YETL

Yet Another ETL Framework

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

Yet Another ETL Framework