Pytest Mock Resources

Release 1.2.2

unknown

Jul 24, 2023

CONTENTS:

CHAPTER

ONE

QUICKSTART

1.1 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.

` .. 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>`_

1.2 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
```

(continues on next page)

(continued from previous page)

```
    ]

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.

1.3 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 == ...
```

1.4 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
```

- Moto

```
from pytest_mock_resources import create_moto_fixture
```

1.5 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

1.6 Installation

```
# Basic fixture support i.e. SQLite
pip install "pytest-mock-resources"

# General, docker-based fixture support
pip install "pytest-mock-resources[docker]"

# Mongo fixture support, installs `pymongo`
pip install "pytest-mock-resources[mongo]"
```

(continues on next page)

(continued from previous page)

```
# Moto fixture support, installs non-driver extras specific to moto support
pip install "pytest-mock-resources[moto]"

# Redis fixture support, Installs `redis` client
pip install "pytest-mock-resources[redis]"

# Redshift fixture support, installs non-driver extras specific to redshift support
pip install "pytest-mock-resources[redshift]"
```

Additionally there are number of **convenience** extras currently provided for installing drivers/clients of specific features. However in most cases, you **should** already be installing the driver/client used for that fixture as as first-party dependency of your project.

As such, we recommend against using these extras, and instead explcitly depending on the package in question in your own project's 1st party dependencies.

```
# Installs psycopg2/psycopg2-binary driver
pip install "pytest-mock-resources[postgres-binary]"
pip install "pytest-mock-resources[postgres]"

# Installs asyncpg driver
pip install "pytest-mock-resources[postgres-async]"

# Installs pymysql driver
pip install "pytest-mock-resources[mysql]"
```

1.7 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!

1.8 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

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.

A new resource (database or otherwise) is created on a per test database, which allows each fixture to be used in multiple tests without risking data leakage or side-effects from one test to another.

Note: By default the underlying containers are reused across tests to amortize the container startup cost. Tests then create new "resources" (e.g. databases) within that container to avoid inter-test pollution.

This **can** cause inter-test dependencies if your tests are altering container-global resources like database users. In the event this is a problem, resources can be configured to **not** be session fixtures, although this will likely be drastically slower overall.

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

2.1 Relational Database Fixtures

All of the officially supported relational databases (SQLite, Postgres, Redshift, and MYSQL) support a minimum level of parity of features. Generally they will all have intercompatibile function signatures, except wherein there are particular features supported by a database which is unsupported in one of the others.

For database-specific support, see the corresponding *Redshift*, *Postgres*, or *SQLite* pages.

2.1.1 Basics

Say you've written a function, which accepts a `sqlalchemy.Engine`, and performs some operation which is literally not able to be tested without a connection to a real database.

Listing 1: package/utilities.py

```
def sql_sum(redshift_conn):
    """SUPER OPTIMIZED WAY to add up to 15.
    """
    redshift_conn.execute("CREATE TEMP TABLE mytemp(c INT);")
    redshift_conn.execute(
        """
        INSERT INTO mytemp(c)
```

(continues on next page)

(continued from previous page)

```
        VALUES (1), (2), (3), (4), (5);
        """
    )

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

With this library, you would define your test fixture, for the corresponding database in use. And then any references to that fixture in a test, will produce a `sqlalchemy.Engine`.

Alternatively, you can specify *session=True*, to ensure you're handed a `sqlalchemy.orm.Session` instead.

Listing 2: tests/test_utilities.py

```
# Redshift Example:
from pytest_mock_resources import create_redshift_fixture
from package.utilities import sql_sum

db = create_redshift_fixture()
# or
db = create_redshift_fixture(session=True)

def test_sql_sum(db):
    sql_sum(db)


# Postgres Example:
from pytest_mock_resources import create_postgres_fixture
from package.utilities import sql_sum

db = create_postgres_fixture()
# or
db = create_postgres_fixture(session=True)

def test_sql_sum(db):
    sql_sum(db)
```

Note that beyond your definition of the fixture, the test code remains exactly the same between examples among different databases.

What's happening when under the hood is that a docker container (except with SQLite) is being spun up on a per-test-session basis, and then individual sub-container databases are being created on a per-test basis, and yielded to each test.

2.1.2 Preset DDL/Data

In previous examples, we've largely ignored the reality that an empty database is often not useful by itself. You'll need to populate that database with some minimal amount of schemata and/or data in order to be useful.

To address this, the `create_*_fixture` functions take in an optional number of "Ordered Actions" which can be used to *setup* the fixture prior to you using it. As the name might imply, the "actions" are executed, in order, before the test body is entered.

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 most direct way to pre-create all the DDL which your code depends on, **particularly** if you already define `sqlalchemy.MetaData` or declarative models, would be to specify either as an ordered action.

For example, given a models package:

Listing 3: 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"}

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

A corresponding test file could look like

Listing 4: tests/test_user.py

```
from package.models import Base
from pytest_mock_resources import create_postgres_fixture

pg = create_redshift_fixture(Base)
# or
pg = create_redshift_fixture(Base.metadata)

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
```

Note: If you have split MetaData, you can pass in as many unique MetaData or declarative_base instances as necessary.

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!

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 might 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!

Listing 5: tests/conftest.py

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

redshift = create_redshift_fixture(
    meta,
    # ONLY create this small set of tables for this test.
    tables=[
        example_table,
        "example_table_mapping_table",
    ]
)
```

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 define your schema, you have the capability to use the `Rows` class to conveniently pre-populate your db fixture with data.

This will automatically insert any records defined by the `Rows` before test execution.

---

info

You can also omit the above `Base` reference to the model base or metadata when using rows, yielding `redshift = create_redshift_fixture(rows)`.

Rows will backtrack to the corresponding metadata and treat it as though the metadata were passed in immediately preceding the `Rows` action.

Statements/StaticStatements

Either a `Statements` or `StaticStatements` object can be constructed, which will execute arbitrary SQL before entering the test.

Both operate in exactly the same way, however `StaticStatements` let the library know that the included SQL statements are safe to "cache" in order to reduce database creation costs. For that reason, you should prefer a `StaticStatements` over a `Statements` where possible.

For example, the execution of DDL for which there is not a supported SQLALchemy abstraction, or other transaction-specific operations, are places where a static statement might be inappropriate.

Listing 6: tests/test_something.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)

def test_something_exists(redshift):
    execute = redshift.execute("SELECT password FROM account")
    result = sorted([row[0] for row in execute])
    assert ["password1"] == result
```

Note: You can either supply an SQL `str`, or `SQLAlchemy` statements (such as `text()`, select, insert, DDL, or other constructs),

---

Functions

Sometimes `Rows` or `Statements` are not dynamic enough. So any callable can be passed as an action. The only requirement is that it accept a lone argument for the test engine/session.

Note: The same object which is injected into the test function is handed to the provided function as its sole argument.

That is, if you provide `session=True`, you will receive a session object, whereas otherwise you will receive a vanilla engine object.

```
# 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)

    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']
```

2.1.3 Manually Constructed Engines

Due to the dynamic nature of the creation of the databases themselves, its non-trivial for a user to know what the connection string, for example, would be for the database ahead of time. Which makes testing code which manually constructs its own `sqlalchemy.Engine` objects internally more difficult.

Therefore, generally preferable way to use the fixtures is that you will be yielded a preconstructed engine pointing at the database to which your test is intended to run against; and to write your code such that it accepts the engine as a function/class parameter.

However, this is not always possible for all classes of tests, nor does it help for code which might already be written with a tightly coupled mechanism for engine creation.

For (contrived) example:

Listing 7: 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, in order to test these functions, we must pass in valid **credentials** rather than an engine itself.

pmr_credentials

Each of the fixtures you might create will attach a `pmr_credentials` attribute onto the engine it yields to the test which will be an instance of a `Credentials` class.

Attributes on this class include all the credentials required to connect to the particular database. Additionally, there are convenience methods specifically meant to coerce the credentials into a form directly accepted by common connection mechanisms like `psycopg2.connect` or `sqlalchemy.engine.url.URL`.

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

def test_sqlalchemy_main_postgres(postgres):
```

(continues on next page)

(continued from previous page)

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

2.1.4 Template Databases

Note: This feature was added in v2.4.0 and **currently only supports Postgres**.

The *template_database* fixture keyword argument, and :class:`StaticStatements` did not exist prior to this version.

By default, the supported fixtures attempt to amortize the cost of performing fixture setup through the creation of database templates (postgres). If, for whatever reason, this feature does not interact well with your test setup, you can disable the behavior by setting `template_database=False`.

With this feature enabled, all actions considered to be "safe" to statically will performed exactly once per test session, in a template database. This amortizes their initial cost and offloads the majority of the work to postgres itself. Then all "dynamic" actions will be performed on a per-test-database basis.

Consider the following fixture

```
from models import Base, Example
from pytest_mock_resources import create_postgres_fixture, StaticStatements,␣
↪Statements, Rows

def some_setup(engine):
    # some complex logic, not able to be performed as a `Statements`

pg = create_postgres_fixture(
    Base,
    Rows(Example(id=1)),
    StaticStatements('INSERT INTO foo () values ()'),
    Statements('INSERT INTO foo () values ()'),
    some_setup,
)
```

Each of the arguments given to `create_postgres_fixture` above are "actions" performed in the given order. Typically (in particular for non-postgres fixtures, today), all of the steps would be performed on a completely empty database prior to the engine/session being handed to the test function.

Static Actions

Static actions are actions which are safe to be executed exactly once, because they have predictable semantics which both safely be executed once per test session, as well as happen in a completely separate transactions and database, from the one handed to the test.

Static actions include:

- `MetaData`: `Base` in the above example is an alias to `Base.metadata`; i.e. a `sqlalchemy.MetaData` object that creates all objects defined on the metadata.
- `Rows`: Rows only work in the context of a session, and essentially define a set of `INSERT` statements to be run.

- `StaticStatements`: Are exactly like a `Statements` object. The subclass is simply a sentinel to signify that the user asserts their statement is "safe" to be executed as one of these static actions.

Dynamic Actions

Dynamic actions have unpredictable semantics, and as such the library cannot safely amortize their cost.

Dynamic actions include:

- `Statements`: A statement can be any arbitrary SQL, which therefore means we cannot know whether it will react negatively with the test, if executed in a separate transaction. If you're executing typical `CREATE`/`INSERT` statements, prefer `StaticStatements`.
- Functions: Obviously a function can do anything it wants, and therefore must be dynamic.

---

warning

It is important to consider action ordering when using dynamic actions. Upon encountering a dynamic action in a list of ordered actions, all subsequent actions will be executed as though they were dynamic (i.e. per-test-database).

You should therefore prefer to group all static actions before dynamic ones wherever possible to ensure you get the most optimal amortization of actions.

For example:

```
pg = create_postgres_fixture(
    Statements('CREATE TABLE ...'),
    Base,
)
```

This will execute all setup dynamically because it encountered a dynamic action (`Statements` in this case) first. Ideally the above actions would be reversed, or that the `Statements` be swapped for a `StaticStatements`.

2.2 Postgres

Note: The default postgres driver support is *psycopg2* for synchronous fixtures, and *asyncpg* for async fixtures. If you want to use a different driver, you can configure the *drivername* field using the *pmr_postgres_config* fixture:

```python from pytest_mock_resources import PostgresConfig

@pytest.fixture def pmr_postgres_config():

    PostgresConfig(drivername='postgresql+psycopg2') # but whatever drivername you require.

```

Note however, that the *asyncpg* driver **only** works with the async fixture, and the *psycopg2* driver **only** works with the synchronous fixture. These are inherent attributes of the drivers/support within SQLAlchemy.

2.3 Redshift

2.3.1 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)

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()`.

2.4 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.

2.4.1 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, programmatic use altering the search path from the default "public" (in postgres), or referencing a "public" table as "public.tablename" would not be supported.

2.4.2 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.

2.4.3 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.

2.4.4 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.

2.5 Mongo

Users can test MongoDB dependent 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"}
```

2.5.1 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]
```

(continues on next page)

(continued from previous page)

```
    assert returned == to_insert
```

2.6 Moto

Users can test AWS dependent code using the *create_moto_fixture*.

pytest_mock_resources.**create_moto_fixture**(**ordered_actions*, *region_name='us-east-1'*, *scope='function'*)

Produce a Moto fixture.

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

Note: Each test executes using a different (fake) AWS account through moto. If you create boto3 `client`/`resource` objects outside of the one handed to the test (for example, in the code under test), they should be sure to use the `aws_access_key_id`, `aws_secret_access_key`, `aws_session_token`, and `endpoint_url` given by the `<fixturename>.pmr_credentials` attribute.

Note: A moto dashboard should be available for debugging while the container is running. By default it would be available at `http://localhost:5555/moto-api/#` (but the exact URL may be different depending on your host/port config.

Parameters

- **ordered_actions** – Any number of ordered actions to be run on test setup.
- **region_name** (*str*) – The name of the AWS region to use, defaults to "us-east-1".
- **scope** (*str*) – The scope of the fixture can be specified by the user, defaults to "function".

Consider the following example:

```
# src/some_module.py

def list_files(s3_client):
    return s3_client.list_objects_v2(Bucket="x", Key="y")
```

A user could test this as follows:

```
# tests/some_test.py

from pytest_mock_resources import create_moto_fixture
from pytest_mock_resources.fixture.moto import Session

from some_module import list_files

moto = create_moto_fixture()

def test_list_files(moto: Session):
    s3_client = moto.client("s3")
    files = list_files(s3_client)
    assert ...
```

The test is handed a proxy-object which should functionally act like a *boto3.Session* object. Namely you would generally want to call *.client(...)* or *.resource(...)* on it.

Note: Each test executes using a different (fake) AWS account through moto. If you create boto3 `client`/`resource` objects using boto3 directly, outside of the object handed to your test, you should make sure to pass all the credentials fields into the constructor such that it targets the correct AWS instance/account.

For example:

```
import boto3
from pytest_mock_resources import create_moto_fixture
from pytest_mock_resources.fixture.moto import Session

moto = create_moto_fixture()

def test_list_files(moto: Session):
    kwargs = moto.pmr_credentials.as_kwargs()
    s3_client = boto3.client("s3", **kwargs)
```

Note: A moto dashboard should be available for debugging while the container is running. By default it would be available at `http://localhost:5555/moto-api/#` (but the exact URL may be different depending on your host/port config.

2.6.1 Actions

Similar to ordered "actions" in other fixtures, moto supports the static creation of certain kinds of objects ahead of the actual test execution as well.

For moto, this represents as physical infrastructure/configuration/objects within the moto "AWS account" being used by your test.

Per the note above, each test executes in a unique moto "AWS account". This means that two `create_moto_fixture` fixtures with different infrastructure will be completely distinct and not leak state (either between tests or between fixtures).

Note: we will absolutely accept feedback on additional kinds of supported objects to add, the current set is motivated by internal use, but is certainly not exhaustive.

S3

Currently: `S3Bucket`, `S3Object`

These objects help reduce boilerplate around setting up buckets/files among tests.

```
from pytest_mock_resources import create_moto_fixture, S3Bucket, S3Object
from pytest_mock_resources.fixture.moto import Session

bucket = S3Bucket("test")
moto = create_moto_fixture(
    S3Bucket("other_bucket"),
```

(continues on next page)

(continued from previous page)

```
    bucket,
    bucket.object("test.csv", "a,b,c\n1,2,3"),
)

 def test_ls(moto: Session):
     resource = moto.resource("s3")
     objects = resource.Bucket("test").objects.all()
     assert len(objects) == 1

     assert objects[0].key == "test.txt"
     assert objects[0].get()["Body"].read() == b"a,b,c\n1,2,3"
```

2.7 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 limit). 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()
```

(continues on next page)

(continued from previous page)

```
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"}
```

2.7.1 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"
```

CHAPTER

THREE

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"))
```

3.1 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

FOUR

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

4.1 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.

4.2 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')
```

4.3 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.

CHAPTER

FIVE

CLI

As you start writing tests, you might notice that there's a small delay after invoking the tests before they execute. Which can get particularly annoying if you're only running a small subset of your tests at a time. This delay is because 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 up to 30s! Even Postgres, has ~ 1-2 seconds startup time; which you'll pay each time you invoke `pytest`.

Pytest Mock Resources ships with a small CLI utility `pmr`, which can be used to help amortize the cost of container startup between test runs. With it, you can pre-create the container against which the tests will connect.

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:

```
$ pmr --stop postgres
711f5d5a86896bb4eb76813af4fb6616aee0eff817cdec6ebaf4daa0e9995441
```

CHAPTER

SIX

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.

6.1 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
```

6.1.1 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.

6.1.2 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.

6.2 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

SEVEN

API

7.1 Fixture Functions

class pytest_mock_resources.**StaticStatements**(**statements*)
A discriminator for statements which are safe to execute exactly once.

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*, *engine_kwargs=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.
- **engine_kwargs** – Optional set of kwargs to send into the engine on creation.

pytest_mock_resources.**create_postgres_fixture**(**ordered_actions*, *scope='function'*, *tables=None*, *session=None*, *async_=False*, *createdb_template='template1'*, *engine_kwargs=None*, *template_database=True*, *actions_share_transaction=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.
- **template_database** – Defaults to True. When True, amortizes the cost of performing database setup through *ordered_actions*, by performing them once into a postgres "template" database, then creating all subsequent per-test databases from that template.
- **actions_share_transaction** – When True, the transaction used by *ordered_actions* context will be the same as the one handed to the test function. This is required in order to support certain usages of *ordered_actions, such as the creation of temp tables through a `Statements* object. By default, this behavior is enabled for synchronous fixtures for backwards compatibility; and disabled by default for asynchronous fixtures (the way v2-style/async features work in SQLAlchemy can lead to bad default behavior).

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 limit). 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*, *template_database=True*, *actions_share_transaction=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 approximations 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.
- **template_database** – Defaults to True. When True, amortizes the cost of performing database setup through *ordered_actions*, by performing them once into a postgres "template" database, then creating all subsequent per-test databases from that template.
- **actions_share_transaction** – When True, the transaction used by *ordered_actions* context will be the same as the one handed to the test function. This is required in order to support certain usages of *ordered_actions, such as the creation of temp tables through a `Statements* object. By default, this behavior is enabled for synchronous fixtures for backwards compatibility; and disabled by default for asynchronous fixtures (the way v2-style/async features work in SQLAlchemy can lead to bad default behavior).

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.

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"`.
- **drivername** (*str*) – The sqlalchemy driver to use Defaults to `"postgresql+psycopg2"`.

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

CONTRIBUTING

8.1 Prerequisites

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

8.2 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
```

8.2.1 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.

8.2.2 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
```

8.3 Need help

Submit an issue!

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

C