Pytest Mock Resources

Release 1.2.2

unknown

Apr 05, 2022

CONTENTS:

CHAPTER

ONE

QUICKSTART

` .. image:: https://readthedocs.org/projects/pytest-mock-resources/badge/?version=latest

target https://readthedocs.org/projects/pytest-mock-resources/badge/?version=latest

alt Documentation

Status <https://pytest-mock-resources.readthedocs.io/en/latest/?badge=latest>`_

1.1 Introduction

Code which depends on external resources such a databases (postgres, redshift, etc) can be difficult to write automated tests for. Conventional wisdom might be to mock or stub out the actual database calls and assert that the code works correctly before/after the calls.

However take the following, *simple* example:

```
def serialize(users):
    return [
        {
            'user': user.serialize(),
            'address': user.address.serialize(),
            'purchases': [p.serialize() for p in user.purchases],
        }
        for user in users
    ]

def view_function(session):
    users = session.query(User).join(Address).options(selectinload(User.purchases)).
↪all()
    return serialize(users)
```

Sure, you can test `serialize`, but whether the actual **query** did the correct thing *truly* requires that you execute the query.

1.2 The Pitch

Having tests depend upon a **real** postgres instance running somewhere is a pain, very fragile, and prone to issues across machines and test failures.

Therefore `pytest-mock-resources` (primarily) works by managing the lifecycle of docker containers and providing access to them inside your tests.

As such, this package makes 2 primary assumptions:

- You're using `pytest` (hopefully that's appropriate, given the package name)
- For many resources, `docker` is required to be available and running (or accessible through remote docker).

If you aren't familiar with Pytest Fixtures, you can read up on them in the Pytest documentation.

In the above example, your test file could look something like

```
from pytest_mock_resources import create_postgres_fixture
from models import ModelBase

pg = create_postgres_fixture(ModelBase, session=True)

def test_view_function_empty_db(pg):
  response = view_function(pg)
  assert response == ...

def test_view_function_user_without_purchases(pg):
  pg.add(User(...))
  pg.flush()

  response = view_function(pg)
  assert response == ...

def test_view_function_user_with_purchases(pg):
  pg.add(User(..., purchases=[Purchase(...)]))
  pg.flush()

  response = view_function(pg)
  assert response == ...
```

1.3 Existing Resources (many more possible)

- SQLite

  ```
  from pytest_mock_resources import create_sqlite_fixture
  ```

- Postgres

  ```
  from pytest_mock_resources import create_postgres_fixture
  ```

- Redshift

  note Uses postgres under the hood, but the fixture tries to support as much redshift functionality as possible (including redshift's `COPY`/`UNLOAD` commands).

```
from pytest_mock_resources import create_redshift_fixture
```

- Mongo

```
from pytest_mock_resources import create_mongo_fixture
```

- Redis

```
from pytest_mock_resources import create_redis_fixture
```

- MySQL

```
from pytest_mock_resources import create_mysql_fixture
```

1.4 Features

General features include:

- Support for "actions" which pre-populate the resource you're mocking before the test
- Async fixtures
- Custom configuration for container/resource startup

1.5 Installing

```
# Basic fixture support
pip install "pytest-mock-resources"

# For postgres install EITHER of the following:
pip install "pytest-mock-resources[postgres-binary]"
pip install "pytest-mock-resources[postgres]"

# For postgres async
pip install "pytest-mock-resources[postgres-async]"

# For redshift install EITHER of the following:
# (redshift fixtures require postgres dependencies...)
pip install "pytest-mock-resources[postgres, redshift]"
pip install "pytest-mock-resources[postgres-binary, redshift]"

# For mongo install the following:
pip install "pytest-mock-resources[mongo]"

# For redis
pip install "pytest-mock-resources[redis]"

# For mysql
pip install "pytest-mock-resources[mysql]"
```

1.6 Possible Future Resources

- Rabbit Broker
- AWS Presto

Feel free to file an issue if you find any bugs or want to start a conversation around a mock resource you want implemented!

1.7 Python 2

Releases in the 1.x series were supportive of python 2. However starting from 2.0.0, support for python 2 was dropped. We may accept bugfix PRs for the 1.x series, however new development and features will not be backported.

CHAPTER

TWO

DOCKER

In order to run tests which interact with **most** fixture types (sqlite being an example of one such exception). Docker needs to be available and running:

Make sure you have docker installed:

- MacOs
- Nix
- Windows

Once you have docker installed, `pytest` will automatically up and down any necessary docker containers so you don't have to, by default all containers will be spun up/down per `pytest` invocation.

2.1 Startup Lag

The default execution mode will kill all the containers that were started by the test suite each time it's executed.

However, some containers have a larger startup cost than others; Mongo and presto, in particular have significant startup costs that don't play well with rapid development. Even postgres though, has ~ 1-2 seconds delay each time you run `pytest` which can quickly become annoying.

To circumvent this cost, `pytest-mock-resources` ships with a small CLI tool called `pmr`, which simply starts a single persistent container per fixture type (though redshift/postgres share one), which avoids that cost!

For Redshift and Postgres:

```
$ pmr postgres
711f5d5a86896bb4eb76813af4fb6616aee0eff817cdec6ebaf4daa0e9995441
# or specify the image
PMR_POSTGRES_IMAGE=postgres:11 pmr postgres
711f5d5a86896bb4eb76813af4fb6616aee0eff817cdec6ebaf4daa0e9995441
```

For Mongo:

```
$ pmr mongo
711f5d5a86896bb4eb76813af4fb6616aee0eff817cdec6ebaf4daa0e9995441
# or specify the image
PMR_MONGO_IMAGE=mongo:5.0 pmr mongo
711f5d5a86896bb4eb76813af4fb6616aee0eff817cdec6ebaf4daa0e9995441
```

For MySQL:

```
$ pmr mysql
711f5d5a86896bb4eb76813af4fb6616aee0eff817cdec6ebaf4daa0e9995441
# or specify the image
PMR_MYSQL_IMAGE=postgres:8.0 pmr mysql
711f5d5a86896bb4eb76813af4fb6616aee0eff817cdec6ebaf4daa0e9995441
```

You can check on the instance's state via:

```
$ docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED
↪     STATUS              PORTS                     NAMES
711f5d5a8689        postgres:9.6.10-alpine   "docker-entrypoint.s..."   16 seconds␣
↪ago       Up 15 seconds       0.0.0.0:5532->5432/tcp   determined_euclid
```

You can terminate the instance whenever you want via:

```
$ docker stop 711f5d5a8689  # where 711f5d5a8689 is the `CONTAINER ID` from `docker␣
↪ps`
711f5d5a86896bb4eb76813af4fb6616aee0eff817cdec6ebaf4daa0e9995441
```

2.2 Config

In order to support various projects and environments in which tests might be run, each docker-based fixture has the ability to customize its default configuration.

The precedence of the available config mechanisms follow the order:

- Environment variables
- Fixture Configuration
- Default Configuration

2.2.1 Environment Variables

In general we would only recommend use of the environment variable config for temporary changes to a value, or for configuration that is specific to the environment in which it is being run.

A common use case for this mechanism is local port conflicts. When a container is started up, we bind to a pre-specified port for that resource kind. We (attempt to) avoid conflicts by binding to a non-standard port for that resource by default, but conflicts can still happen

All configuration options for the given resource are available under env vars named in the pattern:

```
PMR_{RESOURCE}_{CONFIG}
# e.x.
export PMR_POSTGRES_PORT=54321
```

Resource is the name of the resource, i.e. POSTGRES, MONGO, REDIS, etc

CONFIG is the name of the config name. Every container will support at **least**: IMAGE, HOST, PORT, and CI_PORT.

2.2.2 Fixture Configuration

In general, we recommend fixture configuration for persistent configuration that is an attribute of the project itself, rather than the environment in which the project is being run.

The most common example of this will be `image`. If you're running postgres:8.0.0 in production, you should not be testing with our default image version! Other resource-specific configurations, such as `root_database`, might also be typical uses of this mechanism.

Here, the pattern is by defining a fixture in the following pattern:

```
@pytest.fixture(scope='session')
def pmr_{resource}_config():
    return {Resource}Config(...options...)
```

I.e. `pmr_postgres_config`, returning a `PostgresConfig` type. might look like

Listing 1: conftest.py

```
import pytest
from pytest_mock_resources import PostgresConfig

@pytest.fixture(scope='session')
def pmr_postgres_config():
    return PostgresConfig(image='postgres:11.0.0')
```

2.2.3 Default Configuration

Default configuration uses the same mechanism (i.e. fixture configuration) as you might, to pre-specify the default options, so that the plugin can usually be used as-is with no configuration.

The configuration defaults should not be assumed to be static/part of the API (and typically changes should be irrelevant to most users).

See the *API* docs for details on the current defaults.

2.3 Testing from WITHIN a container

Add the following mount to your `docker run` command which will allow `pytest` to communicate with your host machine's docker CLI:

```
docker run -v /var/run/docker.sock:/var/run/docker.sock [..other options] <IMAGE>
```

CHAPTER

THREE

FIXTURES

This package gives you the capability to create as many fixtures to represent as many mock instances of e.g. SQLite, Postgres, Mongo, etc might **actually** exist in your system.

Furthermore, you can prepopulate the connections those fixtures yield to you with whatever DDL, preset data, or functions you might require.

Each fixture you create can be used in multiple tests without risking data leakage or side-effects from one test to another, see *Internals* for more details.

See *Config* for information on customizing the configuration for docker-based fixtures.

3.1 Relational Database Fixtures

There is a page specifically for *Redshift* , but all the relational database (sqlite, postgres, redshift, mysql etc) fixtures have the same signatures, and therefore support the same utilities for running DDL, setup, and whatnot.

3.1.1 Basics

Say you had a function which required a redshift engine.

```
# src/package/utilities.py
def sql_sum(redshift_conn):
    """SUPER OPTIMIZED WAY to add up to 15.

    Better than C or Go!
    """
    redshift_conn.execute("CREATE TEMP TABLE mytemp(c INT);")
    redshift_conn.execute(
        """
        INSERT INTO mytemp(c)
        VALUES (1),
            (2),
            (3),
            (4),
            (5);
        """
    )

    return redshift_conn.execute("SELECT SUM(c) FROM mytemp;").fetchone()
```

To test this, we first need to create a `redshift fixture`:

```
# tests/conftest,py
from pytest_mock_resources import create_redshift_fixture

redshift = create_redshift_fixture()
```

By putting this in the top-level `conftest.py` file, it is made available to all tests, though **note** you can create the fixture at at any location, in order scope it to a subset of your tests.

We then create a test that leverages the fixture to test the function:

```
# tests/test_utilities.py
from package import utilities

# Supply the redshift fixture as an argument
def test_sql_sum(redshift):
    # Run the function we want to test
    result = utilities.sql_sum(conn)

    assert result == (15,)
```

We can then run a *pytest* command and confirm the function works as expected!

3.1.2 Custom Engines

Suppose you you're doing end-to-end testing, which includes the code which loads your configuration required to *create* an engine. The vanilla `redshift` fixture above won't quite cut it in that case.

```
# src/package/entrypoint.py

import psycopg2
import sqlalchemy

def psycopg2_main(**config):
    conn = psycopg2.connect(**config)
    do_the_thing(conn)
    ...

def sqlalchemy_main(**config):
    conn = sqlalchemy.create_engine(**config)
    do_the_thing(conn)
    ...
```

As you can see, we must pass in valid credentials to *create* an engine, rather than the engine itself. You'll need to make use of the `pmr_credentials` attribute.

```
from pytest_mock_resources import (
    create_postgres_fixture,
    create_redshift_fixture,
)

from package import entrypoint

postgres = create_postgres_fixture()
redshift = create_redshift_fixture()

def test_psycopg2_main_postgres(postgres):
```

(continues on next page)

(continued from previous page)

```
    credentials = postgres.pmr_credentials
    result = entrypoint.psycopg2_main(**credentials.as_psycopg2_connect_args())
    assert result ...

def test_psycopg2_main_redshift(redshift):
    credentials = redshift.pmr_credentials
    result = entrypoint.psycopg2_main(**credentials.as_psycopg2_connect_args())
    assert result ...
```

As you can see, `postgres` and `redshift` both work the same way, and can provide arguments directly into the psycopg2 `connect` function. Now lets see how `sqlalchemy` can work.

```
def test_sqlalchemy_main_postgres(postgres):
    credentials = postgres.pmr_credentials
    result = entrypoint.sqlalchemy_main(**credentials.as_url())
    assert result ...

def test_sqlalchemy_main_redshift(redshift):
    credentials = redshift.pmr_credentials
    result = entrypoint.sqlalchemy_main(**credentials.as_url())
    assert result ...
```

Again, all fixtures have the same interface. `as_url()` returns an inter-dbapi compatible connection string, which should be a valid input to all the supported connection modes. However, there are also sqlalchemy, psycopg2, and mongo specific conversion functions. In case that's required. And finally, if all else fails, `pmr_credentials` is an object with public attributes for all the connection information.

3.1.3 Preset DDL/Data

The above examples are fairly trivial, in a more realistic situation you would be dealing with pre-set DDL and data.

To address this, the `create_*_fixture` functions can take in an arbitrary amount of **ordered** arguments which can be used to *setup* the fixture prior to you using it. The following is a list of all possible arg types:

- A `Statements` instance.
  - An iterable of executable `strings` to run against the database fixture.
- A SQLAlchemy metadata or declarative base instance.
  - Used to pre-populate the fixture with all Schemas and Tables found on the instance.
- A `Rows` instance.
  - An iterable of SQLAlchemy model instances to create DDL and then populate it with data from these model instances.

Adding any of these to your `create_*_fixture` call will result in a fixture which is preset with whatever DDL and data you provided.

These fixtures will also reset to the pre-set data/DDL every time they are used - there will be NO data leakage or side-effects from one test to the other.

Statements

The `Statements` class is used to supply a `create_*_function` with an ordered list of statments to execute.

```
# tests/conftest.py:

from pytest_mock_resources import create_redshift_fixture, Statements

statements = Statements(
    """
    CREATE TABLE account(
      user_id serial PRIMARY KEY,
      username VARCHAR (50) UNIQUE NOT NULL,
      password VARCHAR (50) NOT NULL
    );
    """,
    """
    INSERT INTO account VALUES (1, 'user1', 'password1')
    """,
)

redshift = create_redshift_fixture(
    statements,
)
```

Note, you can execute arbitrary SQL strings or `SQLAlchemy` statements (DDL or otherwise), and the listed statements will be executed on a fresh database for each test. This ensures that the state of the database is identical across all tests which share that fixture.

```
# tests/test_something.py:

def test_something_exists(redshift):
    execute = redshift.execute("SELECT password FROM account")

    result = sorted([row[0] for row in execute])
    assert ["password1"] == result
```

Metadata/Models

The SQLAlchemy ORM allows you to define declarative models to represent your database tables and then use those models to interface with your database.

The `create_*_fixture` functions can take in any number of SQLAlchemy metadata or Base instances and use them to to create DDL.

For example, given a models package:

```
# src/package/models.py:

from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = "user"
    __table_args__ = {"schema": "stuffs"}
```

(continues on next page)

(continued from previous page)

```
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String)
```

A corresponding test file could look like

```
# tests/test_user.py:

from package.models import Base
from pytest_mock_resources import create_postgres_fixture

pg = create_redshift_fixture(
    Base,

    # Of course you can use this with statements
    Statements("INSERT INTO stuffs.user(name) VALUES ('Picante', )"),
)

def test_something_exists(pg):
    # Insert a row into the user table DURING the test
    pg.execute("INSERT INTO stuffs.user(name) VALUES ('Beef', )")

    # Confirm that the user table exists and the row was inserted
    rows = pg.execute("SELECT name FROM stuffs.user")
    result = [row[0] for row in rows]
    assert ["Picante", "Beef"] == result
```

Even if you don't plan on using SQLAlchemy models or the ORM layer throughout your actual code, defining these models can be EXTREMELY beneficial for DDL maintenance and testing.

---

info

If you are working on a new project which requires a SQL Database layer, we HIGHLY recommend using SQLAlchemy in combination with alembic to create and maintain your DDL.

---

info

If you are working on a project with pre-existing DDL, you can use a tool like sqlacodegen to generate the models from your current DDL!

Dealing with Bloated Metadata

By default, each DB fixture recreates the whole database from scratch prior to each test to ensure there are no side-effects from one test to another.

Recreating DDL is generally fairly quick but if there are a large amount of tables to create, test setup-time can begin to suffer. In one of our databases, there are more than a 1000 tables! As a result, it takes ~5 seconds for each test to setup which is unacceptable. If you have 200 tests running linearly, you will be spending an additional ~17 minutes, waiting for tests to complete.

To counteract this, you can provide an iterable of table names to your `create_*_fixture` call. This will tell the call to ONLY create the tables you have specified instead of creating all of them.

This can be a great way to keep track of all the tables a given block of code interacts with as well!

```
# tests/conftest.py:

from pytest_mock_resources import create_redshift_fixture, Statements
from redshift_schema import meta, example_table

redshift = create_redshift_fixture(
    meta,
    statements,
    Statements(
        example_table.insert().values(name="ABCDE"),
    ),

    # ONLY create this single table for this test.
    tables=[
        example_table,
        "example_table_mapping_table",
    ]
)
```

```
# tests/test_something.py:

def test_something_exists(redshift):
    execute = redshift.execute("SELECT network FROM warehouse.warehouse_stacked_data")

    # Confirm that the warehouse.warehouse_stacked_data table exists and the row was␣
↪inserted
    result = sorted([row[0] for row in execute])
    assert ["ABCDE"] == result
```

The `tables` argument accepts any of:

- SQLAlchemy declarative model class
- SQLAlchemy table object
- Exact string table name
- Globbed table name

  Globbing, in comparison to regular expressions, in this context tends to lead to shorter and easier to read definitions. This is especially true when one uses schemas, leading to `.` literals in your fully qualified table names.

  ```
  create_<backend>_fixture(Base, tables=['schema.*'])  # Only tables for a specific␣
  ↪schema
  create_<backend>_fixture(Base, tables=['category_*'])  # Only tables with a␣
  ↪specific suffix
  create_<backend>_fixture(Base, tables=['*_category'])  # Only tables with a␣
  ↪specific prefix
  ```

Rows

If you are using SQLAlchemy to maintain your DDL, you have the capability to use the `Rows` class to conveniently pre-populate your db fixture with DDL and data in a single line.

```
# src/package/models.py:

from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = "user"

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String)
```

```
# tests/conftest.py:

from package.models import User  # These models could be imported from ANYWHERE
from pytest_mock_resources import create_redshift_fixture, Rows

rows = Rows(
    User(name="Harold"),
    User(name="Catherine"),
)

redshift = create_redshift_fixture(
    # This will AUTOMATICALLY create ALL the schemas and tables found in any row's␣
↪metadata
    # and then populate those tables via the given model instances.
    rows,
)
```

```
# tests/test_something.py:

def test_something_exists(redshift):
    execute = redshift.execute("SELECT * FROM user")

    # Confirm that the user table exists and the rows were inserted
    result = sorted([row[1] for row in execute])
    assert ["Catherine", "Harold"] == result
```

Functions

Uses can supply a function as an input argument to the fixtures as well:

```
# Create models with relationships
class User(Base):
    __tablename__ = "user"
    __table_args__ = {"schema": "stuffs"}

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
```

(continues on next page)

(continued from previous page)

```
    objects = relationship("Object", back_populates="owner")


class Object(Base):
    __tablename__ = "object"
    __table_args__ = {"schema": "stuffs"}

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
    belongs_to = Column(Integer, ForeignKey('stuffs.user.id'))

    owner = relationship("User", back_populates="objects")


# Leverage model relationships in a seed data function
def session_fn(session):
    session.add(User(name='Fake Name', objects=[Object(name='Boots')]))


# Leverage seed data function to create seeded fixture
postgres = create_postgres_fixture(Base, session_fn)


# Leverage seeded fixture
def test_session_function(postgres):
    execute = postgres.execute("SELECT * FROM stuffs.object")
    owner_id = sorted([row[2] for row in execute])[0]

    execute = postgres.execute("SELECT * FROM stuffs.user where id = {id}".
↪format(id=owner_id))
    result = [row[1] for row in execute]

    assert result == ['Fake Name']
```

3.2 Redshift

3.2.1 COPY/UNLOAD

`COPY` and `UNLOAD` will work out of the box, when you're testing code which accepts a sqlalchemy engine or session object, because we can preconfigure it to work properly. In these scenarios, you should simply be able to send in the fixture provided into your test and be on your merry way.

This **should** also work seamlessly if you're testing code which creates its own connection directly. Consider the following module that creates a redshift engine and then uses said engine to run a `COPY` command:

```
# src/app.py
from sqlalchemy import create_engine

def main(**connect_args):
    engine = get_redshift_engine(connect_args)

    return run_major_thing(engine)
```

(continues on next page)

(continued from previous page)

```
def get_redshift_engine(config):
    return create_engine(**config)

def run_major_thing(engine):
    engine.execute(
        """
        COPY x.y FROM 's3://bucket/file.csv' credentials....
        """
    )
```

The `redshift` fixture should automatically instrument direct calls to `psycopg2.connect()` (or `sqlalchemy.create_engine()`.

3.3 SQLite

While SQLite is a widely used database in its own right, we also aim to make SQLite a reasonable stand-in for (at least) postgres, in tests where possible. We **do** make postgres tests run as fast as possible, but tests using postgres objectively run more slowly than those with SQLite.

While SQLite cannot match Postgres perfectly, in many scenarios (particularly those which use ORMs, which tend to make use of cross-compatible database features) it can be used to more quickly verify the code. And in the event that you begin using a feature only supportable in postgres, or which behaves differently in SQLite, you're one `s/create_slite_fixture/create_postgres_fixture` away from resolving that problem. Additionally, you can choose to only use postgres for the subset of tests which require such features.

To that end, we've extended the sqlalchemy SQLite dialect to include features to match postgres as closely as possible. We **do** however, recommend that use of this dialect is restricted purely to **tests** in order to be used as a postgres stand-in, rather than for use in actual application code.

3.3.1 Schemas

As an in-memory database (for the most part), SQLite does not behave the same way when encountering schemas.

For example, given sqlalchemy model defined as:

```
from .models import ModelBase

class User(ModelBase):
    __tablename__ ='user'
    __table_args__ = {'schema': 'private'}
```

SQLite generally would produce an error upon use of that table, but will now work by default, and behave similarly to postgres.

A caveat to this is that SQLite has no notion of a "search path" like in postgres. Therefore, programatic use altering the search path from the default "public" (in postgres), or referencing a "public" table as "public.tablename" would not be supported.

3.3.2 Foreign Keys

SQLite supports FOREIGN KEY syntax when emitting CREATE statements for tables, however by default these constraints have no effect on the operation of the table.

We simply, turn that support on by default, to match the postgres behavior.

3.3.3 JSON/JSONB

Tables which use either `sqlalchemy.dialects.postgresql.JSON/JSONB` or `sqlalchemy.types.Json` will work as they would in postgres.

SQLite itself, recently added support for json natively, but this allows a much wider version range of SQLite to support that feature.

3.3.4 Datetime (timezone support)

By default, SQLite does not respect the `Datetime(timezone=True)` flag. This means that normally a `Datetime` column would behave differently from postgres. For example, where postgres would return timezone-aware `datetime.datetime` objects, SQLite would return naive `datetime.datetime` (which do **not** behave the same way when doing datetime math).

This does **not** actually store the timezones of the datetime (as is also true for postgres). It simply matches the timezone-awareness and incoming timezone conversion behavior you see in postgres.

3.4 Mongo

Users can test MongoDB dependant code using the *create_mongo_fixture*.

Consider the following example:

```
# src/some_module.py

def insert_into_customer(mongodb_connection):
    collection = mongodb_connection['customer']
    to_insert = {"name": "John", "address": "Highway 37"}
    collection.insert_one(to_insert)
```

A user can test this as follows:

```
# tests/some_test.py

from pytest_mock_resources import create_mongo_fixture
from some_module import insert_into_customer

mongo = create_mongo_fixture()

def test_insert_into_customer(mongo):
    insert_into_customer(mongo)

    collection = mongo['customer']
    returned = collection.find_one()

    assert returned == {"name": "John", "address": "Highway 37"}
```

3.4.1 Custom Connections

Custom connections can also be generated via the fixture's yielded attributes/MONGO_* fixtures:

```
# tests/some_test.py

from pymongo import MongoClient

from pytest_mock_resources import create_mongo_fixture

mongo = create_mongo_fixture()


def test_create_custom_connection(mongo):
    client = MongoClient(**mongo.pmr_credentials.as_mongo_kwargs())
    db = client[mongo.config["database"]]

    collection = db["customers"]
    to_insert = [
        {"name": "John"},
        {"name": "Viola"},
    ]
    collection.insert_many(to_insert)

    result = collection.find().sort("name")
    returned = [row for row in result]

    assert returned == to_insert
```

3.5 Redis

Users can test Redis dependent code using the *create_redis_fixture*.

pytest_mock_resources.**create_redis_fixture**(*scope='function'*)

Produce a Redis fixture.

Any number of fixture functions can be created. Under the hood they will all share the same database server.

Note: If running tests in parallel, the implementation fans out to different redis "database"s, up to a 16 (which is the default container fixed limite). This means you can only run up to 16 simultaneous tests.

Additionally, any calls to *flushall* or any other cross-database calls **will** still represent cross-test state.

Finally, the above notes are purely describing the current implementation, and should not be assumed. In the future, the current database selection mechanism may change, or databases may not be used altogether.

Parameters **scope** (*str*) – The scope of the fixture can be specified by the user, defaults to "function".

Raises **KeyError** – If any additional arguments are provided to the function than what is necessary.

Consider the following example:

```
# src/some_module.py

def insert_into_friends(redis_client):
    redis_client.sadd("friends:leto", "ghanima")
    redis_client.sadd("friends:leto", "duncan")
    redis_client.sadd("friends:paul", "duncan")
    redis_client.sadd("friends:paul", "gurney")
```

A user can test this as follows:

```
# tests/some_test.py

from pytest_mock_resources import create_redis_fixture
from some_module import insert_into_friends

redis = create_redis_fixture()

def test_insert_into_friends(redis):
    insert_into_friends(redis)

    friends_leto = redis.smembers("friends:leto")
    friends_paul = redis.smembers("friends:paul")

    assert friends_leto == {b"duncan", b"ghanima"}
    assert friends_paul == {b"gurney", b"duncan"}
```

3.5.1 Manual Engine Creation

Engines can be created manually via the fixture's yielded attributes/REDIS_* fixtures:

```
# tests/some_test.py

from redis import Redis

from pytest_mock_resources import create_redis_fixture

redis = create_redis_fixture()


def test_create_custom_connection(redis):
    client = Redis(**redis.pmr_credentials.as_redis_kwargs())
    client.set("foo", "bar")
    client.append("foo", "baz")
    value = client.get("foo").decode("utf-8")
    assert value == "barbaz"
```

3.6 Internals

TODO: Describe internals of what PMR is doing under the hood, i think it's interesting!

CHAPTER

FOUR

ASYNC

In general, pytest-mock-resources >=2.0 is required for async support and will naturally require python >= 36.

Async is easily supportable **outside** pytest-mock-resources, by simply using the *pmr_<resource>_config* fixture for the given resource to get a handle on the requisite configuration required to produce a client yourself.

For example:

```
from sqlalchemy.engine.url import URL
from sqlalchemy.ext.asyncio import create_async_engine

@pytest.fixture
def async_pg(pmr_postgres_config):
    # or `URL.create` in sqlalchemy 1.4+
    create_async_engine(URL(host=pmr_postgres_config.host, database=pmr_postgres_
↪config.database, ...))
```

However, we're happy to support default/built-in async client implementations where applicable.

Today, async engines are implemented for: * postgres, using sqlalchemy >=1.4 (with the asyncpg driver)

Generally, support will be available on a per-fixture basis by way of specifying *async_=True* to the fixture creation function.

For example

```
import pytest
from sqlalchemy import text
from pytest_mock_resources import create_postgres_fixture

postgres_async = create_postgres_fixture(async_=True)

@pytest.mark.asyncio
async def test_basic_postgres_fixture_async(postgres_async):
    async with postgres_async.connect() as conn:
        await conn.execute(text("select 1"))
```

4.1 pytest-asyncio

Generally you will want *pytest-asyncio* or similar to be installed. This will allow your async fixture to work the same way normal fixtures function.

CHAPTER

FIVE

CI SERVICE SUPPORT

Depending on the CI service, access to docker-related fixtures may be different than it would be locally. As such, below is an outline of how to support those fixtures within specific CI services.

5.1 CircleCi

CircleCI 2.0+ default jobs do not have access to a docker directly, but instead interact with a remote docker.

As such, you will need to include the a step in your job to setup remote docker like so:

```
steps:
  - setup_remote_docker
  - checkout
  ...
```

Furthermore, you should start the service ahead of time using their mechanism of choice:

For 2.0 jobs

```
jobs:
  <YOUR JOB NAME>:
      docker:
      - image: <YOUR IMAGE>
      - image: <SERVICE IMAGE>
```

For 2.1+ jobs

```
version: 2.1

executors:
  foo:
    docker:
      - image: <YOUR IMAGE>
      - image: <SERVICE IMAGE>

jobs:
  test:
    executor: foo
```

5.1.1 Postgres/Redshift Container

Specifically for postgres/redshift, the `- image: <SERVICE IMAGE>` portion should look like

```
- image: postgres:9.6.10-alpine  # or whatever image/tag you'd like
  environment:
    POSTGRES_DB: dev
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
```

You will receive a *ContainerCheckFailed: Unable to connect to [...] Postgres test container* error in CI if the above is not added to you job config.

5.1.2 Mongo Container

Specifically for mongo, the `- image: <SERVICE IMAGE>` portion should look like

```
- image: circleci/mongo:3.6.12   # or whatever image/tag you'd like
  command: "mongod --journal"
```

You will receive a *ContainerCheckFailed: Unable to connect to [...] Mongo test container* error in CI if the above is not added to you job config.

5.2 GitLab

For `pytest-mock-resources` to work on GitLab use of `dind` service is required. Below is a sample configuration:

```
services:
  - docker:dind

variables:
  DOCKER_TLS_CERTDIR: ''

stages:
  - testing

testing-job:
  image: python:3.6.8-slim  # Use a python version that matches your project
  stage: testing
  variables:
    DOCKER_HOST: tcp://docker:2375
    PYTEST_MOCK_RESOURCES_HOST: docker
  before_script:
    - apt-get update && apt-get install -y wget libpq-dev gcc
    - wget -O get-docker.sh https://get.docker.com
    - chmod +x get-docker.sh && ./get-docker.sh
  script:
    - pip install -r requirements.txt
    - pytest -x tests
```

CHAPTER

SIX

CONTRIBUTING

6.1 Prerequisites

If you are not already familiar with Poetry, this is a poetry project, so you'll need this!

6.2 Getting Setup

See the `Makefile` for common commands, but for some basic setup:

```
# Installs the package with all the extras
make install
```

And you'll want to make sure you can run the tests and linters successfully:

```
# Runs CI-level tests, with coverage reports
make test lint
```

6.2.1 Tests

A feature of the package is that it doesn't stop you from running tests in parallel, such as by using `pytest-xdist`. As such `make test` runs the tests in a few different modes.

In general, you can simply run *pytest*, or e.x. *pytest tests/fixture/database/test_udf.py* to run specific subsets of the tests.

6.2.2 Docs

First, install the docs requirements with `pip install -r docs/requirements.txt`, then use `sphinx` as normal. i.e.

```
cd docs
make html  # one-time build of the docs
# or
make livehtml  # Starts a webserver with livereload of changes
```

6.3 Need help

Submit an issue!

CHAPTER

SEVEN

API

7.1 Fixture Functions

pytest_mock_resources.**create_mongo_fixture**(*scope='function'*)

Produce a mongo fixture.

Any number of fixture functions can be created. Under the hood they will all share the same database server.

Parameters **scope** – Passthrough pytest's fixture scope.

pytest_mock_resources.**create_mysql_fixture**(**ordered_actions*, *scope='function'*, *tables=None*, *session=None*)

Produce a MySQL fixture.

Any number of fixture functions can be created. Under the hood they will all share the same database server.

Parameters

- **ordered_actions** – Any number of ordered actions to be run on test setup.
- **scope** – Passthrough pytest's fixture scope.
- **tables** – Subsets the tables created by *ordered_actions*. This is generally most useful when a model-base was specified in *ordered_actions*.
- **session** – Whether to return a session instead of an engine directly. This can either be a bool or a callable capable of producing a session.

pytest_mock_resources.**create_postgres_fixture**(**ordered_actions*, *scope='function'*, *tables=None*, *session=None*, *async_=False*, *createdb_template='template1'*, *engine_kwargs=None*)

Produce a Postgres fixture.

Any number of fixture functions can be created. Under the hood they will all share the same database server.

Parameters

- **ordered_actions** – Any number of ordered actions to be run on test setup.
- **scope** – Passthrough pytest's fixture scope.
- **tables** – Subsets the tables created by *ordered_actions*. This is generally most useful when a model-base was specified in *ordered_actions*.
- **session** – Whether to return a session instead of an engine directly. This can either be a bool or a callable capable of producing a session.
- **async** – Whether to return an async fixture/client.

- **createdb_template** – The template database used to create sub-databases. "template1" is the default chosen when no template is specified.
- **engine_kwargs** – Optional set of kwargs to send into the engine on creation.

pytest_mock_resources.**create_redis_fixture**(*scope='function'*)

Produce a Redis fixture.

Any number of fixture functions can be created. Under the hood they will all share the same database server.

Note: If running tests in parallel, the implementation fans out to different redis "database"s, up to a 16 (which is the default container fixed limite). This means you can only run up to 16 simultaneous tests.

Additionally, any calls to *flushall* or any other cross-database calls **will** still represent cross-test state.

Finally, the above notes are purely describing the current implementation, and should not be assumed. In the future, the current database selection mechanism may change, or databases may not be used altogether.

Parameters **scope** (*str*) – The scope of the fixture can be specified by the user, defaults to "function".

Raises **KeyError** – If any additional arguments are provided to the function than what is necessary.

pytest_mock_resources.**create_redshift_fixture**(**ordered_actions*, *scope='function'*, *tables=None*, *session=None*, *async_=False*, *createdb_template='template1'*, *engine_kwargs=None*)

Produce a Redshift fixture.

Any number of fixture functions can be created. Under the hood they will all share the same database server.

Note that, by default, redshift uses a postgres container as the database server and attempts to reintroduce appoximations of Redshift features, such as S3 COPY/UNLOAD, redshift-specific functions, and other specific behaviors.

Parameters

- **ordered_actions** – Any number of ordered actions to be run on test setup.
- **scope** – Passthrough pytest's fixture scope.
- **tables** – Subsets the tables created by *ordered_actions*. This is generally most useful when a model-base was specified in *ordered_actions*.
- **session** – Whether to return a session instead of an engine directly. This can either be a bool or a callable capable of producing a session.
- **async** – Whether to return an async fixture/client.
- **createdb_template** – The template database used to create sub-databases. "template1" is the default chosen when no template is specified.
- **engine_kwargs** – Optional set of kwargs to send into the engine on creation.

pytest_mock_resources.**create_sqlite_fixture**(**ordered_actions*, *scope='function'*, *tables=None*, *session=None*, *decimal_warnings=False*, *postgres_like=True*)

Produce a SQLite fixture.

Any number of fixture functions can be created. Under the hood they will all share the same database server.

Parameters

- **ordered_actions** – Any number of ordered actions to be run on test setup.
- **scope** – Passthrough pytest's fixture scope.
- **tables** – Subsets the tables created by *ordered_actions*. This is generally most useful when a model-base was specified in *ordered_actions*.
- **session** – Whether to return a session instead of an engine directly. This can either be a bool or a callable capable of producing a session.
- **decimal_warnings** – Whether to show sqlalchemy decimal warnings related to precision loss. The default *False* suppresses these warnings.
- **postgres_like** – Whether to add extra SQLite features which attempt to mimic postgres enough to stand in for it for testing.

class pytest_mock_resources.fixture.database.generic.**Credentials**(*drivername*, *host*, *port*, *database*, *username*, *password*)

Return as *pmr_credentials* attribute on supported docker-based fixtures.

Examples

It's also directly dict-able. >>> creds = Credentials('d', 'l', 'p', 'baz', 'user', 'pass') >>> dict_creds = dict(creds)

as_mongo_kwargs()
Return the valid arguments to a mongo client.

as_psycopg2_kwargs()
Return the valid arguments to sqlalchemy `psycopg2.connect`.

as_redis_kwargs()
Return the valid arguments to a redis client.

as_sqlalchemy_url()
Return a sqlalchemy `sqlalchemy.engine.url.URL`.

as_sqlalchemy_url_kwargs()
Return the valid arguments to sqlalchemy `sqlalchemy.engine.url.URL`.

as_url()
Return a stringified dbapi URL string.

7.2 Fixture Config

class pytest_mock_resources.**MongoConfig**(***kwargs*)
Define the configuration object for mongo.

Parameters

- **image** (*str*) – The docker image:tag specifier to use for mongo containers. Defaults to `"mongo:3.6"`.
- **host** (*str*) – The hostname under which a mounted port will be available. Defaults to `"localhost"`.
- **port** (*int*) – The port to bind the container to. Defaults to `28017`.

- **ci_port** (*int*) – The port to bind the container to when a CI environment is detected. Defaults to `27017`.
- **root_database** (*str*) – The name of the root mongo database to create. Defaults to `"dev-mongo"`.

class pytest_mock_resources.**MysqlConfig**(***kwargs*)

Define the configuration object for MySql.

Parameters

- **image** (*str*) – The docker image:tag specifier to use for mysql containers. Defaults to `"mysql:5.6"`.
- **host** (*str*) – The hostname under which a mounted port will be available. Defaults to `"localhost"`.
- **port** (*int*) – The port to bind the container to. Defaults to `5532`.
- **ci_port** (*int*) – The port to bind the container to when a CI environment is detected. Defaults to `5432`.
- **username** (*str*) – The username of the root user Defaults to `"user"`.
- **password** (*str*) – The password of the root password Defaults to `"password"`.
- **root_database** (*str*) – The name of the root database to create. Defaults to `"dev"`.

class pytest_mock_resources.**PostgresConfig**(***kwargs*)

Define the configuration object for postgres.

Parameters

- **image** (*str*) – The docker image:tag specifier to use for postgres containers. Defaults to `"postgres:9.6.10-alpine"`.
- **host** (*str*) – The hostname under which a mounted port will be available. Defaults to `"localhost"`.
- **port** (*int*) – The port to bind the container to. Defaults to `5532`.
- **ci_port** (*int*) – The port to bind the container to when a CI environment is detected. Defaults to `5432`.
- **username** (*str*) – The username of the root postgres user Defaults to `"user"`.
- **password** (*str*) – The password of the root postgres password Defaults to `"password"`.
- **root_database** (*str*) – The name of the root postgres database to create. Defaults to `"dev"`.

class pytest_mock_resources.**RedisConfig**(***kwargs*)

Define the configuration object for redis.

Parameters

- **image** (*str*) – The docker image:tag specifier to use for redis containers. Defaults to `"redis:5.0.7"`.
- **host** (*str*) – The hostname under which a mounted port will be available. Defaults to `"localhost"`.
- **port** (*int*) – The port to bind the container to. Defaults to `6380`.

- **ci_port** (*int*) – The port to bind the container to when a CI environment is detected. Defaults to `6379`.

pytest_mock_resources.**pmr_mongo_config**()
Override this fixture with a `MongoConfig` instance to specify different defaults.

Examples

```
>>> @pytest.fixture(scope='session')
... def pmr_mongo_config():
...     return MongoConfig(image="mongo:3.4", root_database="foo")
```

pytest_mock_resources.**pmr_mysql_config**()
Override this fixture with a `MysqlConfig` instance to specify different defaults.

Examples

```
>>> @pytest.fixture(scope='session')
... def pmr_mysql_config():
...     return MysqlConfig(image="mysql:5.2", root_database="foo")
```

pytest_mock_resources.**pmr_postgres_config**()
Override this fixture with a `PostgresConfig` instance to specify different defaults.

Examples

```
>>> @pytest.fixture(scope='session')
... def pmr_postgres_config():
...     return PostgresConfig(image="postgres:9.6.10", root_database="foo")
```

pytest_mock_resources.**pmr_redis_config**()
Override this fixture with a `RedisConfig` instance to specify different defaults.

Examples

```
>>> @pytest.fixture(scope='session')
... def pmr_redis_config():
...     return RedisConfig(image="redis:6.0")
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

A

C

M

P