BeETL

Release 2023

Lars Scheibling

Jun 15, 2023

GENERAL INFORMATION

classes/beetl

CHAPTER

ONE

README

BeETL was born from a job as Integration Developer where a majority of the integrations we develop follow the same pattern - get here, transform a little, put there (with the middle step frequently missing altogether).

After building our 16th integration between the same two systems with another manual template, we decided to build BeETL. BeETL is currently limited to one datasource per source and destination per sync, but this will be expanded in the future. One configuration can contain multiple syncs.

Note: Even though most of the configuration below is in YAML format, you can also use JSON or a python dictionary.

1.1 Todo:

- [ ] Soft Delete/Hard Delete

1.2 TOC

1.3 Installation

1.3.1 From PyPi

```
pip3 install beetl
```

1.3.2 From Source

```
git clone https://
python3 setup.py install
```

1.4 Quick Start

The following is the minimum amount of configuration needed to get started with a simple sync

```
from src.beetl.beetl import Beetl, BeetlConfig

sync_config = {
    # The version of the config file, currently V1
    "version": "V1",

    # The datasources to move data between
    "sources": [
        {
            # The identifier for the datasource
            "name": "mysql_db",

            # The type (ex. Sqlserver, Rest, Itop)
            "type": "Mysql",

            # The connection settings for the datasource (connection string or host/user/
↪password)
            "connection": {
              "settings": {
                "connection_string": "mysql://user:password@host:3306/database"
              }
            }
        },
        {
            "name": "postgres_db",
            "type": "Postgres",
            "connection": {
              "settings": {
                "connection_string": "postgresql://user:password@host:5432/database"
              }
            }
        }
    ],
    # The configuration for the sync(s) to run
    "sync": [
        {
            # The source and destination identifiers
            "source": "mysql_db",
            "destination": "postgres_db",

            # The configuration for source/destination
            "sourceConfig": {
```

(continues on next page)

(continued from previous page)

```
            # The query with data to fetch
            "query": "SELECT field1, field2, field3 FROM table1",

            # The column descriptions for the query
            "columns": [
                {
                    # The name of the column/field
                    "name": "field1",

                    # The data type
                    "type": "Int32",

                    # Whether the column is considered unique
                    # (unique cols will be used for comparison)
                    "unique": True
                },
                {
                    "name": "field2",
                    "type": "Utf8",
                    "unique": False
                },
                {
                    "name": "field3",
                    "type": "Utf8",
                    "unique": False
                }
            ]
        },
        "destinationConfig": {
            # The table to insert data into
            "table": "table1",

            # The columns to insert data into
            "columns": [
                {
                    # The name of the column/field
                    "name": "field1",

                    # The data type
                    "type": "Int32",

                    # Whether the column is considered unique
                    # (unique cols will be used for comparison)
                    "unique": True
                },
                {
                    "name": "field2",
                    "type": "Utf8",
                    "unique": False
                },
                {
                    "name": "field3",
```

(continues on next page)

(continued from previous page)

```
                        "type": "Utf8",
                        "unique": False,

                        # Will be created on insert, but not updated
                        "skip_update": True
                    }
                ]
            },
            "sourceTransformers": {},
            "insertionTransformers": {}
        }
    ]
}
```

1.4.1 Secrets from Environment Variables

In case you want to save your secrets in environment variables instead of in the yaml configuration file, you can save them as a json object to an environment variable and replace the "sources"-section with sourcesFromEnv setting.

Note that the "sources" and "sourcesFromEnv" options are mutually exclusive.

```
sync_config = {
    # The version of the config file, currently V1
    "version": "V1",

    # Fetch source configuration from environment variable BEETL_SOURCES
    "sourcesFromEnv": "BEETL_SOURCES",

    # The datasources to move data between
    "sync": [
        .....
```

```
version: "V1"
sourcesFromEnv: "BEETL_SOURCES"
sync:
  - ......
```

```
{
    "version": "V1",
    "sourcesFromEnv": "BEETL_SOURCES",
    "sync": [
        ......
```

The format of the sources configuration is the same as the one normally under the "sources"-section:

```
[
    {
        # The identifier for the datasource
        "name": "mysql_db",

        # The type (ex. Sqlserver, Rest, Itop)
```

(continues on next page)

(continued from previous page)

```
        "type": "Mysql",

        # The connection settings for the datasource (connection string or host/user/
↪password)
        "connection": {
            "settings": {
            "connection_string": "mysql://user:password@host:3306/database"
            }
        }
    },
    {
        "name": "postgres_db",
        "type": "Postgres",
        "connection": {
            "settings": {
            "connection_string": "postgresql://user:password@host:5432/database"
            }
        }
    }
]
```

CHAPTER

TWO

TO DO

- [ ] Add dry-run option
- [ ] Add support for various databases
- [ ] Add support for various APIs
- [ ] Add support for various file formats
- [ ] Add support for various generation (UUID, faker, etc.)

CHAPTER

THREE

BACKGROUND

Beetl was born out of an endless list of tickets for "well, if we can't have access to the data, can we have a copy on [insert-server-here]?", for which a lot of manual work was previously done. We've previously been using a long list of different tools for this, everything from SQL Server replication and small scripts to export/import databases every night (which is a pain to maintain given that the database loses all knowledge of users on import), small scripts to syncronize Azure or AD users to a database, some data from a database has to go to an XML file and be sent here and there.

This all led to the idea behind this software; a simple, easy to use, low/no-code (even though I don't love the term, that's more or less what it's supposed to be) way for even our slightly less technical system administrators and colleagues to be able to set up the skeleton of a sync (with supervised pushes to production).

Out of that idea, BeETL was born.

Starting off, our list of specifications was as follows:

- Written in Python, Go or Rust for efficiency (We'll get back to that efficiency statement about Python later....)
- For simpler syncs, no code should be required and should in the long run work with a simple configuration interface
- A workflow that enables either basic transformation of data via included tools, or more advanced transformation through code
- Easy extensibility for new datasources and transformers
- Logging and observability - we want to be able to see what's going on and where things are going wrong
- Last, but not least, we don't want to re-create entire databases or "just update all changed rows so it's always the latest data" - we wanted a program that would only update the changed rows in the respective data source so that we'd have a better transaction log of what actually changed between runs.

The first version of BeETL was written in Go, but we quickly found that although we could have reached a similar level of performance as with Python, the amount of code was quite a bit larger than we'd like and we currently only have one proficient Go developer. Rust hasn't been adopted at our workplace yet, so we put that aside for the moment and had a look at Python and Pandas.

We'd worked with Pandas before, and it was usually quite quick and performant but once dealing with datasets running in the millions of rows, it started to become somewhat of a bottleneck. After some searching, we fell upon Polars, a Python library loosely based on Pandas but with a focus on performance.

The performant parts of Polars are written in Rust, meaning we'd get all of (well, most of) the performance benefits with none of the low-level code, hooray!

After adopting one of our earlier pieces of code, the comparator (based on Pandas) to the polars equivalent, we found the performance increase compared to Pandas was massive. On comparisons that Pandas took 30 seconds to one minute, Polars completed in just under a second; we couldn't have been happier about that.

When that was done, it was just a matter of writing the rest of the code around it, and we were done!

That was 6 months ago, and since then we've been using a (very duct-taped together version of) BeETL since, and we're finally getting to actually cleaning up the code, documenting the software and making it open source.

CHAPTER

FOUR

BEETL STRUCTURE

BeETL is based on three main components, which are explained in more detail below:

- Synchronizations
- Data Sources
- Transformers

The easiest way to explain this is with an example. Let's say you want a database full of your Azure users, with some minor modifications along the way.

4.1 Example

The example below will cover Config, Source Transformers, Field Transformers and Sync with a YAML configuration file.

The data you fetch from Azure (and example):

```
userprincipalname (user@domain.com)
givenname (John)
sn (Doe)
department (IT)
```

The data format in your database (and example):

```
userprincipalname (user@domain.com)
displayname (John Doe)
department (Information Technology)
email (user@domain.com)
username (user)
domain (domain.com)
```

This means we want to do the following:

1. Fetch data from Azure
2. Translate the Azure "department" column to the extended names (source transformer)
3. Split the "userprincipalname" column into "username" and "domain" (field transformer)
4. Join the "givenname" and "sn" columns into "displayname" (field transformer)
5. Clone the "userprincipalname" column into "email" (field transformer)
6. Run the sync

You'd create a configuration as follows (can be done in YAML, Python or JSON):

```yaml
version: "V1"
datasources:
  # Azure AD
  - name: "azusers"
    type: "MSGraph"
    connection:
      graph_object: "users"
      tenant_id: "tenant_id"
      client_id: "client_id"
      client_secret: "client_secret"
      client_scope: "https://graph.microsoft.com/.default"
      client_type: "client_credentials"
    config:
      columns:
        - name: "userprincipalname"
          type: "Utf8"
          unique: True
          skip_update: True

        - name: "givenname"
          type: "Utf8"
          unique: False
          skip_update: False

        - name: "sn"
          type: "Utf8"
          unique: False
          skip_update: False

        - name: "department"
          type: "Utf8"
          unique: False
          skip_update: False

    # Database
  - name: "mydatabase"
    type: "SQLServer"
    connection:
      connection_string: "mssql+pyodbc://user:password@server/database"
      fast_executemany: True
    config:
      columns:
        - name: "userprincipalname"
          type: "Utf8"
          unique: True
          skip_update: True

        - name: "displayname"
          type: "Utf8"
          unique: False
          skip_update: False
```

(continues on next page)

(continued from previous page)

```
        - name: "department"
          type: "Utf8"
          unique: False
          skip_update: False

        - name: "email"
          type: "Utf8"
          unique: False
          skip_update: False

        - name: "username"
          type: "Utf8"
          unique: True
          skip_update: False

        - name: "domain"
          type: "Utf8"
          unique: False
          skip_update: False

sync:
  - source: "azusers"
    destination: "mydatabase"
    sourceTransformer: "mycustom.transformer"
    fieldTransformers:

        # Split userprincipalname into username and domain
      - transformer: "strings.split"
        config:
          inField: "userprincipalname"
          outFields:
            - "username"
            - "domain"
          separator: "@"

        # Join givenname and sn into displayname
      - transformer:
        config:
          inFields:
            - "givenname"
            - "sn"
          outField: "displayname"

      - transformer: "frames.clone_field"
        config:
          inField: "userprincipalname"
          outField: "email"

      # The above transformers will always preserve the original data,
      # you can use the frames.drop_field transformer to remove those.
      # Although, it's not necessary since the source column specification
```

(continues on next page)

(continued from previous page)

```
      # is used to determine which columns to compare.

        - transformer: "frames.drop_field"
            config:
            inField: "givenname"
```

And using the following Python code:

```
from beetl.beetl import Beetl
from beetl.transformers.interface import register_transformer

# Register your custom transformer
@register_transformer('source', 'mycustom', 'transformer')
def translate_department(dataset: polars.DataFrame) -> polars.DataFrame:
    return dataset.with_column(
        polars.col('department').str.replace('IT', 'Information Technology')
    )

# Create a Beetl instance with the configuration
beetlsync = Beetl.from_yaml("config.yaml", "utf-8")

# Start the sync
beetlsync = beetlsync.sync()
```

4.2 Datasources

A datasource is a connection to a storage unit for data, such as a database, file, API, manually specified or faked data. In this example, the two datasources are “MSGraph” and “SQLServer”.

If we start by taking a look at the overall structure, a datasource has three configuration parts:

```
datasources:
  - name: "azusers"
    type: "MSGraph"
    connection:
      graph_object: "users"
      tenant_id: "tenant_id"
      client_id: "client_id"
      client_secret: "client_secret"
      client_scope: "https://graph.microsoft.com/.default"
      client_type: "client_credentials"
    config:
      columns:
        - name: "userprincipalname"
          type: "Utf8"
          unique: True
          skip_update: True
```

Name is used to identify the datasource when specifying the sync later on, type identifies which connector to use for the connection.

In the “connection” settings, you specify the details for how to connect and to what.

In the "config" settings, you describe the data that is to be retrieved from the source.

The "columns" section will determine how the comparison is made by looking at the unique and skip_update fields, the "type" field will ensure the data from both sides is in the same format.

4.3 Synchronizations

A synchronization is a description of the process to follow when retrieving, comparing and updating data in the destination. Given this example:

```
sync:
  - source: "azusers"
    destination: "mydatabase"
    sourceTransformer: "mycustom.transformer"
    fieldTransformers:

        # Split userprincipalname into username and domain
      - transformer: "strings.split"
        config:
          inField: "userprincipalname"
          outFields:
            - "username"
            - "domain"
          separator: "@"

        # Join givenname and sn into displayname
      - transformer: "strings.join"
        config:
          inFields:
            - "givenname"
            - "sn"
          outField: "displayname"
          separator: " "

      - transformer: "frames.clone_field"
        config:
          inField: "userprincipalname"
          outField: "email"

      # The above transformers will always preserve the original data,
      # you can use the frames.drop_field transformer to remove those.
      # Although, it's not necessary since the source column specification
      # is used to determine which columns to compare.

        - transformer: "frames.drop_field"
            config:
            inField: "givenname"
```

4.3.1 1. Data Retrieval Stage

In this stage, the datasource specified under source will be queried for the data according to its settings. For databases, this can be done by specifying a manual query or by setting the table and columns to retrieve in the configuration above.

When the data is retrieved and loaded into memory, this starts Stage 2

4.3.2 2. Transformation Stage

In this stage, the data from the source is tranformed in such a way so that it matches the destination so that a comparison can occur on equal terms.

There are two types of tranformers, source transformers (1 per sync) which are run first and meant to provide a more advanced way of tranforming data in the form of a Python function. A simple type of source transformer is the one in the example:

```
@register_transformer('source', 'mycustom', 'transformer')
def translate_department(dataset: polars.DataFrame) -> polars.DataFrame:
    return dataset.with_column(
        polars.col('department').str.replace('IT', 'Information Technology')
    )
```

This will check the "department" column for the contents "IT" and translate them to "Information Technology".

The second type of transformer is a field transformer, which are run after the source transformers. There are a number of built-in field transformers (see list under "Classes" in the menu), you can also register your own in the same way as a source transformer. The inputs and outputs are the same, but there can be multiple field transformers per sync and they behave slightly differently.

4.3.3 3. Comparison Stage

At this stage, the data should be formatted roughly the same. If there are a couple of extra columns in the source or destination data, this shouldn't matter much. When starting the comparison, the fields of the destination are used to choose fields for comparison.

The first comparison stage is for inserts, where we compare the series of columns marked "unique" between the dataset and determine which rows are missing in the destination.

The second comparison stage is for updates, where we use the unique fields as identifiers to find the rows that are present in both the source and destination and compare them to determine which rows need to be updated.

The third comparison stage is for deletes, which basically is a reverse comparison to the inserts.

4.3.4 4. Insertion Stage

At this stage, the data is sent to the datasource class for the destination to be inserted, updated and deleted

This is a rough overview over what happens in the various steps for the software. You can find more of the technical documentation, advanced options, fields and examples in the configuration and classes-sections of the documentation.

CHAPTER

FIVE

STATIC

5.1 Static Source

5.2 Faker Source

CHAPTER

SIX

DATABASES

CHAPTER

SEVEN

STRING TRANSFORMERS

The tranformers below are called with the strings.[function_name] syntax.

For example: strings.drop_columns for the drop_columns function.

CHAPTER

EIGHT

FRAME TRANSFORMERS

The tranformers below are called with the frames.[function_name] syntax.

For example: frames.drop_columns for the drop_columns function.

CHAPTER

NINE

MAIN CLASS

CHAPTER

TEN

CONFIGURATION

CHAPTER

ELEVEN

INTERFACES

11.1 Source Interface

11.2 Transformer Interface

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search