redbird

Mikael Koli

Jan 17, 2023

CONTENTS:

1 Why Repository Pattern? 3

2 Main Features 5

3 Terminology 7
- 3.1 Unit of Work . . . 7
- 3.2 Database Operations . . . 7

4 Example 9
- 4.1 Tutorial . . . 10
- 4.2 Repositories . . . 11
- 4.3 CRUD: Create, Read, Update, Delete . . . 34
- 4.4 Operations . . . 35
- 4.5 SQL Tools . . . 36
- 4.6 Logging Handler . . . 48
- 4.7 Examples . . . 49
- 4.8 References . . . 52
- 4.9 Version history . . . 59

5 Indices and tables 61

Index 63

Repository pattern is a technique to abstract the data access from the domain/business logic. In other words, it decouples the database access from the application code. The aim is that the code runs the same regardless if the data is stored to an SQL database, NoSQL database, file or even as an in-memory list.

- Source code
- Releases (PyPI)

CHAPTER

ONE

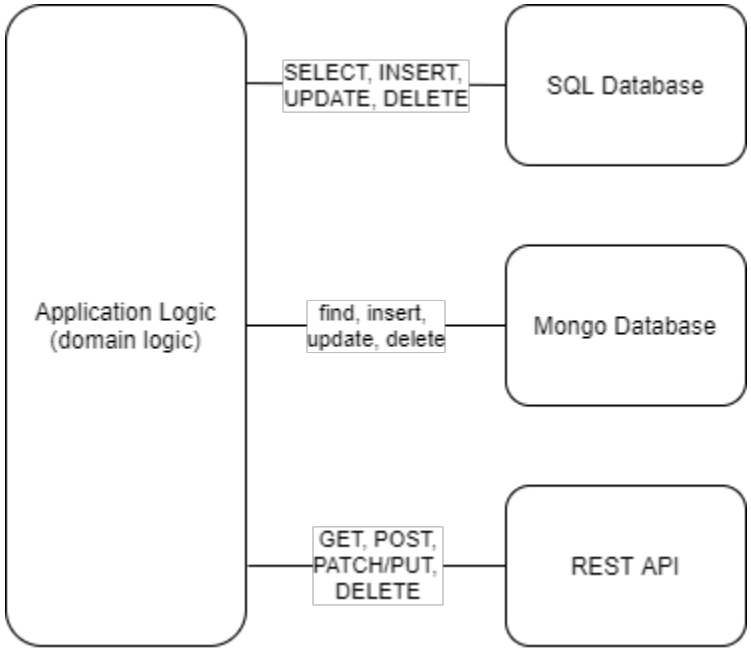
WHY REPOSITORY PATTERN?

Short Answer:

Because it simplifies things, makes prototyping faster and testing easier.

Long answer:

Typically, data access in an application looks like this:



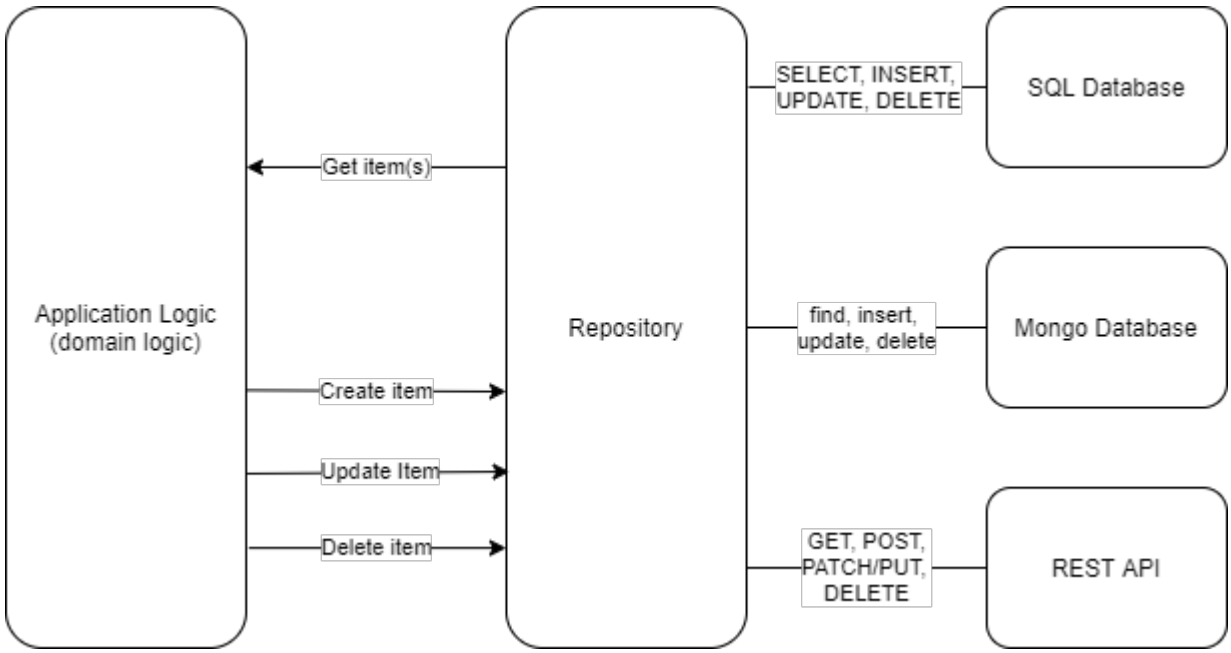
In other words, querying the databases is embedded with the application code, ie. the application code executes raw SQL, MongoDB queries or HTTP requests to APIs. This makes the application code to be data store specific. In order for the application to function, it must be connected to a specific type of a database.

For many projects this approach may not cause additional difficulities but there are several problems with this approach related to readability and maintainability:

- Application code is data store specific thus later switching to another data store may require a lot of work.
- Testing the application code is non-trivial if there is no test database of the same type as the production.
- Understanding the application requires understanding how the underlying database works.
- The code may become hard to read if multiple types of databases are used, ie. SQL databases and MongoDB databases.

Repository pattern aims to separate the domain layer (application logic) from the database layer (data access) by unifying the syntax for creating, fetching, modifying and deleting data in the data stores. It transforms generic actions to the language a specific database understands.

In practice, this is illustrated below:



The repositories act as translators to transform generic actions (read, create, update, delete data) to language the specific database understands. The repositories may be configured at the configuraitons of the application and they may easily be swapped to different database servers operating on the same or different querying language or data types. The application code is identical regardless if the data is in an SQL database, MongoDB or in-memory lists.

This has several benefits:

- Unit testing is easy as the repositories could be swapped to in-memory lists
- Database migrations is trivial as it require no code changes
- Using different databases or types of databases adds no additional maintenance costs
- Creating separate environments operating on different connections is easy.

However, there are some downsides with repository pattern as well. Most notably, some features in querying languages cannot be replicated in a way that works with all others simply due to that these features are missing in them. Especially, some optimizations are such that cannot be replicated. However, most applications don't require optimized or complex queries. Furthermore, applications that do need them may still implement repository pattern and use database specific queries only in places where this is unavoidable.

CHAPTER

TWO

MAIN FEATURES

In short, Red Bird offers:

- Identical way accross data stores of doing the following operations:
  - Create an item to the data store
  - Read items from the data store
  - Update items in the data store
  - Delete items in the data store
- Data validation via Pydantic
- Basic querying operations (ie. equal, greater or less than)
- *A log handler* for logging to data stores

Supported repositories:

- SQL (via SQLAlchemy)
- MongoDB (via Pymongo)
- In-memory (objects in Python list)
- CSV (each row is an item)
- JSON (a JSON file per item)

CHAPTER

THREE

TERMINOLOGY

3.1 Unit of Work

In Red Bird, the term *repository* is used to describe a specific data store that is a collection of items of the same type (ie. a list of cars) and *item* is used to describe a record or a document with some attributes (ie. a car with registration number 123-456-789). An attribute consists of the name of the attribute (ie. car color) and its value for specific item (ie. color of the car with registration number 123-456-789 is red).

The definition of these terms between various data stores are illustrated below:

Repository	Repository definition	Item definition	Attribute definition
Python memory	list	object	attribute (object) or item (dict)
SQL	table	row	column
MongoDB	collection	document	field
REST (HTTP)	URL endpoint	JSON object	field

3.2 Database Operations

There are four generic operations implemented by almost every data store: create, read, update and delete. These are often labeled CRUD. Read-only data store may only have read operation and some simple data stores may only have read, create and delete (such as a CSV file). For basic CRUD actions, Red Bird uses tems more commonly used in Python language: add, get, update and delete, respectively. Red bird also adds one more operation that is implemented in some data stores and could be implemented to others by combining existing operations: replace. Replace is an update but it also removes attributes that were not changed.

These unified methods for manipulating data (a single item) are illustrated below:

Repository	get	add	delete	update	replace
Python memory	list[...]	list.append	list.pop	setattr	list.pop & list.append
SQL	SELECT	INSERT	DELETE	UPDATE	DELETE & INSERT
MongoDB	findOne	insertOne	deleteOne	updateOne	deleteOne & insertOne
REST (HTTP)	GET	POST	DELETE	PATCH	PUT

In addition, operations can also be divided to those that affect only one item and to those that affect multiple items. If you search for a car with registration number 123-456-789 you should should only get one car or none whereas if you search for cars with a red color you may get multiple, one or none. The same applies for other operations as well.

CHAPTER

FOUR

EXAMPLE

First we create a simple in-memory repository that has `registration_number` as the attribute that is unique for each item:

```
from redbird.repos import MemoryRepo
repo = MemoryRepo(id_field="registration_number")
```

Create some items to the database:

```
repo.add({"registration_number": "123-456-789", "color": "red"})
repo.add({"registration_number": "111-222-333", "color": "red"})
repo.add({"registration_number": "444-555-666", "color": "blue"})
```

Get items from the database:

```
# One item
repo["123-456-789"]

# Multiple items
repo.filter_by(color="red").all()
```

Update items in the database:

```
# One item
repo["123-456-789"] = {"condition": "good"}

# Multiple items
repo.filter_by(color="blue").update(color="green")
```

Delete items from the database:

```
# One item
del repo["123-456-789"]

# Multiple items
repo.filter_by(color="red").delete()
```

4.1 Tutorial

This section covers basic tutorials of Red Bird.

4.1.1 Installation

Install the package:

```
pip install redbird
```

See PyPI for Red Bird releases.

4.1.2 Configuring Repository

The full list of built-in repositories and their examples are found from repository section <repositories>. Below is a simple example to configure in-memory repository.

```
from redbird.ext import MemoryRepo
repo = MemoryRepo()
```

By default, the items are manipulated as dictionaries. You may also create a Pydantic model in order to have better data validation and control over the structure of the items:

```
from pydantic import BaseModel

class Car(BaseModel):
    registration_number: str
    color: str
    value: float

from redbird.ext import MemoryRepo
repo = MemoryRepo(model=Car)
```

See more about configuring repositories from *here*.

4.1.3 Usage Examples

Create operation:

```
# If you use dict as model
repo.add({"registration_number": "123-456-789", "color": "red"})

# If you Pydantic model:
repo.add(Car(registration_number="111-222-333", color="red"))
repo.add(Car(registration_number="444-555-666", color="blue"))
```

Get operation:

```
# One item
repo["123-456-789"]
```

(continues on next page)

(continued from previous page)

```
# Multiple items
repo.filter_by(color="red").all()
```

Update operation:

```
# One item
repo["123-456-789"] = {"condition": "good"}

# Multiple items
repo.filter_by(color="blue").update(color="green")
```

Delete operation:

```
# One item
del repo["123-456-789"]

# Multiple items
repo.filter_by(color="red").delete()
```

4.2 Repositories

This section consists of configuring various built-in repositories. After configuration, the usage of the repositories should be identical for all features except those stated otherwise.

4.2.1 In-Memory Repository

In-memory repository is a data store that is simply a Python list in temporary memory.

```
from redbird.repos import MemoryRepo
repo = MemoryRepo()
```

Usage

Now you may use the repository the same way as any other repository. Please see:

- *Reading the repository*
- *Creating an item to the repository*
- *Deleting an item from the repository*
- *Updating an item in the repository*

Class

class redbird.repos.**MemoryRepo**(*, *model=<class 'dict'>*, *id_field=None*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=False*, *collection=[]*)

Memory repository

This is a repository which operates in memory. Useful for unit tests and prototyping.

Parameters

- **collection** (*list*) – The collection
- **model** (*Type*) – Class of an item in the repository. Commonly dict or subclass of Pydantic BaseModel. By default dict
- **id_field** (*str, optional*) – Attribute or key that identifies each item in the repository.
- **field_access** (*{'attr', 'key'}, optional*) – How to access a field in an item. Either by attribute ('attr') or key ('item'). By default guessed from the model.
- **query** (*Type, optional*) – Query model of the repository.
- **errors_query** (*{'raise', 'warn', 'discard'}*) – Whether to raise an exception, warn or discard the item in case of validation error in converting data to the item model from the repository. By default raise

Examples

```
repo = MemoryRepo()
```

```
repo = MemoryRepo(collection=[
    {"car_type": "van", "color": "red"},
    {"car_type": "truck", "color": "red"}
])
```

insert(*item*)

Insert item to the repository

Parameters **item** (*instance of model*) – Item to insert to the repository

Examples

```
repo.insert(Item(id="a", color="red"))
```

filter_by(***kwargs*)

Filter the repository

Parameters ****kwargs** (*dict*) – Query which is used to conduct furher operation.

Examples

```
repo.filter_by(color="red")
```

update(*item*)

Update item in the repository

Parameters **item** (*instance of model*) – Item to update in the repository

Examples

```
repo.update(Item(id="a", color="red"))
```

delete(*item*)

Delete item from the repository

Parameters **item** (*instance of model*) – Item to delete from the repository

Examples

```
repo.delete(Item(id="a", color="red"))
```

4.2.2 CSV Repository

CSV (Comma-separated values file) repository is a data store in which each item represents a row in the given CSV file.

With dict:

```
from redbird.repos import CSVFileRepo
repo = CSVFileRepo(filename="my_repo.csv", fieldnames=['id', 'name', 'age'])
```

Warning: CSV files don't maintain the data types. All field values are considered `str` and empty values are considered `None`. It is adviced to use Pydantic model (see below) instead if the type matters.

With Pydantic model:

```
from pydantic import BaseModel
from redbird.repos import CSVFileRepo

class MyItem(BaseModel):
    id: str
    name: str
    age: int

repo = CSVFileRepo(filename="my_repo.csv", model=MyItem)
```

Usage

Now you may use the repository the same way as any other repository. Please see:

- *Reading the repository*
- *Creating an item to the repository*
- *Deleting an item from the repository*
- *Updating an item in the repository*

Class

class redbird.repos.**CSVFileRepo**(*, *model=<class 'dict'>*, *id_field=None*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=True*, *filename*, *fieldnames=None*, *kwds_csv={}*)

CSV file repository

This repository has a CSV (comma-separated values) file as a data store. Each item represents a row in the file.

Parameters

- **filename** (*path-like*) – The repository file
- **fieldnames** (*list of str*) – Names of the columns in the CSV file. If unspecified, model's fields are used instead
- **model** (*Type*) – Class of an item in the repository. Commonly dict or subclass of Pydantic BaseModel. By default dict
- **id_field** (*str, optional*) – Attribute or key that identifies each item in the repository.
- **field_access** (*{'attr', 'key'}, optional*) – How to access a field in an item. Either by attribute ('attr') or key ('item'). By default guessed from the model.
- **query** (*Type, optional*) – Query model of the repository.
- **errors_query** (*{'raise', 'warn', 'discard'}*) – Whether to raise an exception, warn or discard the item in case of validation error in converting data to the item model from the repository. By default raise
- **kwds_csv** (*dict*) – Keyword arguments used to create `csv.DictWriter` and `csv.DictReader`

Examples

```
repo = CSVFileRepo(filepath="path/to/repo", fieldnames=['id', 'name', 'age'])
```

insert(*item*)

Insert item to the repository

Parameters **item** (*instance of model*) – Item to insert to the repository

Examples

```
repo.insert(Item(id="a", color="red"))
```

filter_by(***kwargs*)

Filter the repository

Parameters ****kwargs** (*dict*) – Query which is used to conduct furher operation.

Examples

```
repo.filter_by(color="red")
```

update(*item*)

Update item in the repository

Parameters **item** (*instance of model*) – Item to update in the repository

Examples

```
repo.update(Item(id="a", color="red"))
```

delete(*item*)

Delete item from the repository

Parameters **item** (*instance of model*) – Item to delete from the repository

Examples

```
repo.delete(Item(id="a", color="red"))
```

4.2.3 JSON Directory Repository

JSON (JavaScript Object Notation) directory repository is a data store in which each item represents a JSON file in a directory. The stems of the files are the IDs specified in the `id_field` of the repository.

```
from redbird.repos import JSONDirectoryRepo
repo = JSONDirectoryRepo(path="path/to/repo", id_field='id')
```

With Pydantic model:

```
from pydantic import BaseModel
from redbird.repos import JSONDirectoryRepo

class MyItem(BaseModel):
    id: str
    name: str
    age: int

repo = JSONDirectoryRepo(path="path/to/repo", model=MyItem, id_field='id')
```

Note: The `id_field` must be specified as the name of the JSON file depends on the value of this field.

Warning: The order in which the items are stored depends on the operating system thus the order in which the items are returned when reading the repository is not guaranteed to be fixed.

Usage

Now you may use the repository the same way as any other repository. Please see:

- *Reading the repository*
- *Creating an item to the repository*
- *Deleting an item from the repository*
- *Updating an item in the repository*

Class

class redbird.repos.**JSONDirectoryRepo**(*, *model=<class 'dict'>*, *id_field*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=False*, *path*, *field_names=None*, *kwds_json_load={}*, *kwds_json_dump={}*)

JSON directory repository

This repository represents a directory which contains JSON files. Each item represents a file and the `id_field` is the stem of the file.

Parameters

- **path** (*Path-like*) – Path to the repository directory
- **model** (*Type*) – Class of an item in the repository. Commonly dict or subclass of Pydantic BaseModel. By default dict
- **id_field** (*str*) – Attribute or key that identifies each item in the repository.
- **field_access** (*{'attr', 'key'}, optional*) – How to access a field in an item. Either by attribute ('attr') or key ('item'). By default guessed from the model.
- **query** (*Type, optional*) – Query model of the repository.
- **errors_query** (*{'raise', 'warn', 'discard'}*) – Whether to raise an exception, warn or discard the item in case of validation error in converting data to the item model from the repository. By default raise
- **kwds_json_load** (*dict*) – Keyword arguments passed to `json.load`.
- **kwds_json_dump** (*dict*) – Keyword arguments passed to `json.dump`. Useful for prettifying the files

Examples

```
repo = JSONDirectoryRepo(path="path/to/repo", id_field="id")
```

insert(*item*)

Insert item to the repository

Parameters **item** (*instance of model*) – Item to insert to the repository

Examples

```
repo.insert(Item(id="a", color="red"))
```

filter_by(***kwargs*)

Filter the repository

Parameters ****kwargs** (*dict*) – Query which is used to conduct furher operation.

Examples

```
repo.filter_by(color="red")
```

update(*item*)

Update item in the repository

Parameters **item** (*instance of model*) – Item to update in the repository

Examples

```
repo.update(Item(id="a", color="red"))
```

delete(*item*)

Delete item from the repository

Parameters **item** (*instance of model*) – Item to delete from the repository

Examples

```
repo.delete(Item(id="a", color="red"))
```

4.2.4 SQL Repositories

There are two SQL related repositories in Red Bird:

- *redbird.repos.SQLRepo*: Built around object-relational mapping
- *redbird.repos.SQLExprRepo*: Built around SQL expressions

Both of them relies on functionalities provided by SQLAlchemy. In most cases *redbird.repos.SQLExprRepo* is right choice as often it is less error prone as it is simpler. *redbird.repos.SQLRepo* is useful if you need to leverage more from SQLAlchemy's ORM features.

They also differ in terms of how they do the connection:

- *redbird.repos.SQLRepo*: Relies on `sqlalchemy.orm.Session`
- *redbird.repos.SQLExprRepo*: Relies on `sqlalchemy.engine.Engine`

SQL Expressions Repository

SQLExprRepo is a repository that relies on SQLAlchemy's SQL expressions which are pieces of SQL code represented in Python.

It can be initiated using SQLAlchemy engine:

```
from sqlalchemy import create_engine
from redbird.repos import SQLExprRepo

engine = create_engine('sqlite://')
repo = SQLExprRepo(engine=engine, table="my_table")
```

Usage

Now you may use the repository the same way as any other repository. Please see:

- *Reading the repository*
- *Creating an item to the repository*
- *Deleting an item from the repository*
- *Updating an item in the repository*

Class

class redbird.repos.**SQLExprRepo**(**args*, *model=<class 'dict'>*, *id_field=None*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=False*, *table=None*, *engine=None*)

insert(*item*)

Insert item to the repository

Parameters **item** (*instance of model*) – Item to insert to the repository

Examples

```
repo.insert(Item(id="a", color="red"))
```

filter_by(***kwargs*)

Filter the repository

Parameters ****kwargs** (*dict*) – Query which is used to conduct furher operation.

Examples

```
repo.filter_by(color="red")
```

update(*item*)

Update item in the repository

Parameters **item** (*instance of model*) – Item to update in the repository

Examples

```
repo.update(Item(id="a", color="red"))
```

delete(*item*)

Delete item from the repository

Parameters **item** (*instance of model*) – Item to delete from the repository

Examples

```
repo.delete(Item(id="a", color="red"))
```

SQL ORM Repository

SQLRepo is an SQL repository that relies on SQLAlchemy's object-relational mapping (ORM). The difference to `SQLExprRepo` is that `SQLRepo` is more object centric and it may be useful if you already use ORM in some places. It is also more complex and in some cases more error prone.

SQLRepo can be initiated numerous ways. You may initiate it with session, engine or SQLAlchemy connection string and you may optionally supply ORM model.

```
from sqlalchemy import create_engine
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
repo = SQLRepo(engine=engine, table="my_table")
```

You may also supply a session:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import Session
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
session = Session(engine)
repo = SQLRepo(session=session, table="my_table")
```

Using ORM model:

```
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Car(Base):
    __tablename__ = 'my_table'
    color = Column(String, primary_key=True)
    car_type = Column(String)
    milage = Column(Integer)

from sqlalchemy import create_engine
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
repo = SQLRepo(model_orm=Car, engine=engine)
```

Using ORM model and reflect Pydantic Model:

```
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Car(Base):
    __tablename__ = 'my_table'
    color = Column(String, primary_key=True)
    car_type = Column(String)
    milage = Column(Integer)

from sqlalchemy import create_engine
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
repo = SQLRepo(model_orm=Car, reflect_model=True, engine=engine)
```

Using ORM model and Pydantic Model:

```
from pydantic import BaseModel
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class CarORM(Base):
    __tablename__ = 'my_table'
```

(continues on next page)

(continued from previous page)

```
    color = Column(String, primary_key=True)
    car_type = Column(String)
    milage = Column(Integer)

class Car(BaseModel):
    id: str
    name: str
    age: int

from sqlalchemy import create_engine
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
repo = SQLRepo(model=Car, model_orm=CarORM, engine=engine)
```

Usage

Now you may use the repository the same way as any other repository. Please see:

- *Reading the repository*
- *Creating an item to the repository*
- *Deleting an item from the repository*
- *Updating an item in the repository*

Class

class redbird.repos.**SQLRepo**(**args*, *reflect_model=False*, *conn_string=None*, *engine=None*, *session=None*, *if_missing='raise'*, *model=<class 'dict'>*, *id_field=None*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=True*, *model_orm=None*, *table=None*, *autocommit=True*)

SQL Repository

Parameters

- **conn_string** (*str, optional*) – Connection string to the database. Pass either conn_string, engine or session if model_orm is not defined.
- **engine** (*sqlalchemy.engine.Engine, optional*) – SQLAlchemy engine to connect the database. Pass either conn_string, engine or session if model_orm is not defined.
- **model** (*Type*) – Class of an item in the repository. Commonly dict or subclass of Pydantic BaseModel. By default dict
- **id_field** (*str, optional*) – Attribute or key that identifies each item in the repository.
- **field_access** (*{'attr', 'key'}, optional*) – How to access a field in an item. Either by attribute ('attr') or key ('item'). By default guessed from the model.
- **query** (*Type, optional*) – Query model of the repository.
- **errors_query** (*{'raise', 'warn', 'discard'}*) – Whether to raise an exception, warn or discard the item in case of validation error in converting data to the item model from the repository. By default raise

- **model_orm** (*Type of Base, optional*) – Subclass of SQL Alchemy representation of the item. This is the class that is operated behind the scenes.
- **table** (*str, optional*) – Table name where the items lies. Should only be given if no model_orm specified.
- **session** (*sqlalchemy.orm.Session*) – Connection session to the database. Pass either conn_string, engine or session if model_orm is not defined.

Examples

```
from sqlalchemy import create_engine
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
repo = SQLRepo(engine=engine, table="my_table")
```

You may also supply a session:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import Session
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
session = Session(engine)
repo = SQLRepo(session=session, table="my_table")
```

Using ORM model:

```
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Car(Base):
    __tablename__ = 'my_table'
    color = Column(String, primary_key=True)
    car_type = Column(String)
    milage = Column(Integer)

from sqlalchemy import create_engine
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
repo = SQLRepo(model_orm=Car, engine=engine)
```

Using ORM model and reflect Pydantic Model:

```
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Car(Base):
    __tablename__ = 'my_table'
    color = Column(String, primary_key=True)
```

(continues on next page)

(continued from previous page)

```
    car_type = Column(String)
    milage = Column(Integer)

from sqlalchemy import create_engine
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
repo = SQLRepo(model_orm=Car, reflect_model=True, engine=engine)
```

Using ORM model and Pydantic Model:

```
from pydantic import BaseModel
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class CarORM(Base):
    __tablename__ = 'my_table'
    color = Column(String, primary_key=True)
    car_type = Column(String)
    milage = Column(Integer)

class Car(BaseModel):
    id: str
    name: str
    age: int

from sqlalchemy import create_engine
from redbird.repos import SQLRepo

engine = create_engine('sqlite://')
repo = SQLRepo(model=Car, model_orm=CarORM, engine=engine)
```

insert(*item*)

Insert item to the repository

Parameters **item** (*instance of model*) – Item to insert to the repository

Examples

```
repo.insert(Item(id="a", color="red"))
```

filter_by(***kwargs*)

Filter the repository

Parameters ****kwargs** (*dict*) – Query which is used to conduct furher operation.

Examples

```
repo.filter_by(color="red")
```

update(*item*)

Update item in the repository

Parameters **item** (*instance of model*) – Item to update in the repository

Examples

```
repo.update(Item(id="a", color="red"))
```

delete(*item*)

Delete item from the repository

Parameters **item** (*instance of model*) – Item to delete from the repository

Examples

```
repo.delete(Item(id="a", color="red"))
```

4.2.5 Mongo Repository

MongoRepo is a repository for MongoDB data stores. MongoDB is useful if you have unstructured data or you wish to store data in JSON format.

```
from redbird.repos import MongoRepo
repo = MongoRepo(uri="mongodb://USERNAME:PASSWORD@localhost:27017", database="my_db",␣
↪collection="my_items")
```

Alternatively, you may pass the client:

```
from redbird.repos import MongoRepo
from pymongo import MongoClient
repo = MongoRepo(client=MongoClient("mongodb://USERNAME:PASSWORD@localhost:27017"),␣
↪database="my_db", collection="my_items")
```

With Pydantic model:

```
from redbird.repos import MongoRepo

class MyItem(BaseModel):
    id: str
    name: str
    age: int

repo = MongoRepo(
    uri="mongodb://USERNAME:PASSWORD@localhost:27017",
    database="my_db",
    collection="my_items",
```

(continues on next page)

(continued from previous page)

```
    model=MyItem
)
```

Usage

Now you may use the repository the same way as any other repository. Please see:

- *Reading the repository*
- *Creating an item to the repository*
- *Deleting an item from the repository*
- *Updating an item in the repository*

Class

class redbird.repos.**MongoRepo**(**args*, *uri=None*, *client=None*, *session=None*, *model=<class 'dict'>*, *id_field=None*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=True*, *database=None*, *collection=None*, *default_id_field='_id'*)

MongoDB Repository

Parameters

- **uri** (*str, optional*) – Connection string to the database. Pass uri, client or session.
- **client** (*mongodb.MongoClient, optional*) – MongoDB client for the connection. Pass uri, client or session.
- **model** (*Type*) – Class of an item in the repository. Commonly dict or subclass of Pydantic BaseModel. By default dict
- **id_field** (*str, optional*) – Attribute or key that identifies each item in the repository.
- **field_access** (*{'attr', 'key'}, optional*) – How to access a field in an item. Either by attribute ('attr') or key ('item'). By default guessed from the model.
- **query** (*Type, optional*) – Query model of the repository.
- **errors_query** (*{'raise', 'warn', 'discard'}*) – Whether to raise an exception, warn or discard the item in case of validation error in converting data to the item model from the repository. By default raise
- **session** (*Session, Any*) – A MongoDB session object that should have at least `client` attribute. Pass uri, client or session.

Examples

```
repo = MongoRepo(uri="mongodb://localhost:27017/mydb?authSource=admin", collection=
↪"mycol")
```

```
repo = MongoRepo(uri="mongodb://localhost:27017", database="mydb", collection="mycol
↪")
```

```
from pymongo import MongoClient
repo = MongoRepo(client=MongoClient("mongodb://localhost:27017"))
```

insert(*item*)

Insert item to the repository

Parameters **item** (*instance of model*) – Item to insert to the repository

Examples

```
repo.insert(Item(id="a", color="red"))
```

filter_by(***kwargs*)

Filter the repository

Parameters ****kwargs** (*dict*) – Query which is used to conduct furher operation.

Examples

```
repo.filter_by(color="red")
```

update(*item*)

Update item in the repository

Parameters **item** (*instance of model*) – Item to update in the repository

Examples

```
repo.update(Item(id="a", color="red"))
```

delete(*item*)

Delete item from the repository

Parameters **item** (*instance of model*) – Item to delete from the repository

Examples

```
repo.delete(Item(id="a", color="red"))
```

4.2.6 Creating Custom Repository

Creating custom repositories can be done by simply sublcassing `TemplateRepo`. `TemplateRepo` is a Pydantic model that is integrated with Red Bird repository system.

In order to successfully create a custom repository, the following methods must be implemented:

- `insert(item)`: This method creates an item to the repository
- `query_data(query)`: This method gets items from the repository accoding to `query`
- `query_update(query, values)`: This method updates items in the repository matching to `query` with values defined in `values`
- `query_delete(query)`: This method deletes items from the repository matching the given `query`

In addition, the following methods could also be implemented for better performance:

- `query_read_first(query)`: This method queries the first record from the data store from items matching the `query`
- `query_read_limit(query, n)`: This method queries the first `n` items from the data store from items matching the `query`
- `query_read_last(query)`: This method queries the last item from the data store from items matching the `query`
- `query_replace(query, values)`: This method replaces the found item(s) from the data store from items matching the `query` using the values given in `values`
- `query_count(query)`: This method counts the values from the data store from items matching the `query`

The following methods could also be implemented for handling of the internals:

- `item_to_data(item)`: This method transforms an instance of `repo.model` to data undestood by the underlying data store
- `data_to_item(data)`: This method transforms raw data from the data store to an instance of `repo.model`
- `format_query(query)`: This method formats the `query` to a form suitable for the data store

Example

To demonstrate the subclassing, first we create a simple example that only works with `dict` as a model and without supporting any complicated querying options:

```
from redbird.templates import TemplateRepo

class MyRepo(TemplateRepo):

    store = [] # List that acts as our data store

    def insert(self, item):
        "Insert an item to the data store"
```

(continues on next page)

(continued from previous page)

```
        self.store.append(item)

    def query_data(self, query):
        "Read data from the data store"
        for item in self.store:
            include_item = all(
                query_value == item[key]
                for key, query_value in query.items()
            )
            if include_item:
                yield item

    def query_update(self, query, values):
        "Update items in the data store"
        for item in self.store:
            update_item = all(
                query_value == item[key]
                for key, query_value in query.items()
            )
            if update_item:
                for key, updated_value in values.items():
                    item[key] = updated_value

    def query_delete(self, query):
        "Delete items from the data store"
        for i, item in enumerate(self.store.copy()):
            delete_item = all(
                query_value == item[key]
                for key, query_value in query.items()
            )
            if delete_item:
                del self.store[i]
```

Then we may use this repository as:

```
>>> repo = MyRepo()

>>> # Creating some items
>>> repo.add({'name': 'Jack', 'age': 30})
>>> repo.add({'name': 'John', 'age': 30})
>>> repo.add({'name': 'James', 'age': 40})

>>> # Getting all items
>>> repo.filter_by().all()
[
    {'name': 'Jack', 'age': 30},
    {'name': 'John', 'age': 30},
    {'name': 'James', 'age': 40}
]

>>> # Getting an item
>>> repo.filter_by(age=30).first()
```

(continues on next page)

(continued from previous page)

```
{'name': 'Jack', 'age': 30}

>>> # Getting some items depending on the query
>>> repo.filter_by(age=30).all()
[
    {'name': 'Jack', 'age': 30},
    {'name': 'John', 'age': 30}
]

>>> # Updating some items
>>> repo.filter_by(name='James').update(age=41)

>>> # Deleting some items
>>> repo.filter_by(age=30).delete()
```

Template Class

class redbird.templates.**TemplateRepo**(*, *model=<class 'dict'>*, *id_field=None*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=False*)

Template repository for easy subclassing

Parameters

- **model** (*Type*) – Class of an item in the repository. Commonly dict or subclass of Pydantic BaseModel. By default dict
- **id_field** (*str, optional*) – Attribute or key that identifies each item in the repository.
- **field_access** (*{'attr', 'key'}, optional*) – How to access a field in an item. Either by attribute ('attr') or key ('item'). By default guessed from the model.
- **query** (*Type, optional*) – Query model of the repository.
- **errors_query** (*{'raise', 'warn', 'discard'}*) – Whether to raise an exception, warn or discard the item inn case of validation error in converting data to the item model from the repository. By default raise

Examples

```
class MyRepo(TemplateRepo):

    def insert(self, item):
        # Insert item to the data store
        ...

    def query_read(self, query: dict):
        # Get data from repository
        for item in ...:
            yield item

    def query_update(self, query: dict, values: dict):
```

(continues on next page)

(continued from previous page)

```
        # Update items with values matcing the query
        ...

    def query_delete(self, query: dict):
        # Delete items matcing the query
        ...

    def item_to_data(self, item):
        # Convert item to type that is understandable
        # by the repository's data store
        ...
        return data
```

cls_result

alias of *redbird.templates.TemplateResult*

query_data(*query*)

Query (read) the data store and return raw data

Override this or *query_items()* method.

Parameters **query** (*dict*) – Repository query, by default dict.

Returns Iterable of raw data that is converted to an item using *data_to_item()*

Return type iterable (any)

Examples

Used in following cases:

```
repo.filter_by(color="red").all()
```

query_items(*query*)

Query (read) the data store and return items

Override this or *query_data()* method.

Parameters **query** (*dict*) – Repository query, by default dict.

Returns Items that are instances of the class in the `model` attibute. Typically dicts or instances of subclassed Pydantic BaseModel

Return type iterable (`model`)

Examples

Used in following cases:

```
repo.filter_by(color="red").all()
```

abstract query_update(*query*, *values*)

Update the result of the query with given values

Override this method.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in following cases:

```
repo.filter_by(color="red").update(color="blue")
```

abstract query_delete(*query*)

Delete the result of the query

Override this method.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in following cases:

```
repo.filter_by(color="red").delete()
```

query_read_first(*query*)

Query the first item

You may override this method. By default, the first item returned by `TemplateRepo.query_read` is returned.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in the following case:

```
repo.filter_by(color="red").first()
```

query_read_limit(*query*, *n*)

Query the first n items

You may override this method. By default, the N first items returned by `TemplateRepo.query_read` are returned.

Parameters

- **query** (*any*) – Repository query, by default dict.
- **n** (*int*) – Number of items to return

Examples

Used in the following case:

```
repo.filter_by(color="red").limit(3)
```

query_read_last(*query*)

Query the last item

You may override this method. By default, the last item returned by `TemplateRepo.query_read` is returned.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in the following case:

```
repo.filter_by(color="red").last()
```

query_replace(*query*, *values*)

Replace the items with given values using given query

You may override this method. By default, the result of the query is deleted and an item from the values is generated.

Parameters

- **query** (*any*) – Repository query, by default dict.
- **values** (*dict*) – Values to replace the items' existing values with

Examples

Used in the following case:

```
repo.filter_by(color="red").replace(color="blue")
```

query_count(*query*)

Count the items returned by the query

You may override this method. By default, the items returned by `TemplateRepo.query_read` are counted.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in the following case:

```
repo.filter_by(color="red").count()
```

format_query(*query*)

Format the query to a format suitable by the repository

You may override this method. By default, the query is as dictionary.

Parameters **query** (*dict*) – Query to reformat

Examples

Used in the following case:

```
repo.filter_by(color="red")
```

4.2.7 Common Configurations

All repositories support some common configurations (unless specified otherwise). The item type can be either dict or Pydantic model and you may also specify which field is used as the identification field.

Specifying the Item Model

The item model can be either dict or Pydantic model. If you wish to have dynamic items in the sense that the items may have arbitrary number of fields with arbitrary type you may just use dict as the model:

```
repo = MemoryRepo(model=dict, ...)
```

However, in most cases it is recommended to specify the item model as a Pydantic model. By doing so, the structure of each item is better documented and you may leverage the extensive data validation Pydantic offers.

```
from pydantic import BaseModel

class Car(BaseModel):
    registration_number: str
    color: str
    value: float

from redbird.repos import MemoryRepo
repo = MemoryRepo(model=Car, ...)
```

Specifying the ID Field

Most repositories also support ID fields. ID field is a field/key/attribute of the items that is unique for all and is

```
repo = MemoryRepo(id_field="registration_number", ...)
```

Alternatively, the `id_field` could be set using `__id_field__` magic attribute:

```
from pydantic import BaseModel

class Car(BaseModel):
    __id_field__ = "registration_number"
    registration_number: str
    color: str
    value: float

repo = MemoryRepo(model=Car)
```

Specifying ID field is necessary for some features to work such as:

- getitem: `repo['123-456-789']`

- delitem: `del repo['123-456-789']`
- get_by: `repo.get_by('123-456-789')`

4.3 CRUD: Create, Read, Update, Delete

The idea of Red Bird is to have unified syntax regardless of the data store. This section contains repository methods that work indentically regardless of the repository. However, there may be some exceptions in special repositories.

In order to use the CRUD operations, remember to set the repository:

```
repo = Repo(...)
```

And the item:

```
from pydantic import BaseModel

class Person(BaseModel):
    id: str
    name: str
    nationality: str
```

4.3.1 Create

To create an item/record to the repository:

```
repo.add(Person(id="11-11-11", name="Jack", nationality='British'))
```

4.3.2 Read

You may get all the items from the repository by simply iterating over it:

```
list(repo)
```

You may also get items that contain given attribute values:

```
repo.filter_by(nationality="Finnish").all()
```

If you are only interested in the first found, you may use `first`:

```
repo.filter_by(nationality="Finnish").first()
```

If you are only interested in the last found, you may use `last`:

```
repo.filter_by(nationality="Finnish").last()
```

You may also fetch the first n found items:

```
repo.filter_by(nationality="Finnish").limit(2)
```

If you have `id_field` specified in the repository, you may also get an item using the ID by using any of the following:

```
repo["11-22-33"]
repo.get_by("11-22-33").first()
```

4.3.3 Update

In order to update an item in a repository, use `update` and pass the updated item:

```
person = repo["11-11-11"]
person.age += 1
repo.update(person)
```

You may also update several at a time:

```
repo.filter_by(nationality="English").update(age=30)
```

You may also update an item by the ID field using `get_by`:

```
repo.get_by("11-22-33").update(age=30)
```

4.3.4 Delete

To delete an item, use `del`:

```
del repo["11-11-11"]
```

or you may also use `delete` method:

```
person = repo["22-22-22"]
repo.delete(person)
```

You may also delete multiple items:

```
repo.filter_by(nationality="English").delete()
```

You may also delete an item by the ID field using `get_by`:

```
repo.get_by("11-22-33").delete()
```

4.4 Operations

Sometimes more sophisticated filtering operations are needed such as getting items that are greater than, less than etc. by specific field. For such purpose, Red Bird provide operations:

- `greater_than`
- `less_than`
- `not_equal`
- `between`
- `in_`

4.4.1 Examples

```
from redbird.oper import greater_than, less_than, not_equal

repo.filter_by(age=greater_than(30))

repo.filter_by(age=less_than(30))

repo.filter_by(age=not_equal(30))
```

```
from redbird.oper import between, in_

repo.filter_by(age=between(20, 40))

repo.filter_by(age=in_([20, 40]))
```

4.5 SQL Tools

Red Bird's SQL tools are small wrappers to SQLAlchemy's features to make basic database operations intuitive. Even though repository patterns are the main focus of Red Bird, these tools are fully supported.

`sqlalchemy.Table` is lower level than needed in simple create, read, update and delete operations (know as CRUD). The user is responsible of properly reflecting the table and executing SQL operations requires engine interaction from the user. Red Bird's *redbird.sql.Table* wraps the table with the engine so that they always point to a specific table in a specific SQL database. It also has methods to execute SQL operations directly to the table.

Here is an example:

```
from sqlalchemy import create_engine
from redbird.sql import Table

table = Table(name="mytable", bind=create_engine("sqlite://"))

# Create the table as it does not yet exists
table.create({"player_name": str, "score": int})

# Insert rows to the table
table.insert([
    {"player_name": "Player 1", "score": 100},
    {"player_name": "Player 2", "score": 200}
])

# Read the rows
table.select({"player_name": "Player 1"})

# Update a row
table.update(
    where={"player_name": "Player 1"},
    values={"score": 150}
)

# Delete a row
```

(continues on next page)

(continued from previous page)

```
table.delete({"player_name": "Player 2"})

# Drop (delete) the table
table.drop()
```

4.5.1 Table

redbird.sql.Table is further abstraction from `sqlalchemy.Table`. It combines the connection with the table itself thus it is always pointing to an actual SQL table (which may not yet exist). It also makes certain operations more intuitive and let you leverage Python's native data types more than SQLAlchemy does.

Class

class redbird.sql.**Table**(*name*, *bind*)

SQL Table

Similar to `sqlalchemy.Table` except this class always points to a specific SQL table in a specific database (which may not yet exist) and provides further abstraction.

name

Name of the table.

Type str

bind

SQLAlchemy engine or connection.

Type sqlalchemy.engine.Connectable

Examples

```
from sqlalchemy import create_engine
from redbird.sql import Table

table = Table("mytable", bind=create_engine("sqlite://"))
```

select(*qry=None*, *columns=None*, *parameters=None*)

Read the database table using a query.

Parameters

- **qry** (*str, dict, sqlalchemy.sql.ClauseElement, optional*) – Query to filter the data. The argument can take various forms:
  - `str`: Query is considered to be raw SQL
  - `dict`: Query is considered to be column-filter pairs. The pairs are combined using AND operator. If the filter is Operation, it is turned to corresponing SQLAlchemy expression. Else, the filter is considered to be an equal operator.
  - sqlalchemy expression: The query is considered to be the *where* clause of the select query.

  If not given, all rows are returned.

- **columns** (*list of string, optional*) – List of columns to return. By default returns all columns.
- **parameters** (*dict, optional*) – Parameters for the query. If the query is as raw SQL, they should be set in the query with `:myparam`, ie. `select * from table where col = :myparam`.

Returns Found rows as dicts.

Return type List of dicts

Examples

Select all rows:

```
table.select()
```

Select using raw SQL:

```
table.select("SELECT * FROM mytable")
```

Select using dictionary:

```
table.select({"column_1": "a value", "column_2": 10})
```

Note: The above is same as:

```
SELECT *
FROM mytable
WHERE column_1 = 'a value' AND column_2 = 10
```

Select using SQLAlchemy expressions:

```
from sqlalchemy import Column
table.select((Column("column_1") == "a value") & (Column("column_2") == 10))
```

Note: The above is same as:

```
SELECT *
FROM mytable
WHERE column_1 = 'a value' AND column_2 = 10
```

Select and return specific column(s):

```
table.select(columns=["column_1", "column_2"])
```

Note: The above is same as:

```
SELECT column_1, column_2
FROM mytable
```

Select using raw strings and SQL parameters:

```
table.select(
    "SELECT * FROM mytable WHERE column_1 = :myparam",
    parameters={"myparam": "a value"}
)
```

insert(*data*)

Insert data to the database.

Parameters **data** (*dict, list of dicts*) – Data to be inserted.

Examples

Insert a single row:

```
table.insert({"column_1": "a", "column_2": 1})
```

Insert multiple rows:

```
table.insert([
    {"column_1": "a", "column_2": 1},
    {"column_1": "b", "column_2": 2},
])
```

delete(*where*)

Delete row(s) from the table.

Parameters **where** (*dict, sqlalchemy expression*) – Where clause to delete data.

Returns Count of rows deleted.

Return type int

Examples

Delete using dictionary:

```
table.delete({"column_1": "a", "column_2": 1})
```

Note: The above is same as:

```
DELETE FROM mytable
WHERE column_1 = 'a' AND column_2 = 1
```

Delete using SQL expressions:

```
from sqlalchemy import Column
table.delete((Column("column_1") == "a") & (Column("column_2") == 1))
```

Note: The above is same as:

```
DELETE FROM mytable
WHERE column_1 = 'a' AND column_2 = 1
```

Delete all:

```
table.delete({})
```

update(*where*, *values*)

Update row(s) in the table.

Parameters

- **where** (*dict, sqlalchemy expression*) – Where clause to update rows.
- **values** (*dict*) – Column-value pairs to update the rows matching the where clause.

Returns Count of rows updated.

Return type int

Examples

Update using dicts:

```
table.update({"column_1": "a", "column_2": 1}, {"column_3": "new value"})
```

Note: The above is same as:

```
UPDATE mytable
SET column_3='new value'
WHERE column_1 = 'a' and column_2 = 1
```

Update using expressions:

```
from sqlalchemy import Column
table.update((Column("column_1") == "a") & (Column("column_2") == 1), {"column_3
↪": "new value"})
```

Note: The above is same as:

```
UPDATE mytable
SET column_3='new value'
WHERE column_1 = 'a' and column_2 = 1
```

Update all:

```
table.update({}, {"column_3": "new value"})
```

count(*where=None*)

Count the number of rows.

Parameters **where** (*dict, sqlalchemy expression, optional*) – Where clause to be satisfied for counting the rows.

Returns Count of rows (satisfying the where clause).

Return type int

Examples

Count based on dict:

```
table.count({"column_1": "a", "column_2": 1})
```

drop()

Drop the table.

exists()

Check if the table exists.

create(*columns*, *exist_ok=False*)

Create the table

Parameters

- **columns** (*dict, list of sqlalchemy.Column, dict or string*) – Columns to be created.
- **exist_ok** (*bool*) – If false (default), an exception is raised.

Examples

There are various ways to call this method:

- Using list of columns (all of the)

Create a table with columns column_1, column_2 and column_3 (all of them are textual columns):

```
table.create(["column_1", "column_2", "column_3"])
```

Create a table with columns column_1, column_2 and column_3 with varying data types:

```
import datetime
table.create({"column_1": str, "column_2": int, "column_3": datetime.datetime})
```

Create a table with columns column_1, column_2 and column_3 using SQLAlchemy columns:

```
from sqlalchemy import Column, String, Integer, DateTime
table.create([
    Column("column_1", type_=String()),
    Column("column_2", type_=Integer()),
    Column("column_3", type_=DateTime())
])
```

Create a table with columns column_1, column_2 and column_3 using list of dicts:

```
from sqlalchemy import DateTime
table.create([
    {"name": "column_1", "type_": str},
    {"name": "column_2", "type_": int},
    {"name": "column_3", "type_": DateTime()},
])
```

execute(**args*, ***kwargs*)

Execute SQL statement or raw SQL.

Parameters

- ***args** (*tuple*) – Passed directly to sqlalchemy.Connection.execute.
- ****args** (*dict*) – Passed directly to sqlalchemy.Connection.execute.

open_transaction()

Open a transaction.

Examples

```
from redbird.sql import Table
from sqlalchemy import create_engine

table = Table("mytable", bind=create_engine(...))

# Open the transaction
transaction = table.open_transaction()

# Perform operations
transaction.insert({"col_1": "a", "col_2": "b"})
transaction.delete({"col_2": "c"})

# Commit or rollback the changes
if successful:
    transaction.commit()
else:
    transaction.rollback()
```

transaction()

Open a transaction context.

If an error is raised inside the with block, the changes are rollbacked. Else they are commited.

Examples

```
from redbird.sql import Table
from sqlalchemy import create_engine

table = Table("mytable", bind=create_engine(...))

with table.transaction() as trans:

    # Perform operations
    trans.insert({"col_1": "a", "col_2": "b"})
    trans.delete({"col_2": "c"})
```

4.5.2 SQL Functions

Red Bird's SQL functions are functional tools to perform simple SQL operations. They provide functional alternative to *redbird.sql.Table*. They simply create an instance of the class and call its methods.

Basic Operations

redbird.sql.**select**(**args*, *bind*, *table=None*, ***kwargs*)

Read rows from a table in a SQL database.

Parameters

- **engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy engine for the connection
- **table** (*str*) – Name of the table to use.
- ***args** – Passed to *redbird.sql.Table.select()*
- ****kwargs** – Passed to *redbird.sql.Table.select()*

redbird.sql.**insert**(**args*, *bind*, *table=None*, ***kwargs*)

Insert row(s) to a table in a SQL database.

Parameters

- **bind** (*sqlalchemy.engine.Engine*) – SQLAlchemy engine for the connection
- **table** (*str*) – Name of the table to use.
- ***args** – Passed to *redbird.sql.Table.insert()*
- ****kwargs** – Passed to *redbird.sql.Table.insert()*

redbird.sql.**update**(**args*, *bind*, *table=None*, ***kwargs*)

Update row(s) to a table in a SQL database.

Parameters

- **bind** (*sqlalchemy.engine.Engine*) – SQLAlchemy engine for the connection
- **table** (*str*) – Name of the table to use.
- ***args** – Passed to *redbird.sql.Table.update()*
- ****kwargs** – Passed to *redbird.sql.Table.update()*

redbird.sql.**delete**(**args*, *bind*, *table=None*, ***kwargs*)

Delete row(s) in a table in a SQL database.

Parameters

- **bind** (*sqlalchemy.engine.Engine*) – SQLAlchemy engine for the connection
- **table** (*str*) – Name of the table to use.
- ***args** – Passed to *redbird.sql.Table.delete()*
- ****kwargs** – Passed to *redbird.sql.Table.delete()*

Additional Functions

redbird.sql.**execute**(**args*, *bind*, ***kwargs*)

Execute raw SQL or a SQL expression in a SQL database.

Parameters

- **bind** (*sqlalchemy.engine.Engine*) – SQLAlchemy engine for the connection
- **table** (*str*) – Name of the table to use.
- ***args** – Passed to *redbird.sql.Table.execute()*
- ****kwargs** – Passed to *redbird.sql.Table.execute()*

redbird.sql.**create_table**(**args*, *bind*, *table*, ***kwargs*)

Create a table to a SQL database.

Parameters

- **bind** (*sqlalchemy.engine.Engine*) – SQLAlchemy engine for the connection
- **table** (*str*) – Name of the table to be created.
- ***args** – Passed to *redbird.sql.Table.create()*
- ****kwargs** – Passed to *redbird.sql.Table.create()*

redbird.sql.**count**(**args*, *bind*, *table=None*, ***kwargs*)

Count the number of rows in a table in a SQL database.

Parameters

- **bind** (*sqlalchemy.engine.Engine*) – SQLAlchemy engine for the connection
- **table** (*str*) – Name of the table to use.
- ***args** – Passed to *redbird.sql.Table.count()*
- ****kwargs** – Passed to *redbird.sql.Table.count()*

4.5.3 Cookbook

Using Native Python

This is an example to operate on *redbird.sql.Table()* using native Python data types:

```
from redbird.sql import Table
from sqlalchemy import create_engine

engine = create_engine("sqlite://")
table = Table("shopping_list", bind=engine)

# Create a table
table.create({"product": str, "quantity": int})

# Insert one
table.insert({"product": "milk", "quantity": 2})

# Insert multiple
table.insert([
    {"product": "juice", "quantity": 1},
    {"product": "eggs", "quantity": 6},
])

# Select rows where product is milk and quantity is 1
table.select({"product": "milk", "quantity": 1})

# Update item(s)
table.update(where={"product": "milk"}, values={"quantity": 3})

# Delete item(s)
table.delete({"product": "juice"})
```

Using SQL Expressions

This is an example to operate on *redbird.sql.Table()* using SQLAlchemy's SQL expressions:

```
from redbird.sql import Table
from sqlalchemy import create_engine
import sqlalchemy

engine = create_engine("sqlite://")
table = Table("shopping_list", bind=engine)

# Create a table
table.create([
    sqlalchemy.Column("product", type_=sqlalchemy.String(), primary_key=True),
    sqlalchemy.Column("quantity", type_=sqlalchemy.Integer(), primary_key=False),
])

# Insert multiple
table.insert([
```

(continues on next page)

(continued from previous page)

```
    {"product": "milk", "quantity": 2},
    {"product": "juice", "quantity": 1},
    {"product": "eggs", "quantity": 6},
])

# Select rows where product is milk and quantity is 1
qry = (sqlalchemy.Column("product") == "milk") & (sqlalchemy.Column("quantity") == 1)
table.select(qry)

# Update item(s)
table.update(where=sqlalchemy.Column("product") == "milk", values={"quantity": 3})

# Delete item(s)
table.delete(sqlalchemy.Column("product") == "juice")
```

Operating on an Item

First we create an example table with some example data:

```
from sqlalchemy import create_engine
from redbird.sql import Table

table = Table(name="mytable", bind=create_engine("sqlite://"))

# Create a table that has index/indices
table.create([
    {"name": "index_1", "type_": str, "primary_key": True},
    {"name": "index_2", "type_": int, "primary_key": True},
    {"name": "firstname", "type_": str, "primary_key": False},
])

# Create data
table.insert([
    {"index_1": "a", "index_2": 1, "firstname": "Jack"},
    {"index_1": "a", "index_2": 2, "firstname": "John"},
    {"index_1": "b", "index_2": 1, "firstname": "James"},
    {"index_1": "b", "index_2": 2, "firstname": "Johnny"},
    {"index_1": "c", "index_2": 1, "firstname": "Jimmy"},
    {"index_1": "c", "index_2": 2, "firstname": "Jim"},
])
```

Selecting a specific item:

```
table[("a", 1)]
```

Note: This is same as:

```
SELECT *
FROM mytable
WHERE index_1 = 'a' AND index_2 = 1
```

You can also select range of items:

- `table["a"]`: Get all where `index_1` is `"a"`
- `table["a":"b"]`: Get all where `index_1` from `"a"` to `"b"`
- `table[["a", "c"]]`: Get all where `index_1` is `"a"` or `"c"`

Similarly, you can also delete item(s):

```
del table[("a", 1)]
```

Note: This is same as:

```
DELETE FROM mytable
WHERE index_1 = 'a' AND index_2 = 1
```

4.5.4 Transaction

There are three ways to do SQL transactions with *redbird.sql.Table*

- Using *redbird.sql.Table.open_transaction()*
- Using *redbird.sql.Table.transaction()*
- Opening it manually and create the table

The first two are covered in Table's documentation but in some cases you might need to open the transaction manually.

For example, if the transaction operates on multiple tables:

```
from sqlalchemy import create_engine
from redbird.sql import Table

engine = create_engine("sqlite://")

with engine.begin() as trans:

    car_table = Table("car", bind=trans)
    van_table = Table("van", bind=trans)

    # Do operations
    car_table.insert(...)
    van_table.delete(...)
```

4.6 Logging Handler

Red Bird also has a logging handler which is useful if you need to store the log records in a database or you wish to read the records later programmatically.

```
import logging
from redbird.logging import RepoHandler
from redbird.repos import MemoryRepo

# Create a repo
log_repo = MemoryRepo()

# Create a handler
handler = RepoHandler(repo=log_repo)

# Set the handler to a logger
logger = logging.getLogger('mylogger')
logger.addHandler(handler)
```

Note: In this example we used a repository that logs the records to an in-memory list. Read more about supported repositories here: *Repositories*

Then to use it:

```
logger.debug("A debug message")
logger.info("An info message")
logger.warning("A warning message")
```

To read the log records:

```
log_repo.filter_by(levelname="INFO").all()
```

Note: Read more about log record attributes: logging.LogRecord.

`RepoHandler` adds one extra attribute, `formatted_message`, that represents the message after it has been processed by the handler's formatter.

4.6.1 With a Record Model

If you need customization on the log record that is inserted to the database, you may create a Pydantic model and set that as the model of the repository. Here is an example to do so:

```
from pydantic import BaseModel, Field

class LogRecord(BaseModel):
    """A logging record

    See attributes: https://docs.python.org/3/library/logging.html#logrecord-attributes
    """
```

(continues on next page)

(continued from previous page)

```
    name: str
    msg: str
    levelname: str
    levelno: int
    pathname: str
    filename: str
    module: str
    exc_info: str
    exc_text: str
    stack_info: str
    lineno: int
    funcName: str
    created: float
    msecs: float
    relativeCreated: float
    thread: int
    threadName: str
    processName: str
    process: int
    message: str

    formatted_message: str = Field(description="Formatted message. This field is created␣
↪by RepoHandler.")
```

Then to set the model:

```
import logging
from redbird.logging import RepoHandler
from redbird.repos import MemoryRepo

log_repo = MemoryRepo(model=LogRecord)
handler = RepoHandler(repo=log_repo)

logger = logging.getLogger('mylogger')
logger.addHandler(handler)
```

4.7 Examples

This section contains a collection of small practical applications using Red Bird.

4.7.1 CRUD App with FastAPI

This is a fully functional minimal CRUD app with API:

```
from fastapi import FastAPI
from pydantic import BaseModel
from redbird.repos import MemoryRepo

app = FastAPI()
```

(continues on next page)

(continued from previous page)

```
class Item(BaseModel):
    id: int
    name: str

repo = MemoryRepo(model=Item, id_field="id")

@app.post("/items", description="Create an item")
def create_item(item: Item):
    repo.add(item)

@app.get("/items", description="Get all items")
def get_items():
    return repo.filter_by().all()

@app.get("/items/{item_id}", description="Get item by ID")
def get_item(item_id: int):
    return repo[item_id]

@app.patch("/items/{item_id}", description="Update an item")
def update_item(item_id: int, values: dict):
    repo[item_id] = values

@app.delete("/items/{item_id}", description="Delete an item")
def delete_item(item_id: int):
    del repo[item_id]
```

Read more about FastAPI: FastAPI docs

4.7.2 CRUD App with Command Line

This is a fully functional CRUD command-line app:

```
import argparse

from pydantic import BaseModel, Field
from redbird.repos import SQLRepo

class Item(BaseModel):
    id: int
    name: str = Field(description="Name of the item", default="No name")

repo = SQLRepo(conn_string="sqlite:///example.db", table="items", model=Item, id_field=
↪"id", if_missing="create")

def create_item(**kwargs):
    "Create an item"
    item = Item(**kwargs)
    repo.add(item)

def read_item(item_id=None):
```

(continues on next page)

(continued from previous page)

```
    "Read one item (or all)"
    if item_id is None:
        # Read all
        for item in repo:
            print(repr(item))
    else:
        # Read one
        item = repo[item_id]
        print(repr(item))

def update_item(item_id, **values):
    "Update one item"
    values = {key: value for key, value in values.items() if value is not None}
    repo[item_id] = values
    repo.filter_by(id=2).update(**values)

def delete_item(item_id):
    "Delete one item"
    del repo[item_id]

def add_model_to_parser(parser: argparse.ArgumentParser, model):
    "Add Pydantic model's attributes as arguments to the parser"
    fields = model.__fields__
    for name, field in fields.items():
        parser.add_argument(
            f"--{name}",
            dest=name,
            type=field.type_,
            default=field.default,
            help=field.field_info.description,
        )

def parse_args(args=None):
    "Parse CLI arguments"
    parser = argparse.ArgumentParser(prog='Simple CRUD app')
    subparsers = parser.add_subparsers(dest='action')

    # Create subparsers for a single item
    parser_create = subparsers.add_parser("create", help="Create an item")
    parser_read = subparsers.add_parser("read", help="Read an item")
    parser_update = subparsers.add_parser("update", help="Update an item")
    parser_delete = subparsers.add_parser("delete", help="Delete an item")

    # Create subparsers for multiple items
    parser_read_all = subparsers.add_parser("read_all", help="Read items")

    # Add arguments
    parser_read.add_argument("item_id", nargs="?", help="ID of the item")
    for sub_parser in (parser_update, parser_delete):
        sub_parser.add_argument("item_id", help="ID of the item")

    add_model_to_parser(parser_create, model=Item)
```

(continues on next page)

(continued from previous page)

```
    add_model_to_parser(parser_update, model=Item)

    return parser.parse_args(args)

def main(args=None):
    "A simple app to operate on a datastore"
    args = parse_args(args)
    args = vars(args)

    action = args.pop("action")
    func = {
        'create': create_item,
        'read': read_item,
        'update': update_item,
        'delete': delete_item,
    }[action]

    return func(**args)

if __name__ == "__main__":
    main()
```

Read more about argparse: argparse docs

4.8 References

This section consists of the API documentation of Red Bird. The classes, functions and methods are documented here. Please see *tutorials* for more thorough instruction of how to use the library.

4.8.1 Abstract Classes

Template

class redbird.templates.**TemplateRepo**(*, *model=<class 'dict'>*, *id_field=None*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=False*)

Template repository for easy subclassing

Parameters

- **model** (*Type*) – Class of an item in the repository. Commonly dict or subclass of Pydantic BaseModel. By default dict
- **id_field** (*str, optional*) – Attribute or key that identifies each item in the repository.
- **field_access** (*{'attr', 'key'}, optional*) – How to access a field in an item. Either by attribute ('attr') or key ('item'). By default guessed from the model.
- **query** (*Type, optional*) – Query model of the repository.
- **errors_query** (*{'raise', 'warn', 'discard'}*) – Whether to raise an exception, warn or discard the item inn case of validation error in converting data to the item model from the repository. By default raise

Examples

```
class MyRepo(TemplateRepo):

    def insert(self, item):
        # Insert item to the data store
        ...

    def query_read(self, query: dict):
        # Get data from repository
        for item in ...:
            yield item

    def query_update(self, query: dict, values: dict):
        # Update items with values matcing the query
        ...

    def query_delete(self, query: dict):
        # Delete items matcing the query
        ...

    def item_to_data(self, item):
        # Convert item to type that is understandable
        # by the repository's data store
        ...
        return data
```

cls_result

alias of *redbird.templates.TemplateResult*

query_data(*query*)

Query (read) the data store and return raw data

Override this or *query_items()* method.

Parameters **query** (*dict*) – Repository query, by default dict.

Returns Iterable of raw data that is converted to an item using *data_to_item()*

Return type iterable (any)

Examples

Used in following cases:

```
repo.filter_by(color="red").all()
```

query_items(*query*)

Query (read) the data store and return items

Override this or *query_data()* method.

Parameters **query** (*dict*) – Repository query, by default dict.

Returns Items that are instances of the class in the `model` attibute. Typically dicts or instances of subclassed Pydantic BaseModel

Return type iterable (`model`)

Examples

Used in following cases:

```
repo.filter_by(color="red").all()
```

abstract query_update(*query*, *values*)

Update the result of the query with given values

Override this method.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in following cases:

```
repo.filter_by(color="red").update(color="blue")
```

abstract query_delete(*query*)

Delete the result of the query

Override this method.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in following cases:

```
repo.filter_by(color="red").delete()
```

query_read_first(*query*)

Query the first item

You may override this method. By default, the first item returned by `TemplateRepo.query_read` is returned.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in the following case:

```
repo.filter_by(color="red").first()
```

query_read_limit(*query*, *n*)

Query the first n items

You may override this method. By default, the N first items returned by `TemplateRepo.query_read` are returned.

Parameters

- **query** (*any*) – Repository query, by default dict.
- **n** (*int*) – Number of items to return

Examples

Used in the following case:

```
repo.filter_by(color="red").limit(3)
```

query_read_last(*query*)

Query the last item

You may override this method. By default, the last item returned by `TemplateRepo.query_read` is returned.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in the following case:

```
repo.filter_by(color="red").last()
```

query_replace(*query*, *values*)

Replace the items with given values using given query

You may override this method. By default, the result of the query is deleted and an item from the values is generated.

Parameters

- **query** (*any*) – Repository query, by default dict.
- **values** (*dict*) – Values to replace the items' existing values with

Examples

Used in the following case:

```
repo.filter_by(color="red").replace(color="blue")
```

query_count(*query*)

Count the items returned by the query

You may override this method. By default, the items returned by `TemplateRepo.query_read` are counted.

Parameters **query** (*any*) – Repository query, by default dict.

Examples

Used in the following case:

```
repo.filter_by(color="red").count()
```

format_query(*query*)

Format the query to a format suitable by the repository

You may override this method. By default, the query is as dictionary.

Parameters **query** (*dict*) – Query to reformat

Examples

Used in the following case:

```
repo.filter_by(color="red")
```

class redbird.templates.**TemplateResult**(*query=None*, *repo=None*)

query()

Get actual result

query_data()

Get actual result

update(***kwargs*)

Update the resulted items

delete()

Delete the resulted items

replace(*_TemplateResult__values=None*, ***kwargs*)

Replace the existing item(s) with given

count()

Count the resulted items

first()

Return first item

limit(*n*)

Return n items

last()

Return last item

format_query(*query*)

Turn the query to a form that's understandable by the underlying database

Base Classes

class redbird.base.**BaseRepo**(*, *model=<class 'dict'>*, *id_field=None*, *query_model=<class 'redbird.base.BasicQuery'>*, *errors_query='raise'*, *field_access='infer'*, *ordered=False*)

Abstract Repository

Base class for the repository pattern.

Parameters

- **model** (*Type*) – Class of an item in the repository. Commonly dict or subclass of Pydantic BaseModel. By default dict
- **id_field** (*str, optional*) – Attribute or key that identifies each item in the repository.
- **field_access** (*{'attr', 'key'}, optional*) – How to access a field in an item. Either by attribute ('attr') or key ('item'). By default guessed from the model.
- **query** (*Type, optional*) – Query model of the repository.
- **errors_query** (*{'raise', 'warn', 'discard'}*) – Whether to raise an exception, warn or discard the item in case of validation error in converting data to the item model from the repository. By default raise

add(*item*, *if_exists='raise'*)

Add an item to the repository

abstract insert()

Insert item to the repository

Parameters **item** (*instance of model*) – Item to insert to the repository

Examples

```
repo.insert(Item(id="a", color="red"))
```

upsert(*item*)

Upsert item to the repository

Upsert is an insert if the item does not exist in the repository and update if it does.

Parameters **item** (*instance of model*) – Item to upsert to the repository

Examples

```
repo.upsert(Item(id="a", color="red"))
```

delete(*item*)

Delete item from the repository

Parameters **item** (*instance of model*) – Item to delete from the repository

Examples

```
repo.delete(Item(id="a", color="red"))
```

update(*item*)

Update item in the repository

Parameters **item** (*instance of model*) – Item to update in the repository

Examples

```
repo.update(Item(id="a", color="red"))
```

replace(*item*)

Update an item in the repository

get_by(*id*)

Get item based on ID but returns result for further operations

filter_by(***kwargs*)

Filter the repository

Parameters ****kwargs** (*dict*) – Query which is used to conduct furher operation.

Examples

```
repo.filter_by(color="red")
```

data_to_item(*data*)

Turn object from repo (row, doc, dict, etc.) to item

to_item(*obj*)

Turn an object to item

get_field_value(*item*, *key*)

Utility method to get key's value from an item

If item's fields are accessed via attribute, getattr is used. If fields are accessed via items, getitem is used.

set_field_value(*item*, *key*, *value*)

Utility method to set field's value in an item

If item's fields are accessed via attribute, setattr is used. If fields are accessed via items, setitem is used.

class redbird.base.**BaseResult**(*query=None*, *repo=None*)

Abstract filter result

Result classes add additional alchemy to Red Bird providing convenient ways to read, modify or delete data.

Subclass of BaseRepo should also have custom subclass of BaseResult as cls_result attribute.

first()

Return first item

last()

Return last item

all()
Return all items

limit(*n*)
Return n items

query()
Get actual result

abstract query_data()
Get actual result

abstract update(***kwargs*)
Update the resulted items

abstract delete()
Delete the resulted items

replace(*_BaseResult__values=None*, ***kwargs*)
Replace the existing item(s) with given

count()
Count the resulted items

format_query(*query*)
Turn the query to a form that's understandable by the underlying database

4.9 Version history

- `0.7.1`
  - SQL: Fix bug in `redbird.sql.Table.select` about closed resource.
- `0.7.0`
  - SQL: add new repo, *redbird.repos.SQLExprRepo*, which relies on SQLAlchemy' SQL expressions
  - SQL: add SQL tools (`redbird.sql`) to make SQL operations more intuitive
- `0.6.1`
  - SQL: fix model based `__id_field__` in reflection
  - SQL: fix date-like types in table creation with `if_missing='create'`
- `0.6.0`
  - Added alternative way to set repository's ID field: setting `__id_field__` in model
  - Packaging updated to use pyproject.toml
- `0.5.1`
  - Fixed `CSVFileRepo` error if read and the file does not exists
- `0.5.0`
  - Added `in` operator
- `0.4.0`
  - Added logging utility: `RepoHandler`

  - Fixed not committing deletes and updates in SQLRepo
  - Fixed ORM model reflection in SQLRepo
  - Continued documentation
- `0.3.0`
  - First official release
  - Stabilized the API
  - Updated documentation
  - Added several new repositories
- `0.2.4`
  - First release

CHAPTER

FIVE

INDICES AND TABLES

- genindex

INDEX

A

add() (*redbird.base.BaseRepo method*), 57
all() (*redbird.base.BaseResult method*), 58

B

BaseRepo (*class in redbird.base*), 57
BaseResult (*class in redbird.base*), 58
bind (*redbird.sql.Table attribute*), 37

C

cls_result (*redbird.templates.TemplateRepo attribute*), 53
count() (*in module redbird.sql*), 44
count() (*redbird.base.BaseResult method*), 59
count() (*redbird.sql.Table method*), 40
count() (*redbird.templates.TemplateResult method*), 56
create() (*redbird.sql.Table method*), 41
create_table() (*in module redbird.sql*), 44
CSVFileRepo (*class in redbird.repos*), 14

D

data_to_item() (*redbird.base.BaseRepo method*), 58
delete() (*in module redbird.sql*), 43
delete() (*redbird.base.BaseRepo method*), 57
delete() (*redbird.base.BaseResult method*), 59
delete() (*redbird.repos.CSVFileRepo method*), 15
delete() (*redbird.repos.JSONDirectoryRepo method*), 17
delete() (*redbird.repos.MemoryRepo method*), 13
delete() (*redbird.repos.MongoRepo method*), 26
delete() (*redbird.repos.SQLExprRepo method*), 19
delete() (*redbird.repos.SQLRepo method*), 24
delete() (*redbird.sql.Table method*), 39
delete() (*redbird.templates.TemplateResult method*), 56
drop() (*redbird.sql.Table method*), 41

E

execute() (*in module redbird.sql*), 44
execute() (*redbird.sql.Table method*), 42
exists() (*redbird.sql.Table method*), 41

F

filter_by() (*redbird.base.BaseRepo method*), 58
filter_by() (*redbird.repos.CSVFileRepo method*), 15
filter_by() (*redbird.repos.JSONDirectoryRepo method*), 17
filter_by() (*redbird.repos.MemoryRepo method*), 12
filter_by() (*redbird.repos.MongoRepo method*), 26
filter_by() (*redbird.repos.SQLExprRepo method*), 19
filter_by() (*redbird.repos.SQLRepo method*), 23
first() (*redbird.base.BaseResult method*), 58
first() (*redbird.templates.TemplateResult method*), 56
format_query() (*redbird.base.BaseResult method*), 59
format_query() (*redbird.templates.TemplateRepo method*), 56
format_query() (*redbird.templates.TemplateResult method*), 56

G

get_by() (*redbird.base.BaseRepo method*), 58
get_field_value() (*redbird.base.BaseRepo method*), 58

I

insert() (*in module redbird.sql*), 43
insert() (*redbird.base.BaseRepo method*), 57
insert() (*redbird.repos.CSVFileRepo method*), 14
insert() (*redbird.repos.JSONDirectoryRepo method*), 17
insert() (*redbird.repos.MemoryRepo method*), 12
insert() (*redbird.repos.MongoRepo method*), 26
insert() (*redbird.repos.SQLExprRepo method*), 18
insert() (*redbird.repos.SQLRepo method*), 23
insert() (*redbird.sql.Table method*), 39

J

JSONDirectoryRepo (*class in redbird.repos*), 16

L

last() (*redbird.base.BaseResult method*), 58
last() (*redbird.templates.TemplateResult method*), 56
limit() (*redbird.base.BaseResult method*), 59

limit() (*redbird.templates.TemplateResult method*), 56

M

MemoryRepo (*class in redbird.repos*), 12
MongoRepo (*class in redbird.repos*), 25

N

name (*redbird.sql.Table attribute*), 37

O

open_transaction() (*redbird.sql.Table method*), 42

Q

query() (*redbird.base.BaseResult method*), 59
query() (*redbird.templates.TemplateResult method*), 56
query_count() (*redbird.templates.TemplateRepo method*), 55
query_data() (*redbird.base.BaseResult method*), 59
query_data() (*redbird.templates.TemplateRepo method*), 53
query_data() (*redbird.templates.TemplateResult method*), 56
query_delete() (*redbird.templates.TemplateRepo method*), 54
query_items() (*redbird.templates.TemplateRepo method*), 53
query_read_first() (*redbird.templates.TemplateRepo method*), 54
query_read_last() (*redbird.templates.TemplateRepo method*), 55
query_read_limit() (*redbird.templates.TemplateRepo method*), 54
query_replace() (*redbird.templates.TemplateRepo method*), 55
query_update() (*redbird.templates.TemplateRepo method*), 54

R

replace() (*redbird.base.BaseRepo method*), 58
replace() (*redbird.base.BaseResult method*), 59
replace() (*redbird.templates.TemplateResult method*), 56

S

select() (*in module redbird.sql*), 43
select() (*redbird.sql.Table method*), 37
set_field_value() (*redbird.base.BaseRepo method*), 58
SQLExprRepo (*class in redbird.repos*), 18
SQLRepo (*class in redbird.repos*), 21

T

Table (*class in redbird.sql*), 37
TemplateRepo (*class in redbird.templates*), 52
TemplateResult (*class in redbird.templates*), 56
to_item() (*redbird.base.BaseRepo method*), 58
transaction() (*redbird.sql.Table method*), 42

U

update() (*in module redbird.sql*), 43
update() (*redbird.base.BaseRepo method*), 58
update() (*redbird.base.BaseResult method*), 59
update() (*redbird.repos.CSVFileRepo method*), 15
update() (*redbird.repos.JSONDirectoryRepo method*), 17
update() (*redbird.repos.MemoryRepo method*), 13
update() (*redbird.repos.MongoRepo method*), 26
update() (*redbird.repos.SQLExprRepo method*), 19
update() (*redbird.repos.SQLRepo method*), 24
update() (*redbird.sql.Table method*), 40
update() (*redbird.templates.TemplateResult method*), 56
upsert() (*redbird.base.BaseRepo method*), 57