SQL Connectors Documentation

Release 1.0.2

Diego Fernandez

Mar 04, 2019

Contents:

CHAPTER 1

SQL Connectors

A simple wrapper for SQL connections using SQLAlchemy and Pandas read_sql to standardize SQL workflow. The main goals of this project is to reduce boilerplate code when working with SQL based data sources and to enable interactive exploration of data sources in Python.

- Free software: MIT license
- Documentation: https://sql-connectors.readthedocs.io.
- Repo: https://github.com/aiguofer/sql_connectors.

1.1 Features

- Standardized client for working with different SQL datasources, including a standardized format for defining your connection configurations
- A SqlClient interface based off the SQLAlchemy `Engine` with some helpful functions like Pandas' `read_sql` and functions to leverage `reflection` from SQLAlchemy

1.2 Installation

1.2.1 Stable release

To install SQL Connectors, run this command in your terminal:

```
$ pip install --process-dependency-links sql_connectors
```

This is the preferred method to install SQL Connectors, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2.2 Dev install

The sources for SQL Connectors can be downloaded from the Github repo.

You can clone the public repository and install in development mode:

```
$ git clone git://github.com/aiguofer/sql_connectors
$ cd sql_connectors
$ pip install --process-dependency-links -e .[dev]
```

1.3 Configurations

Configurations can be stored wherever you want by implementing your own `Storage`. However, the default is `LocalStorage` reading in configuration files from `~/.config/sql_connectors`.

You can change the `Storage` class using the `SQL_CONNECTORS_STORAGE` environment variable (for example `sql_connectors.storage.LocalStorage`), and you can specify a different configuration directory or URI with `SQL_CONNECTORS_PATH_OR_URI`.

The `example_connection.json` file is provided as a template; feel free to replace this with your own connection details and re-name the file.

The contents of the example file are:

```
{
    "drivername": "sqlite",
    "relative_paths": ["database"],
    "default_env": "default",
    "default": {
        "database": "example_connection.db"
    }
}
```

The fields mean the following:

drivername (string) This required field is a SQLAlchemy dialect or dialect+driver. See the SQLAlchemy Engine documentation for more details. You may first have to install the required python modules for your dialect+driver to work if it's a third party plug-in.

relative_paths (list of strings) This optional field lets you specify if an option for your connection needs to load a file relative to your config directory. For example, if you had a connection that needed to use a cert, you could add `query.sslrootcert` to this list, set `"query": { "sslmode": "verify-ca", "sslrootcert": "certs/root.crt"}`, and drop the cert in `$SQL_CONNECTORS_CONFIG_DIR/certs/root.crt`.

default_env (string) This optional field lets you specify which environment should be used by default. If not included, it will use `default`.

default_schema (string) This optional field lets you specify which schema should be used by default. If not included, it will use `None`.

default_reflect (boolean) This optional field lets you specify whether it should reflect the data source by default. If not included, it will use `False`.

env.username (string) This optional field specifies the username for the connection. If it's left out or set to null and the driver is not 'sqlite', the user will be prompte when they try to create the client. If the connection doesn't have credentials, set this to an empty string. Should not be set for 'sqlite'.

env.password (string) This optional field specifies the password for the connection. If it's left out or set to null and the driver is not 'sqlite', the user will be prompte when they try to create the client. If the connection doesn't have credentials, set this to an empty string. Should not be set for 'sqlite'.

env.host (string) This optional field specifies the host for the connection. Should not be set for 'sqlite'.

env.port (string or integer) This optional field specifies the port for the connection. Should not be set for 'sqlite'.

env.database (string) This optional field specifies the database name for the connection. If it's a 'sqlite' connection and left empty, it will use `:memory:`. Otherwise, you can specify a relative path or an absolute path; if you want the file in your config directory, you can use the `relative_paths` property.

env.query (object) This optional field is a json object with options to pass onto the dialect and/or DBAPI upon connect.

env.allowed_hosts (list of strings) This optional field is a list of strings containing hostnames where the given credentials are accepted. If the hostname is not in the list, it will prompt the user for credentials. This was added due to some specific usecase where we share service credentials but they're only allowed on our common servers.

1.4 How-To

The module will check your available connection configurations and create variables within the top level module for each of them. It will create 2 variables for each config, `connection_name` and `connection_name_envs`; these are both functions, the first will return a `get_client` function with some defaults set based on the config, and the second will return a `get_available_envs` function that when called returns available environments for the given data source. When `reflection` is enabled, the client will hold metadata about the available tables.

Here's a basic usage example assuming the example config file exists:

```
from sql_connectors import connections
client = connections.example_connection()
client.read_sql('select 1')
```

Here's a more complex example that's pretty redundant but shows more functionality

```
from sql_connectors import connections

available_envs = connections.example_connection_envs()
client = connections.example_connection(env=available_envs[0], reflect=True)

client.read_sql('select 1').to_sql('example_table', client, if_exists='replace')
available_tables = client.table_names()
table1 = client.get_table(available_tables[0])
df = client.read_sql(table1.select())
```

1.5 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install SQL Connectors, run this command in your terminal:

```
$ pip install --process-dependency-links sql_connectors
```

This is the preferred method to install SQL Connectors, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for SQL Connectors can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/aiguofer/sql_connectors
```

Or download the tarball:

```
$ curl  -OL https://github.com/aiguofer/sql_connectors/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use SQL Connectors in a project:

```
import sql_connectors
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/aiguofer/sql_connectors/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

SQL Connectors could always use more documentation, whether as part of the official SQL Connectors docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/aiguofer/sql_connectors/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *sql_connectors* for local development.

1. Fork the *sql_connectors* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/sql_connectors.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv sql_connectors
$ cd sql_connectors/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 sql_connectors tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/aiguofer/sql_connectors/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_sql_connectors
```

4.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 5

Credits

5.1 Development Lead

- Diego Fernandez <aiguo.fernandez@gmail.com>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 1.0.0 (2019-01-14)

- BREAKING: Connections are now stored in a namespace instead of being submodules. New usage:

```
from sql_connectors import connections
client = connections.example_connection()
```

  Instead of:

```
from sql_connectors import example_connection
client = example_connection()
```

- New `Storage` abstract class can be extended to implement different backend
- Configuration is now handled by `Traitlets`. Default storage class can be specified with `SQL_CONNECTORS_STORAGE` env var and the connection string or path can be specified with `SQL_CONNECTORS_PATH_OR_URI`

6.2 0.1.0 (2018-03-20)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search