falcon-sqla

Release 0.4.0

Vytautas Liuolia et al.

Jan 27, 2023

PACKAGE DOCUMENTATION

The `falcon-sqla` package provides a middleware component for managing SQLAlchemy sessions. The manager component can also serve as a base building block or a recipe for more complex use cases, such as applications leveraging multiple database binds.

CHAPTER

ONE

INSTALLATION

Simply install the falcon-sqla package from PyPi:

```
$ pip install falcon-sqla
```

For more installation options, see *Installation*.

CHAPTER

TWO

USAGE

Configuring the *falcon_sqla.Manager* middleware in a falcon.App:

```
engine = create_engine('dialect+driver://my/database')
manager = falcon_sqla.Manager(engine)

app = falcon.App(middleware=[manager.middleware])

# The database session will be available as req.context.session
```

More usage scenarios are covered in the *User Guide*.

2.1 Installation

2.1.1 PyPi

To obtain the stable version, simply install the falcon-sqla package from PyPi:

```
$ pip install falcon-sqla
```

Note: `falcon-sqla` requires the following environment to run:

- Python 3.7 or newer; CPython and PyPy are supported.
- Falcon 2.0.0 or newer (3.1+ is recommended).
- SQLAlchemy 1.3.0 or newer (2.0+ is recommended).

2.1.2 GitHub

To install the latest commit directly from GitHub:

```
$ pip install git+https://github.com/vytas7/falcon-sqla
```

2.2 User Guide

The `falcon_sqla` session *Manager* can be used in two ways:

- As a Falcon middleware component.
- As a context manager to explicitly provide a database session.

2.2.1 Configuration

- Create a SQLAlchemy engine.
- Pass the engine to the *Manager()* initializer as its first parameter.
- If using the manager as a middleware component, pass its *middleware* property to a falcon.App's middleware list:

```
engine = create_engine('dialect+driver://my/database')
manager = falcon_sqla.Manager(engine)

app = falcon.App(middleware=[manager.middleware])

# The database session will be available as req.context.session
```

2.2.2 Context Manager

A *falcon_sqla.Manager* can also explicitly provide a database session using the *session_scope()* context manager:

```
# Somewhere inside a responder
with self.manager.session_scope(req, resp) as session:
    # Use the session
    # <...>
```

session_scope() can also be used as a standalone session context outside of the request-response cycle:

```
with self.manager.session_scope() as session:
    # Use the session
    # <...>
```

2.2.3 Custom Vertical Partitioning

Simple random selection of read- and write- database replicas is supported out of the box. Use the *add_engine()* method to instruct the *Manager* to include the provided engines in the runtime bind selection logic:

```
manager = falcon_sqla.Manager(engine)

read_replica = create_engine('dialect+driver://my/database.replica')
manager.add_engine(read_replica, 'r')
```

The *Manager.get_bind()* method can be overridden to implement custom engine selection logic for more complex use cases.

See also this SQLAlchemy recipe: Custom Vertical Partitioning.

2.3 Module Reference

2.3.1 Session Manager

class falcon_sqla.**Manager**(*engine*, *session_cls=<class 'falcon_sqla.session.RequestSession'>*, *binds=None*)

A manager for SQLAlchemy sessions.

This manager allows registering multiple SQLAlchemy engines, specifying if they are read-write or read-only or write-only capable.

Parameters

- **engine** (*Engine*) – An instance of a SQLAlchemy Engine, usually obtained with its `create_engine` function. This engine is added as read-write.
- **session_cls** (*type, optional*) – Session class used by this engine to create the session. Should be a subclass of SQLAlchemy `Session` class. Defaults to *`RequestSession`*.
- **binds** (*dict, optional*) – A dictionary that allows specifying custom binds on a per-entity basis in the session. See also https://docs.sqlalchemy.org/en/13/orm/session_api.html#sqlalchemy.orm.session.Session.params.binds. Defaults to `None`.

add_engine(*engine*, *role='r'*)

Adds a new engine with the specified role.

Parameters

- **engine** (*Engine*) – An instance of a SQLAlchemy Engine.
- **role** (*{'r', 'rw', 'w'}, optional*) – The role of the engine ('r': read-ony, 'rw': read-write, 'w': write-only). Defaults to 'r'.

get_bind(*req*, *resp*, *session*, *mapper*, *clause*)

Choose the appropriate bind for the given request session.

This method is not used directly, it's called by the session instance if multiple engines are defined.

get_session(*req=None*, *resp=None*)

Returns a new session object.

property middleware

Create a new *`Middleware`* instance connected to this manager.

property read_engines

A tuple of read capable engines.

session_scope(*req=None*, *resp=None*)

Provide a transactional scope around a series of operations.

Based on the `session_scope()` recipe from https://docs.sqlalchemy.org/orm/session_basics.html.

property write_engines

A tuple of write capable engines.

2.3.2 Session Management Options

class falcon_sqla.manager.**SessionOptions**

Defines a set of configurable options for the session.

An instance of this class is exposed via `Manager.session_options`.

no_session_methods

HTTP methods that by default do not require a DB session. Defaults to *SessionOptions.NO_SESSION_METHODS*.

Type
frozenset

safe_methods

HTTP methods that can use a read-only engine since they do no alter the state of the db. Defaults to *SessionOptions.SAFE_METHODS*.

Type
frozenset

read_from_rw_engines

When True read operations are allowed from read-write engines. Only used if more than one engine is defined in the *Manager*. Defaults to `True`.

Type
bool

write_to_rw_engines

When True write operations are allowed from read-write engines. Only used if more than one engine is defined in the *Manager*. Defaults to `True`.

Type
bool

write_engine_if_flushing

When True a write engine is selected if the session is in flushing state. Only used if more than one engine is defined in the *Manager*. Defaults to `True`.

Type
bool

sticky_binds

When `True`, the same engine will be used for each database operation for the same request. When `False`, the engine will be chosen randomly from the ones with the required capabilities. Only used if more than one engine is defined in the *Manager*. Defaults to `False`.

Type
bool

request_id_func

A callable object that returns an unique id for to each session. The returned object must be hashable. Only used when *SessionOptions.sticky_binds* is `True`. Defaults to `uuid.uuid4`.

Type
callable

wrap_response_stream

When `True` (default), and the response stream is set, it is wrapped with an instance *`ClosingStreamWrapper`* in order to postpone SQLAlchemy session commit & cleanup after the response has finished streaming.

Type
bool

NO_SESSION_METHODS = frozenset({'OPTIONS', 'TRACE'})

HTTP methods that by default do not require a DB session.

SAFE_METHODS = frozenset({'GET', 'HEAD', 'OPTIONS', 'TRACE'})

HTTP methods that do not alter the server state. These methods are assumed to be fine with read-only replica engines.

2.3.3 Session Management Middleware

class falcon_sqla.middleware.**Middleware**(*manager*)

Falcon middleware that can be used with the session manager.

Parameters
manager (`Manager`) – Manager instance to use in this middleware.

process_request(*req*, *resp*)

Set up a SQLAlchemy session for this request.

The session object is stored as `req.context.session`.

When the *`sticky_binds`* option is set to `True`, a `req.context.request_id` identifier is created (if not already present) by calling the *`request_id_func`* function.

process_response(*req*, *resp*, *resource*, *req_succeeded*)

Clean up the session, if one was provided.

This response hook finalizes the session by calling its `.commit()` if *req_succeeded* is `True`, and `.rollback()` otherwise. Finally, it will close the session.

2.3.4 Request Session

class falcon_sqla.session.**RequestSession**(**args*, ***kwargs*)

Custom session that is associated with a Falcon request.

The Falcon request and response objects are passed inside the session's `info` context as `'req'` and `'resp'` keys, respectively.

get_bind(*mapper=None*, *clause=None*)

Use the manager to get the appropriate bind when `_manager_get_bind` is defined. Otherwise, the default logic is used.

This method is called by SQLAlchemy.

identity_map: IdentityMap

A mapping of object identities to objects themselves.

Iterating through `Session.identity_map.values()` provides access to the full set of persistent objects (i.e., those that have row identity) currently in the session.

See also:

`identity_key()` - helper function to produce the keys used in this dictionary.

2.3.5 Miscellaneous Utilities

class `falcon_sqla.util.`**`ClosingStreamWrapper`**(*stream*, *close*)

Iterator that wraps a WSGI response iterable with support for close().

This class is used to wrap WSGI response streams to provide a side effect when the stream is closed.

If the provided response stream is file-like, i.e., it has a `read` attribute, that attribute is copied to the wrapped instance too.

Parameters

- **stream** (*object*) – Readable file-like stream object.
- **close** (*callable*) – A callable object that is called before the stream is closed.

PYTHON MODULE INDEX

f

INDEX