of key-value pairs that describes the deltalake properties. They can be defined at the table type level or the table level. The lowest level of granularity takes precedence over the higher levels. So you can define properties at a high level but override them at the table level if a table has specific properties that need to be defined.

Example:

```
delta_properties:
  delta.appendOnly: true
  delta.autoOptimize.autoCompact: true
  delta.autoOptimize.optimizeWrite: true
  delta.enableChangeDataFeed: false
```

delta_constraints

Deltalake properties is an object of key-value pairs that describes the deltalake constraints. The key is the constraint name and the value is the sql constraint.

The constraints are added when yetl creates the tables.

```
delta_properties:
  dateWithinRange: "(birthDate > '1900-01-01')"
  validIds: "(id > 1 and id < 99999999)"
```

TableType

Table type is the type of table that is used. Yetl supports the following `table_type`s:

- `read` - These are tables that are read using the spark read data api. Typically these are files with various formats. These types of tables are typically defined on the `landing` stage of the datalake.
- `delta_lake` - These are deltalake tables that written to and read from during a pipeline load.

Example:

```
audit_control:
  delta_lake:
...
landing:
  read:
...
raw:
  delta_lake:
```

index

Index is a string formatted specifically to describe a table index. In the Yetl api the tables are index and the index can be used to quickly find and define dependencies.

The index takes the following form:

```
stage.database.table
```

It supports a wild card form for defining or finding a collection of tables e.g.

- `stage.*.*` - return/configure all the tables in a stage
- `stage.database.*` - return/configure all the tables in a database

Example:

```
audit_control:
  delta_lake:
    yetl_control_header_footer:
      base_audit:
        # this table depends on all the tables in the raw database called yetl_raw_header_footer
        depends_on:
        - raw.yetl_raw_header_footer.*
```

id

`id` is a string or list of strings that is the columns name or names of the table uniqie identifier.

z_order_by

`z_order_by` is a string or list of strings that is the columns name or names to z_order the table by.

partition_by

`partition_by` is a string or list of strings that is the columns name or names to partition the table by.

sql

`sql` is relative path to the directory that holds a file container the explicit SQL to create the table. Note that jinja varaiable can be used for database and table thus defining that the sql directory is structured by database and table.

Example:

```
sql: ../sql/{{database}}/{{table}}.sql
```

thresholds

Thresholds allow to define ETL audit metrics for each table. There are 2 properties for this:

- warning_thresholds - used to define metrics that if exceeded raises a warning
- exception_thresholds - usde to define metrics that if exceeded raises an exception

This is just metadata so how you use it and handle this metadata is entirely down to the developmer however. The pipeline code it self is used to calculate what these values are and compare them to the these thresholds and take appropriate action.

Each threshold type supports the following metrics:

- `invalid_ratio` - number of invalid records divided by the total number of records
- `invalid_rows` - number of invalid records
- `min_rows` - minimum number of rows
- `max_rows` - maximum number of rows

Example:

```
warning_thresholds:
    # if more than 10% of the rows are invalid then raise a warning.
    invalid_ratio: 0
    invalid_rows: 0
    max_rows: null
    # if there's less than 1000 records raise a warning
    min_rows: 1000
exception_thresholds:
    # if more than 50% of the rows are invalid then raise an exception
    invalid_ratio: 0.5
    invalid_rows: null
    max_rows: null
    # if there's less than 1 record raise an exception
    min_rows: 1
```

vacuum

`vacuum` is the day threshold over which to apply the vacuum statement.

custom_properties

`custom_properties` is object of key value pairs for anything that you want to define that's not in the specification. This feature allows yetl to be very flexible for any additional requirement that you may have.

Example:

```
custom_properties:
    # define an affinity group to process tables on the same job clusters
    process_group: 1
    # define the days to retain the data for after which it is archived or deleted
    rentention_days: 365
```

Python API

The goal of yetl is to easily declare data pipeline configuration. Once declared you'll want to access that data strcutures that you've declared using python. This reference shows the canonical patterns to load up the configuration and use it however you see fit.

Initilise

Each time you want to access the configuration using the API you will need to deserialize the configuration into python objects held in memory.

Load Configuration

The following example loads the configuration from the following configuration files into an instance of the Config object:

- ./my_project/pipelines/tables.yaml
- ./my_project/pipelines/batch.yaml

```
from yetl import (
  Config, StageType
)

config = Config(
  project="my_project",
  pipeline="batch"
)
```

The full Config constructor has the following arguments:

- `project: str`

Name for the project.

- `pipeline: str`

Name of the pipeline config file without the extension that has the config you want to use.

- `timeslice: Timeslice = None`

The timeslice that you want to inject into the config and replace the date time mask jinja variables in the configuration (see the Timeslice section). The timeslice is optional since you may just want to pull back a collection of delta tables for operations other than loading data or load the current UTC datetime which is the default.

- `config_path: str = None`

Yetl will do it's best to figure out where the configuration is located based on your project file confg, operating env and the project configuration file. If you're not using the standard settings for whatever reason you can provide an explicit path to where youtr configuration resides.

Inject Timeslice

Yetl provides a timeslice object for the convenience of injecting time periods into custom formats on data file paths and file names. For example it's typical to land data files partitioned as follows:

```
/Volumes/development/my_project/customer/2023/07/30/customer-20230730.json
```

In the pipeline configuration we generalise this by declaring the following expression definition. This tells yetl where to insert the timeslice using what datetime format. The datatime format is expressed as a python datetime format. There are 2 formats because the filename and path datetime format can be different:

```
    location: "/Volumes/{{catalog}}/my_project/{{table}}/{{path_date_format}}"
    filename: "{{table}}-{{filename_date_format}}*.json"
    filename_date_format: "%Y%m%d"
    path_date_format: "%Y/%m/%d"
```

In some loading patterns we need to inject the period of data we want to load into the config. The `Timeslice` object is provided specifically to do this since it internally handles datetime formatting, validation and wildcard handling for bulk data loading.

Specific Day

For example loading a **specific day**, injecting this `Timeslice`:

```
from yetl import (
  Config, Timeslice
)

config = Config(
  project = "my_project",
  timeslice = Timeslice(year=2023, month=7, day=30)
  pipeline = "batch"
)
```

Will result in this path:

```
/Volumes/development/my_project/customer/2023/07/30/customer-20230730*.json
```

Partial Bulk

For example bulk **loading a year**, injecting this `Timeslice`:

```
from yetl import (
  Config, Timeslice
)

config = Config(
  project = "my_project",
  timeslice = Timeslice(year=2023, month="*", day="*")
  pipeline = "batch"
)
```

Will result in this wildcard path:

```
/Volumes/development/my_project/customer/2023/*/*/customer-2023***.json
```

Full Bulk

For example **all time**, injecting this `Timeslice`:

```
from yetl import (
  Config, Timeslice
)

config = Config(
  project = "my_project",
  timeslice = Timeslice(year="*", month="*", day="*")
  pipeline = "batch"
)
```

Will result in this wildcard path:

```
/Volumes/development/my_project/customer/*/*/*/customer-****.json
```

Note:

If you're stream loading using databricks cloud files and trigger now, you don't need to worry about timeslice loading your data since databricks will automatically track and checkpoint the files that you're loading. However batch stream loading also has some downsides, but not worry since yetl has you covered making it easy to inject timeslice loading.

About

Made by Shaun Ryan