asphalt-sqlalchemy

Release 3.1.4.post1

Jan 16, 2019

Contents

This Asphalt framework component provides connectivity to relational databases.

It is a wrapper for the SQLAlchemy library.

CHAPTER 1

Configuration

A typical SQLAlchemy configuration consists of a single database. At minimum, you only need a connection URL (see the SQLAlchemy documentation for how to construct one). Such a configuration would look something like this:

```yaml
components:
  sqlalchemy:
    url: "postgresql://user:password@10.0.0.8/mydatabase"
```

This will make the ORM session (`Session`) accessible as `ctx.sql` and the database engine (`Engine`) as `ctx.sql.bind`.

See also:

`asphalt.sqlalchemy.component.SQLAlchemyComponent.create_engine()`

See also:

`asphalt.sqlalchemy.component.SQLAlchemyComponent.create_sessionmaker()`

1.1 Setting session options

If you need to adjust the options used for creating new sessions, you can do so by specifying them in the `session` option:

```yaml
components:
  sqlalchemy:
    url: "sqlite:///:memory:"
    session:
      info:
        hello: world
```

1.2 Multiple databases

If you need to work with multiple databases, things get a little more complicated. You will need to define the engines with the `engines` option:

```
components:
  sqlalchemy:
    engines:
      db1:
        url: "postgresql:///mydatabase"
      db2:
        url: "sqlite:///mydb.sqlite"
```

This will make the two sessions available as `ctx.db1` and `ctx.db2` respectively.

CHAPTER 2

Using engines and sessions

SQLAlchemy, by its nature, operates in a blocking manner. That is, running a query against the database will block the event loop. This includes implicit queries triggered by accessing lazily loaded relationships and deferred columns.

While simple queries usually complete in a timely manner, it is often difficult to predict the performance of interactions with such blocking APIs. The database server might be slow or completely unreachable, in which case your whole application hangs during the query. Even when it does not, your queries might gradually get slower due to increasing amounts of data. For these reasons, it is recommended that you handle your SQLAlchemy interactions in worker threads. The `Context` class provides a few conveniences for this purpose.

See also:

Working with coroutines and threads

2.1 Transactions

In asphalt-sqlalchemy, database connections are made on demand when you request a connection, either via one of the methods in the `Context` class or by directly accessing the appropriate context attribute of the connection resource. The resulting connection begins a transaction that is automatically committed when the context is was created through is torn down, unless the context was ended by an unhandled exception. The connection is always closed in either case.

2.2 Working with core queries

If you're not familiar with SQLAlchemy's core functionality, you should take a look at the SQL Expression Language Tutorial first.

Here's how the above example would work using core queries:

```
async def handler(ctx):
    # Database queries can block the event loop, so run this in a thread pool
    async with ctx.threadpool():
        parent_id = ctx.sql.scalar(select([people.c.id]).where(name='Senior'))
```

(continues on next page)

(continued from previous page)

```
        ctx.sql.execute(people.insert().values(name='Junior'))

    # Commit happens automatically when the context is torn down
```

2.3 Working with the Object Relational Mapper (ORM)

If you're not familiar with SQLAlchemy's ORM, you should look through the Object Relational Tutorial first.

The previous example would look like this, rewritten for the ORM:

```
async def handler(ctx):
    async with ctx.threadpool():
        parent = ctx.sql.query(Person).filter_by(name='Senior').one()
        parent.children.append(Person(name='Junior'))
```

2.4 Loading data at application startup

It is unadvisable to use connection or session resources from a long lived context. This would unnecessarily tie up connection resources, and if the connection is used repeatedly, it may get stale data due to transaction isolation.

A better way is to create a throwaway child context when you need to load initial data for the application:

```
class ApplicationComponent(ContainerComponent):
    async def start(ctx):
        # ctx here is the root context
        async with Context(ctx) as subctx:
            self.employees = subctx.sql.query(Employee).all()
```

The connection and session will be automatically closed once the context manager block is exited.

CHAPTER 3

Using SQLAlchemy events with Asphalt

While asphalt-sqlalchemy does not provide support for Asphalt style events at this time, you can still listen to the native SQLAlchemy events. Limited support is provided for interacting with the context from **session events**. Every ORM session object will have its associated session object stored in its `info` dictionary (`ctx.sql.info['ctx'] is ctx`).

In order to add a listener that applies to every ORM session created in the future, you can add your listener in the `sessionmaker` which is published by the SQLAlchemy component as a resource:

```
from sqlalchemy.orm import sessionmaker
from sqlalchemy import event


def precommit_hook(session):
    ctx = session.info['ctx']
    # do something with the context


async def handler(ctx):
    session_factory = ctx.require_resource(sessionmaker)
    event.listen(session_factory, 'before_commit', precommit_hook)
```

CHAPTER 4

Testing with asphalt-sqlalchemy

Testing database using code usually involves one of two approaches: either you mock your database connections and return fake data, or you test against a real database engine. This document focuses on the latter approach and provides instructions for setting up your fixtures accordingly.

The basic idea is to have a session level fixture which creates and engine and then makes a single connection, through which all the database interaction will happen during the testing session. It should first remove any previously created tables and then create the tables from scratch, using the current metadata. This ensures that even if the testing session was interrupted previously, the correct set of tables are always properly created. Then, during every test a transaction is started and then rolled back after the tests, thus ensuring test isolation.

In order to force all database interactions to happen within the same transaction, the `sqlalchemy` component is passed the `Connection` instance created by the connection fixture as the `bind` option. This will override any `url` option passed to the component. When the session's `bind` is a `Connection` and not an `Engine`, it will not attempt to actually commit the transaction. However, special measures must be taken if the application code ever rolls back the transaction. Unlike `commit()`, a `rollback()` call from a session does end the underlying transaction. To counter that, a session listener must be set up which restarts the transaction immediately after a session rolls it back.

This technique is based on a chapter of SQLAlchemy documentation dealing with testing.

Note: Always test against the same kind of database(s) as you're deploying on! Otherwise you may see unwarranted errors, or worse, passing tests that should have failed.

4.1 Setting up the SQLAlchemy component and the database connection

This assumes the following:

- You are using py.test for testing
- You have the necessary testing dependencies installed (`pytest`, `pytest-asyncio`)

- You have a package `yourapp.models` and a declarative base class (`Base`) in it
- You have model class named `Person` in `yourapp.models`
- You have a test database accessible (not required with SQLite)
- You have a project subdirectory for tests (named `tests` here)

The following fixtures should go in the `conftest.py` file in your `tests` folder. They ensure that any changes made to the database are rolled back at the end of each test.

See the `tests/test_testing_recipe` module in the source code for a more complete example.

```
from contextlib import closing

import pytest
from asphalt.core import ContainerComponent, Context
from asphalt.sqlalchemy import clear_database
from sqlalchemy import create_engine, event
from sqlalchemy.orm import Session

from yourapp.component import ApplicationComponent
from yourapp.models import Base, Person


@pytest.fixture(scope='session')
def connection():
    # NOTE: SQLite requires the following argument:
    # connect_args=dict(check_same_thread=False)
    engine = create_engine('mysql://user:password@localhost/test')
    conn = engine.connect()

    # Remove existing tables and create new ones based on the current metadata
    clear_database(conn)
    Base.metadata.create_all(conn, checkfirst=False)

    yield conn

    conn.close()
    engine.dispose()


@pytest.fixture(scope='session', autouse=True)
def person(connection):
    # Add some base data to the database here (if necessary for your application)
    with closing(Session(connection, expire_on_commit=False)) as session:
        person = Person(name='Test person')
        session.add(person)
        session.commit()
        return person


@pytest.fixture(autouse=True)
def transaction(connection):
    def restart(session):
        # When any session rolls back its transaction, restart this one if it's the␣
↪one that
        # has been rolled back
        nonlocal tx
        if not connection.in_transaction():
```

(continues on next page)

(continued from previous page)

```
            tx = connection.begin()

    tx = connection.begin()
    event.listen(Session, 'after_rollback', restart)
    yield
    event.remove(Session, 'after_rollback', restart)
    tx.rollback()


@pytest.fixture
def root_context():
    with Context() as ctx:
        yield ctx


@pytest.fixture
async def root_component(connection, root_context):
    components = {
        'sqlalchemy': {'bind': connection, 'ready_callback': None}
    }
    component = ContainerComponent(components=components)
    await component.start(root_context)


@pytest.fixture
def dbsession(connection):
    # A database session for use by testing code
    with closing(Session(connection)) as session:
        yield session
```

CHAPTER 5

Tips and tricks

Here you will find help on performing some common tasks related to SQLAlchemy with Asphalt.

5.1 How to automatically print emitted SQL

This can be done two ways:

1. Add the `echo=True` option to the engine configuration options
2. In your application's configuration, add a logger for `sqlalchemy.engine.base`:

```
logging:
  ...
  loggers:
    root:
      handlers: [console]
      level: WARNING
    asphalt:
      level: INFO
    sqlalchemy.engine.base:
      level: INFO
```

5.2 Handling schema migrations

For schema migrations, it is best to use the Alembic tool, which is made by SQLAlchemy's author. An ideal place to put your Alembic migration code in the application is the `start()` method of your application component, after calling `await super().start(ctx)`, but **before** starting any services that might actually use the database. If this is not feasible, consider fishing the connection URL(s) out of `self.component_configs` dictionary and running the migration before calling the superclass `start()` method.

Assuming that you have an `alembic` directory in the same directory as the module containing the application component class, here's how you might do it:

```
import os

from alembic import command, config

class ApplicationComponent(ContainerComponent):
    async def start(self, ctx):
        await super().start(ctx)

        cfg = config.Config(os.path.dirname(__file__), 'alembic', 'alembic.ini')
        with ctx.sql.bind.begin() as connection:
            cfg.attributes['connection'] = connection
            command.upgrade(cfg, "head")
```

Notice the direct use of the engine here – it's okay as long as the connection created is short lived, as is guaranteed by doing `with engine.begin():`.

CHAPTER 6

Version history

This library adheres to Semantic Versioning.

3.1.4 (2019-01-16)

- Eliminated the possibility of `session.commit()` and `session.close()` being called concurrently

3.1.3 (2018-12-18)

- Implemented better mechanics for exception handling to make sure that `session.close()` is always executed

3.1.2 (2018-12-11)

- Shield `session.close()` from cancellation to ensure that the connection is returned to the pool

3.1.1 (2018-10-01)

- Execute `session.close()` in a worker thread, as it can potentially block the event loop thread

3.1.0 (2017-07-08)

- Allowed the `engine` argument to `clear_database()` to be any `Connectable`
- Added the `ready_callback` option to engine configuration (should be handy for creating tables or doing schema migrations)
- Restored the ability to pass a `Connection` as the `bind` configuration option
- Made the `poolclass` engine option passable as a `module:varname` reference (contributed by Devin Fee)
- Improved the testing recipe and added tests for it
- Added compatibility with Asphalt 4.0

3.0.1 (2017-05-06)

- Fixed `clear_database()` causing an SQLAlchemy error when `metadata.drop_all()` tries to drop constraints that `clear_database()` has already dropped
- Sped up `clear_database()` a bit by not checking for the presence of tables when dropping them right after reflecting the metadata

3.0.0 (2017-04-16)

- **BACKWARD INCOMPATIBLE** Migrated to Asphalt 3.0
- **BACKWARD INCOMPATIBLE** Engine resources are no longer directly accessible as context variables. Instead, every engine gets its own session and can be accessed via the session's `bind` variable.
- **BACKWARD INCOMPATIBLE** The component now longer accepts bare `Connection` objects to be added as resources
- **BACKWARD INCOMPATIBLE** The commit executor is now configured on the component level
- An explicit commit executor is now always used (a new one will be created implicitly if none is defined in the configuration)
- **BACKWARD INCOMPATIBLE** Session configuration can no longer be disabled (no more `session=False`)
- **BACKWARD INCOMPATIBLE** Engines can no longer be bound to `MetaData` objects
- **BACKWARD INCOMPATIBLE** Renamed the `asphalt.sqlalchemy.util` module to `asphalt.sqlalchemy.utils`
- **BACKWARD INCOMPATIBLE** The `connect_test_database()` function in the `util` module was replaced with the `clear_database()` which has somewhat different semantics

2.1.3 (2017-02-11)

- A better fix for the memory leak plugged in v2.1.2.

2.1.2 (2017-02-11)

- Fixed a memory leak that was triggered by using the context's SQLAlchemy session

2.1.1 (2016-12-19)

- Modified session finalization code to work around a suspected Python bug

2.1.0 (2016-12-12)

- Added the `commit_executor` option that lets users specify which executor to use for running automatic commit() on context finish

2.0.0 (2016-05-09)

- **BACKWARD INCOMPATIBLE** Migrated to Asphalt 2.0
- **BACKWARD INCOMPATIBLE** Removed all asynchronous API extensions (`asphalt.sqlalchemy.async`)
- **BACKWARD INCOMPATIBLE** Renamed `asphalt.sqlalchemy.utils` to `asphalt.sqlalchemy.util` to be consistent with the core library
- Allowed combining `engines` with default parameters

1.0.0 (2016-01-06)

- Initial release
- API reference