dsdbmanager

May 04, 2021

Contents

1 Introduction **3**
- 1.1 Objective . . . 3
- 1.2 Extras . . . 4
- 1.3 Footnote - Disclaimer . . . 6

2 Contributing **7**
- 2.1 Bugs, questions and feature requests . . . 7
- 2.2 Pull Requests . . . 7

3 Core Class & Function **9**

4 Manual Connecting without Storing Credentials **11**
- 4.1 Connect to Oracle Databases . . . 11
- 4.2 Connect to Mysql Databases . . . 11
- 4.3 Connect to Mssql Databases . . . 12
- 4.4 Connect to Teradata Databases . . . 12

5 Indices and tables **13**

Index **15**

`dsdbmanager` is a package to help data scientists easily manager their database information. It provides a way to connect to different SqlAlchemy dialects by only providing the `username` and `password`.

Install dsdbmanager via `pip`:

```
pip install dsdbmanager
```

The source code is on GitHub and the readme is very informative.

The package expects a `DSDBMANAGER_CONFIG` environment variable pointing to a secure folder.

CHAPTER 1

Introduction

1.1 Objective

The main goal of this project is to allow anyone working with relational databases to:

1. Create an Oracle, Mysql … engine by only providing a username and a password.
2. Store and Organize database information such as the host, port and schema etc.

That really is it. All the additional work to speed up some data science development is extra and will be explained later.

1.1.1 Setup

First, make sure to have `DSDBMANAGER_CONFIG` set as an environment variable pointing to a secure folder. This is where the database information will be stored. If the environment variable is not available, a default is directly set the first time the package is imported. That default corresponds to `pathlib.Path.home() / ".dsdbmanager"`.

To install the package simply do:

```
pip install dsdbmanager
```

Once installed, the package provides a set of shell commands to add databases or remove them.

1.1.2 Add a Database

In a shell command, type `dsdbmanager add-database` In a python interactive shell:

```
python
from dsbmanager import add_database
add_database()
```

Either approach will ask you a series of questions that will lead to the creation of a `.hosts.json` file that is well structured like so:

```
{
  "mysql": {
    "database1": {
      "name": "database1",
      "host": "localhost",
      "port": 3306
    },
    "database 2": {
      "name": "database2",
      "host": "my.host.com",
      "port": 3306
    }
  },
  "oracle": {
    "database1": {
      "name": "database1",
      "host": "oracle.myhost.com",
      "port": 1521,
      "schema": "myschema",
      "sid": "orcl"
    }
  }
}
```

Note: Keep in mind that in order to use mysql, oracle or any other dialects, the corresponding sqlalchemy extensions should be installed as well as any additional OS requirements. For example: Oracle clients and required software must be available in order to get `cx_Oracle` and `sqlalchemy` to generate a proper engine.

1.1.3 Creating Engines

Armed with a structure like the one above, it is easy to distinguish oracle databases from mysql databases etc. This means that we can easily determine the proper channels to create an engine. Sure it takes little to no effort to copy the engine configuration line from SqlAlchemy but when working with different Jupyter Notebooks, it's easier to go the following route:

```
import dsdbmanager

mysql_database1 = dsdbmanager.mysql_.Mysql('database1').create_engine(username,␣
↪password)
oracle_database1 = dsdbmanager.oracle_.Oracle('database1').create_engine(username,␣
↪password)
```

You can also pass additional Sqlalchemy arguments to the `create_engine` call.

That really is the main objective of this project.

But to build on this concept of structuring the databases and their information, there are some extra perks

1.2 Extras

If I can organize my databases this way, shouldn't I be able to have a similar file for the credentials?

Yes! Well it is definitely possible but credentials are very sensitive information. So before storing them, the package offers a way to encrypt them. This means that a credential file similar in structure to the `.hosts.json` file with totally encrypted usernames and passwords. The key to decrypt and encrypt is generated the very first time the package is imported/used.

The good thing about the credentials being stored aside is that they will never show in your jupyter notebooks or scripts. To achieve this, you can do:

```
import dsdbmanager

mysql_database1 = dsdbmanager.mysql().database1(connect_only=False)
mysql_database2 = dsdbmanager.mysql()['databases'](connect_only=False)
oracle_database1 = dsdbmanager.oracle().database1(connect_only=False)
oracle_database1_with_newschema = dsdbmanager.oracle().database1(connect_only=False,␣
↪schema='newschema')

# to access the engines, use the sqlalchemy_engine property. For example
engine = mysql_database1.sqlalchemy_engine
```

The first time a connection is being attempted, you will be asked for credentials. Those credentials will then be encrypted and stored if the connection is successful.

Why are there a `connect_only=True` and additional `schema` arguments available when connecting to the databases.

The approach above is wrapping the `sqlalchemy engine` in a `dsdbmanager.dboject.DbMiddleware` object. Please read the docs on this object. It has a property `sqlalchemy_engine` that provides the `sqlalchemy engine` but it also has **all the tables and views in the schema of the database as properties**. These properties are actually just functions so you are not reading anything from the database unless you call those functions. This is why there is an option to specify a different schema than the one specified when adding the database (because it would not make sense to have a different json entry for each schema on a database).

Something very important to note: Those functions that when called bring you data from the database, they automatically cache the data. So if somehow your function took a minute to bring the data you need, the next time you call the function, it will take no time at all. That also means that any changes on the database would not be reflected in your new function calls. That is one of the reasons why the `dsdbmanager.dboject.DbMiddleware` can be used as a context manager.

Well that's cool but perhaps you do not want to store your credentials. Maybe you want to pass your username and password to create the engine and then make use of the `dsdbmanager.dboject.DbMiddleware` class. There is a `from_engine` function for that. For example:

```
import dsdbmanager

mysql_database1_engine = dsdbmanager.mysql_.Mysql('database1').create_engine(username,
↪ password)
mysql_database1 = dsdbmanager.from_engine(mysql_database1_engine, schema="some_schema
↪")
```

This effectively simplifies some simple queries like `select * from table` or `select column1, column2 from table limit 10` for example. That is because the functions mentioned above take arguments `rows` and `columns`. Look at the source code for `dsdbmanager.dboject.table_middleware`

It is also possible to do:

```
import dsdbmanager
mysql_database1 = dsdbmanager.mysql().database1(connect_only=False)
mysl_database1.table_1(rows=10, columns=('column 1', 'column_2'), column_3 = value_1,␣
↪**{'column 4': value_2}, column_5 = (value_3, value_4))
```

The last command is equivalent to `select [column 1], column_2 from table_1 where column_3 = value_1 and [column 4] = value_2 and column_5 in (value_3, value_4)`.

1. You will have to use a dictionary to handle columns with spaces or begin with numbers for example.
2. When you provide a tuple as a value, you are indicating a `key in values` type filtering.
3. If your table names have spaces or begin with numbers for example, you couldn't use the `.` notation so you can do `mysl_database1[table 2]` for example.

1.2.1 Creating Subsets By Project

Say you are working on many projects locally and as a result you have many hosts/credentials saved. Say Project_x only uses a subset and now the project must be moved to a server. It would not make sense to move the whole set of credentials to that server and the key used locally should not be shared. It is possible to do:

```
from dsdbmanager import create_subset
create_subset({'oracle': 'db1', 'mysql': {'db1', 'db2'}, 'mssql': 'all'}, 'project_x')
```

The benefits of this is that it creates folders with new key and re-encrypted credentials. The folder can easily be moved wherever user desires. User can then move the key out of the folder and point the *DSDBMANAGER_KEY* variable to the path. Using *all* as above means that user wants to include all databases for a given dialect. This means that it is easy to re-encrypt all your credentials:

```
create_subset({'oracle': 'all', 'mysql': 'all', 'mssql': 'all'}, 'monthly_re_
↪encryption')
```

1.3 Footnote - Disclaimer

1. Saving these type of info is best on a drive that is not locally and well protected by firewall rules!
2. It is also possible to separate the encryption key from the credentials with the *DSDBMANAGER_KEY* environment variable which should point to a path!

CHAPTER 2

Contributing

The source code for `dsdbmanager` is on GitHub.

2.1 Bugs, questions and feature requests

Please help identify any bugs and make feature requests on GitHub Issues. Provide the following information with any issues encountered

1. Operating System
2. Python Version
3. Minimal, Reproducible Example - something like the StackOverflow Examples.

2.2 Pull Requests

PRs are welcomed as it would be nice to cover all SqlAlchemy dialects. PRs to improve the code or add more tests would be very much appreciated.

CHAPTER 3

Core Class & Function

class dsdbmanager.dbobject.**DbMiddleware**(*engine: sqlalchemy.engine.base.Engine, connect_only: bool, schema: str = None*)

This is the main class that is wrapped around the sqlalchemy engines

Assume I have two tables, 'table_1' and 'table 2' in my default schema for an engine

```
>>> dbobject = DbMiddleware(engine, False, None)
>>> dbobject.sqlalchemy_engine.table_names()
['table_1', 'table 2']
```

I can access the tables as they are properties or methods rather

```
>>> dbobject.table1
>>> dbobject['table 2']  # because it is not possible to use the . notation here
```

But these do not do anything, in fact they are all just functions that I can call

```
>>> dbobject.table1(rows=10)  # to get  the first 10 rows
>>> dbobject['table 2'](rows=100, columns=('column', 'column with space'))  # to␣
↪only get the specified columns
```

I can also filter my data.

Say I want column_3 in table1 to be equal to 'some_value'

```
>>> dbobject.table1(column_3='some_value')
```

If I want to get data only when column_3 is either 'some_value' or 'other_value'

```
>>> dbobject.table1(column_3=('some_value', 'other_value'))  # here I pass a␣
↪tuple instead of a single value
```

tuples are used all around simply because we cache the result of these methods i.e. the dataframes

Say I had a column name that had spaces and I couldn't just do what I did above, I could do this

```
>>> dbobject.table1(**{'column with space': 'some_value'})  # simply unpacking␣
↪the dictionary at execution time
```

All those methods to pull data are **table_middleware** functions already evaluated at engine, table name and schema level.

Bonus

Get Metadata on your table

```
>>> dbobject._metadata.table1()
```

`dsdbmanager.dbobject.`**`table_middleware`**(*engine: sqlalchemy.engine.base.Engine*, *table: str*, *schema: str = None*)

This does not directly look for the tables; it simply gives a function that can be used to specify number of rows and columns etc. When this function is evaluated, it returns a function that holds the context. That function has the table name, the schema and engine. It then knows what to query once it is called.

Parameters

- **`engine`** – the sqlalchemy engine for the database
- **`table`** – a table name as in util_function
- **`schema`** – a schema of interest - None if default schema of database is ok

Returns a function that when called, pulls data from the database table specified with 'table' arg

CHAPTER 4

Manual Connecting without Storing Credentials

4.1 Connect to Oracle Databases

Once databases are added, this class can be used to instantiate an object for the proper `database name`. The instance will have a `create_engine` method that should be used with just `username` and `password`.

class `dsdbmanager.oracle_.`**`Oracle`**(*db_name: str*, *host_dict: Dict[str, Dict[str, Dict[str, str]]] = None*)

`create_engine`(*user: str = None*, *pwd: str = None*, ***kwargs*)

Parameters

- **`user`** – username
- **`pwd`** – password
- **`kwargs`** – for compatibility/additional sqlalchemy create_engine kwargs

Returns sqlalchemy engine

4.2 Connect to Mysql Databases

Once databases are added, this class can be used to instantiate an object for the proper `database name`. The instance will have a `create_engine` method that should be used with just `username` and `password`.

class `dsdbmanager.mysql_.`**`Mysql`**(*db_name: str*, *host_dict: Dict[str, Dict[str, Dict[str, str]]] = None*)

`create_engine`(*user: str = None*, *pwd: str = None*, ***kwargs*)

Parameters

- **`user`** – username
- **`pwd`** – password

- **kwargs** – for compatibility/additional sqlalchemy create_engine kwargs

Returns sqlalchemy engine

4.3 Connect to Mssql Databases

Once databases are added, this class can be used to instantiate an object for the proper `database name`. The instance will have a `create_engine` method that should be used with just `username` and `password`.

class `dsdbmanager.mssql_.`**`Mssql`**(*db_name: str, host_dict: Dict[str, Dict[str, Dict[str, str]]] = None*)

create_engine(*user: str = None, pwd: str = None, **kwargs*)

Parameters

- **user** – username
- **pwd** – password
- **kwargs** – for compatibility/additional sqlalchemy create_engine kwargs

Returns sqlalchemy engine

4.4 Connect to Teradata Databases

Once databases are added, this class can be used to instantiate an object for the proper `database name`. The instance will have a `create_engine` method that should be used with just `username` and `password`.

class `dsdbmanager.teradata_.`**`Teradata`**(*db_name: str, host_dict: Dict[str, Dict[str, Dict[str, str]]] = None*)

create_engine(*user: str = None, pwd: str = None, **kwargs*)

Parameters

- **user** – username
- **pwd** – password
- **kwargs** – for compatibility/additional sqlalchemy create_engine kwargs

Returns sqlalchemy engine

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

C

`create_engine()` (*dsdbmanager.mssql_.Mssql method*), 12
`create_engine()` (*dsdbmanager.mysql_.Mysql method*), 11
`create_engine()` (*dsdbmanager.oracle_.Oracle method*), 11
`create_engine()` (*dsdbmanager.teradata_.Teradata method*), 12

D

`DbMiddleware` (*class in dsdbmanager.dbobject*), 9

M

`Mssql` (*class in dsdbmanager.mssql_*), 12
`Mysql` (*class in dsdbmanager.mysql_*), 11

O

`Oracle` (*class in dsdbmanager.oracle_*), 11

T

`table_middleware()` (*in module dsdbmanager.dbobject*), 10
`Teradata` (*class in dsdbmanager.teradata_*), 12