database.py

```python
# sAsync:
# An enhancement to the SQLAlchemy package that provides persistent
# item-value stores, arrays, and dictionaries, and an access broker for
# conveniently managing database access, table setup, and
# transactions. Everything can be run in an asynchronous fashion using
# the Twisted framework and its deferred processing capabilities.
#
# Copyright (C) 2006, 2015 by Edwin A. Suominen, http://edsuom.com/sAsync
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Asynchronous database transactions via C{SQLAlchemy} and
C{Twisted}. You will surely have a subclass of L{AccessBroker}.
"""

import inspect, logging
from contextlib import contextmanager

from twisted.internet import defer, reactor
from twisted.python.failure import Failure

import sqlalchemy as SA
from sqlalchemy import pool

import asynqueue
from asynqueue import iteration, threads

import errors, queue
from selex import SelectAndResultHolder

def transaction(self, func, *t_args, **t_kw):
    """
    Everything making up a transaction, and everything run in the
    thread, is contained within this little function,
    including of course a call to C{func}.
    """
    trans = self.connection.begin()
    if not hasattr(func, 'im_self'):
        t_args = (self,) + t_args
    try:
        result = func(*t_args, **t_kw)
    except Exception as e:
        trans.rollback()
        text = asynqueue.Info().setCall(
            func, t_args, t_kw).aboutException(exception=e)
        raise errors.TransactionError(text)
    else:
        trans.commit()
        return result

def nextFromRP(rp, N=1):
    """
    I{Transaction magic.}
    """
    try:
        if N == 1:
            result = rp.fetchone()
        else:
            result = rp.fetchmany(N)
    except:
        result = []
    if result:
        return result
    raise StopIteration

def isNested(self):
    """
    I{Transaction magic.}
    """
    firstCode = None
    frame = inspect.currentframe()
    while True:
        frame = frame.f_back
        if frame is None:
            result = False
            break
        if frame.f_code == transaction.func_code:
            result = True
            break
        # Check if nested inside first-transaction code if necessary
        if firstCode is None and hasattr(self, 'first'):
            firstCode = getattr(self.first, 'func_code', False)
        if firstCode and frame.f_code == firstCode:
            result = True
            break
    del frame
    del firstCode
    return result


def transact(f):
    """
    Transaction decorator.

    Use this function as a decorator to wrap the supplied method I{f}
    of L{AccessBroker} in a transaction that runs C{f(*args, **kw)} in
    its own transaction.
    Immediately returns a C{Deferred} that will eventually have its
    callback called with the result of the transaction. Inspired by
    and largely copied from Valentino Volonghi's C{makeTransactWith}
    code.

    You can add the following keyword options to your function call:

    @keyword niceness: Scheduling niceness, an integer between -20 and
      20, with lower numbers having higher scheduling priority as in
      UNIX C{nice} and C{renice}.

    @keyword doNext: Set C{True} to assign highest possible priority,
      even higher than with niceness = -20.

    @keyword doLast: Set C{True} to assign lower possible priority,
      even lower than with niceness = 20.

    @keyword consumer: Set this to a consumer object (must implement
      the L{twisted.interfaces.IConsumer} interface) and the
      L{SA.ResultProxy} will write its rows to it in Twisted
      fashion. The returned deferred will fire when all rows have been
      written.

    @keyword raw: Set C{True} to have the transaction result returned
      in its original form even if it's a RowProxy or other iterator.
    """
    def substituteFunction(self, *args, **kw):
        """
        Puts the original function in the synchronous task queue and
        returns a deferred to its result when it is eventually run.

        If the transaction resulted in a C{ResultProxy} object, the
        C{Deferred} fires with an L{asynqueue.iteration.Deferator},
        unless you've supplied an IConsumer with the I{consumer}
        keyword. Then the deferred result is an
        L{asynqueue.iteration.IterationProducer} coupled to your
        consumer.

        This function will be given the same name as the original
        function so that it can be asked to masquerade as the original
        function. As a result, the threaded call to the original
        function that it makes inside its C{transaction} sub-function
        will be able to use the arguments for that original
        function. (The caller will actually be calling this substitute
        function, but it won't know that.)

        The original function should be a method of a L{AccessBroker}
        subclass instance, and the queue for that instance will be
        used to run it.

        @see: U{/AsynQueue/asynqueue.iteration.html}
        """
        def oops(failureObj):
            # Encapsulate the failure object in a list to avoid the
            # errback chain until we are ready.
            return [failureObj]
        @defer.inlineCallbacks
        def doTransaction():
            if self.singleton or not self.running:
                # Not yet running, "wait" here for queue, engine, and
                # connection
                yield self.lock.acquire()
                if not self.singleton:
                    # If we can handle multiple connections (TODO), we
                    # don't want to hold onto the lock because
                    # transactions are queued in the ThreadQueue and
                    # additional connections can be obtained when doing
                    # ResultProxy iteration.
                    self.lock.release()
            result = yield self.q.call(
                transaction, self, f, *args, **kw).addErrback(oops)
            if not raw:
                result = yield self.handleResult(
                    result, consumer=consumer, asList=asList)
            if self.singleton:
                # If we can't handle multiple connections, we held
                # onto the lock throughout all of this
                self.lock.release()
            # If the result is a failure, raise its exception to trigger
            # the errback outside this function. Very weird, I
            # know. Basically, there must be an exception IN this function
            # to trigger the errback OUTSIDE of it. Just returning a
            # Failure doesn't seem to work.
            if isinstance(result, list):
                if len(result) == 1 and isinstance(result[0], Failure):
                    result[0].raiseException()
            defer.returnValue(result)

        # The 'raw' keyword is also used by the queue/worker, but
        # 'asList', 'isNested', and 'consumer' are not.
        raw = kw.get('raw', False)
        asList = kw.pop('asList', False)
        consumer = kw.pop('consumer', None)
        if consumer and raw:
            raise ValueError(
                "Can't supply a consumer for a raw transaction result")
        if kw.pop('isNested', False) or isNested(self):
            # The call and its result only get special treatment in
            # the outermost @transact function
            return f(self, *args, **kw)
        # Here's where the ThreadQueue actually runs the
        # transaction
        return doTransaction()

    # We need to ignore any transact decorator on an AccessBroker's
    # 'first' method, because it can't wait for the lock
    if f.func_name == 'first':
        return f
    substituteFunction.func_name = f.func_name
    return substituteFunction
```

```
class AccessBroker(object):
    """
    I manage asynchronous access to a database.

    Before you use any instance of me, you must specify the parameters
    for creating an SQLAlchemy database engine. A single argument is
    used, which specifies a connection to a database via an RFC-1738
    url. In addition, the I{verbose} keyword options can be employed
    to spew out log messages about calls and errors. (Use that only
    for debugging.)

    You can set an engine globally, for all instances of me via the
    L{sasync.engine} package-level function, or via my L{engine} class
    method. Alternatively, you can specify an engine for one
    particular instance by supplying the parameters to the
    constructor.

    I employ C{AsynQueue} to queue up transactions asynchronously and
    perform them one at a time. But I still want a connection pool for
    my database engine to handle asynchronous iteration of rows from
    query results. See the source for L{handleResult}.

    SQLAlchemy has excellent documentation, which describes the engine
    parameters in plenty of detail. See
    U{http://docs.sqlalchemy.org/en/rel_1_0/core/engines.html}.

    @ivar q: A property-generated reference to a threaded task queue
      that is dedicated to my database connection.

    @ivar connection: The current SQLAlchemy connection object, if
      any yet exists. Generated by my L{connect} method.

    @see: U{http://edsuom.com/AsynQueue/asynqueue.threads.ThreadQueue.html}
    """
    # A single class-wide queue factory
    qFactory = queue.Factory()

    @classmethod
    def setup(cls, url, **kw):
        """
        Constructs a global queue for all instances of me, returning a
        deferred that fires with it.
        """
        return cls.qFactory.setGlobal(url, **kw)

    def __init__(self, *args, **kw):
        """
        Constructs an instance of me, optionally specifying parameters for
        an SQLAlchemy engine object that serves this instance only.
        """
        def firstAsTransaction():
            # Need to call the first transaction here rather than via
            # a regular 'transact' substitute call to avoid competing
            # for the lock
            with self.connection.begin():
                self.first()
        @defer.inlineCallbacks
        def startup(null):
            # Queue with attached engine, possibly shared with other
            # AccessBrokers
            self.q = yield self.qFactory(*args, **kw)
            # A connection of my very own
            self.connection = yield self.connect()
            # Pre-transaction startup, called in main loop after
            # connection made.
            yield defer.maybeDeferred(self.startup)
            # First transaction, called in thread
            yield self.q.deferToThread(firstAsTransaction)
            # Ready for regular transactions
            self.running = True
            self.lock.release()
            reactor.addSystemEventTrigger(
                'before', 'shutdown', self.shutdown)

        self.selects = {}
        self.rowProxies = []
        self.running = False
        # The deferred lock lets us easily wait until setup is done
        # and avoids running multiple transactions at once when they
        # aren't wanted.
        self.lock = asynqueue.DeferredLock()
        self.lock.acquire().addCallback(startup)

    @property
    def singleton(self):
        if not hasattr(self, '_singleton'):
            engine = getattr(getattr(self, 'q', None), 'engine', None)
            self._singleton = getattr(engine, 'name', 'sqlite') == 'sqlite'
        return self._singleton

    def connect(self):
        def nowConnect(null):
            return self.q.call(
                self.q.engine.contextual_connect)
        if not getattr(self, 'q', None):
            return self.waitUntilRunning().addCallback(nowConnect)
        return nowConnect(None)

    @defer.inlineCallbacks
    def waitUntilRunning(self):
        """
        Returns a C{Deferred} that fires when the broker is running and
        ready for transactions.
        """
        if not self.running:
            yield self.lock.acquire()
            self.lock.release()

    def callWhenRunning(self, f, *args, **kw):
        """
        Calls the I{f-args-kw} combo when the broker is running and ready
        for transactions.
        """
        return self.waitUntilRunning().addCallback(lambda _: f(*args, **kw))

    @defer.inlineCallbacks
    def table(self, name, *cols, **kw):
        """
        Instantiates a new table object, creating it in the transaction
        thread as needed.

        One or more indexes other than the primary key can be defined
        via a keyword prefixed with I{index_} or I{unique_} and having
        the index name as the suffix. Use the I{unique_} prefix if the
        index is to be a unique one. The value of the keyword is a
        list or tuple containing the names of all columns in the
        index.
        """
        def haveQueue():
            return getattr(self, 'q', None)

        def makeTable():
            if not hasattr(self, '_meta'):
                self._meta = SA.MetaData(self.q.engine)
            indexes = {}
            for key in kw.keys():
                if key.startswith('index_'):
                    unique = False
                elif key.startswith('unique_'):
                    unique = True
                else:
                    continue
                indexes[key] = kw.pop(key), unique
            kw.setdefault('useexisting', True)
            table = SA.Table(name, self._meta, *cols, **kw)
            table.create(checkfirst=True)
            setattr(self, name, table)
            return table, indexes

        def makeIndex(tableInfo):
            table, indexes = tableInfo
            for key, info in indexes.iteritems():
                kwIndex = {'unique':info[1]}
                try:
                    # This is stupid. Why can't I see if the index
                    # already exists and only create it if needed?
                    index = SA.Index(
                        key, *[
                            getattr(table.c, x) for x in info[0]], **kwIndex)
                    index.create()
                except:
                    pass

        if not hasattr(self, name):
            if not haveQueue():
                # This is tricky; startup hasn't finished, but making
                # a table is likely to be part of the startup. What we
                # really need to wait for is the presence of a queue.
                yield iteration.Delay(backoff=1.02).untilEvent(haveQueue)
            tableInfo = yield self.q.deferToThread(makeTable)
            yield self.q.deferToThread(makeIndex, tableInfo)

    def startup(self):
        """
        This method runs before the first transaction to start my
        synchronous task queue. B{Override it} to get whatever
        pre-transaction stuff you have run in the main loop before a
        database engine/connection is created.
        """
        return defer.succeed(None)

    def first(self):
        """
        This method automatically runs as the first transaction after
        completion of L{startup}. B{Override it} to define table
        contents or whatever else you want as a first transaction that
        immediately follows your pre-transaction stuff.

        You don't need to decorate the method with C{@transact}, but
        it doesn't break anything if you do.
        """

    @defer.inlineCallbacks
    def shutdown(self, *null):
        """
        Shuts down my database transaction functionality and threaded task
        queue, returning a deferred that fires when all queued tasks
        are done and the shutdown is complete.
        """
        def closeConnection():
            conn = getattr(self, 'connection', None)
            if conn is not None:
                if hasattr(conn, 'connection'):
                    # Close the raw DBAPI connection rather than a
                    # proxied one. Does this actually make any
                    # difference?
                    conn = conn.connection
                conn.close()

        if self.running:
            self.running = False
            if self.q.isRunning():
                with self.lock.context() as d:
                    yield d
                    # Calling this via the queue is a problem if the
                    # queue is shared and has been shut down. But
                    # calling it in the main thread seems to work
                    closeConnection()
                    # yield self.q.call(closeConnection)
            yield self.qFactory.kill(self.q)

    @defer.inlineCallbacks
    def handleResult(self, result, consumer=None, conn=None, asList=False, N=1):
        """
        Handles the result of a transaction or connection.execute. If it's
        a C{ResultsProxy} and possibly an implementor of C{IConsumer},
        returned a (deferred) instance of Deferator or couples your
        consumer to an IterationProducer.

        You can supply a connection to be closed after iterations are
        done with the keyword 'conn'. With the keyword I{asList}, you
        can force the rows of a C{ResultProxy} to be fetched (in my
        thread) and returned as a (deferred) list of rows.
        """
        def close(null):
            if callable(getattr(result, 'close', None)):
                result.close()
            if conn:
                conn.close()

        if getattr(result, 'returns_rows', False):
            # A ResultsProxy gets special handling
            if asList:
                # ...except with asList, when all its rows are fetched
                # in my thread and simply returned as a list
                result = yield self.q.deferToThread(result.fetchall)
            else:
                pf = iteration.Prefetcherator(repr(result))
                ok = yield pf.setup(
                    self.q.deferToThread, nextFromRP, result, N)
                if ok:
                    dr = iteration.Deferator(pf)
                    dr.addCallback(close)
                    if consumer:
                        # A consumer was supplied, so try to make an
                        # IterationProducer couple to it.
                        ip = iteration.IterationProducer(dr, consumer)
                        # We "wait" here for the iteration/production
                        # to finish. What actually happens is that the
                        # caller receives a deferred that fires when
                        # iteration/production is done. However, the
                        # consumer gets the iterations in the
                        # meantime.
                        yield ip.run()
                        result = consumer
                    else:
                        # No consumer supplied, just "return" the Deferator
                        result = dr
                else:
                    # Empty/invalid ResultsProxy, just "return" an empty list
                    result = []
        defer.returnValue(result)

    def s(self, *args, **kw):
        """
        Polymorphic method for working with C{select} instances within a
        cached selection subcontext.

          - When called with a single argument (the select object's name
            as a string) and no keywords, this method indicates if the
            named select object already exists and sets its selection
            subcontext to I{name}.

          - With multiple arguments or any keywords, the method acts
```

```
            like a call to C{sqlalchemy.select(...).compile()}, except
            that nothing is returned. Instead, the resulting select
            object is stored in the current selection subcontext.

          - With no arguments or keywords, the method returns the select
            object for the current selection subcontext.

        Call from inside a transaction.

        """
        if kw or (len(args) > 1):
            # It's a compilation.
            context = getattr(self, 'context', None)
            self.selects[context] = SA.select(*args, **kw).compile()
        elif len(args) == 1:
            # It's a lookup to see if the select has been previously
            # seen and compiled; return True or False.
            self.context = args[0]
            return self.context in self.selects
        else:
            # It's a retrieval of a compiled selection object, keyed off
            # the most recently mentioned context.
            context = getattr(self, 'context', None)
            return self.selects.get(context)

    def select(self, *args, **kw):
        """
        Just returns an C{SQLAlchemy} select object. You do everything
        else.

        This is an immediate result, not a C{Deferred}.
        """
        return SA.select(*args, **kw)

    @contextmanager
    def selex(self, *args, **kw):
        """
        Supply columns as arguments and this method generates a select on
        the columns, yielding a placeholder object with the same
        attributes as the select object itself.

        Supply a callable as an argument (along with any of its args)
        and it yields a placeholder whose attributes are the same as
        the result of that call.

        In either case, you do stuff with the placeholder and call it
        to execute the connection with it. Supply the name of a
        resultsproxy method (and any of its args) to the call to get
        the result instead of the rp. Do all of this within the
        context of the placeholder::

          with <me>.selex(<table>.c.foo) as sh:
              sh.where(<table>.c.bar == "correct")
          for dRow in sh():
              row = yield dRow
              ...
        You can call this outside of a transaction and get a deferred
        result from calling the placeholder object. For such usage,
        you can specify the usual transaction keywords via keywords to
        this method.
        """
        kw['isNested'] = isNested(self)
        sh = SelectAndResultHolder(self, *args, **kw)
        yield sh
        sh.close()

    @defer.inlineCallbacks
    def selectorator(self, selectObj, consumer=None, de=None, N=1):
        """
        When called with a select object that results in an iterable
        C{ResultProxy} when executed, returns a deferred that fires
        with a C{Deferator} that can be iterated over deferreds, each
        of which fires with a successive row of the select's
        C{ResultProxy}.

        If you supply an C{IConsumer} with the I{consumer} keyword, I
        will couple an C{iteration.IterationProducer} to your consumer
        and run it. The returned C{Deferred} will fire (with a
        reference to your consumer) when the iterations are done, but
        you don't need to wait for that before doing another
        transaction.

        If you supply a C{Deferred} via the I{de} keyword, it will be
        fired (unless already fired for some reason) with C{None} when
        the select object has been executed, but before iterations
        have begun.

        If you want multiple rows produced at once using the
        C{fetchmany} method of the C{ResultProxy}, set I{N} to the
        number of rows you want at a time.

        Call directly, *not* from inside a transaction::

          dr = yield <me>.selectorator(<select>)
          for d in dr:
              row = yield d
              <proceed with row>

          d = <me>.selectorator(<select>, <consumer>)
          <do other stuff>
          consumer = yield d
          consumer.revealMagnificentSummary()
        """
        yield self.waitUntilRunning()
        # A new connection just for this iteration, so that other
        # transactions can (hopefully) proceed while iteration is
        # happening.
        connection = yield self.connect()
        rp = yield self.q.call(connection.execute, selectObj)
        if isinstance(de, defer.Deferred) and not de.called:
            de.callback(None)
        result = yield self.handleResult(
            rp, consumer=consumer, conn=connection, N=N)
        defer.returnValue(result)

    @transact
    def execute(self, *args, **kw):
        """
        Does a C{connection.execute(*args, **kw)} as a transaction, in my
        thread with my current connection, with all the usual handling
        of the C{ResultProxy}.
        """
        return self.connection.execute(*args, **kw)

    def sql(self, sqlText, **kw):
        """
        Executes raw SQL as a transaction, in my thread with my current
        connection, with any rows of the result returned as a list.
        """
        kw['asList'] = True
        return self.execute(SA.text(sqlText), **kw)

    @defer.inlineCallbacks
    def produceRows(self, f, iterator, table, colName, **kw):
        """
        Calls the single-argument function I{f} repeatedly, with each
        value yielded from the supplied I{iterator}, to produce values
        for new rows in the specified I{table}. The named column
        I{colName} will have that value set for each row, in
        order. Other column will be set to values that are constant
        across the rows, as specified via keywords.

        The function I{f} must not block, but it may return either an
        immediate or deferred result. It doesn't matter if some calls
        to I{f} take longer than others; the rows will be written in
        the same order as the B{input} values to I{f} are yielded from
        I{iterator}.

        B{Warning:} Needs unit testing.

        @return: A C{Deferred} that fires with a list of the primary
          key values for each row, in the same order as I{iterator},
          when all the values have been generated and written.
        """
        def insert(i):
            pkList = []
            with self.connection.begin():
                for value in i:
                    kw[colName] = value
                    rp = table.insert().execute(**kw)
                    pkList.append(rp.inserted_primary_key)
            return pkList

        p = threads.OrderedItemProducer()
        yield p.start(insert)
        for x in iterator:
            p.produceItem(f, x)
        result = yield p.stop()
        defer.returnValue(result)
    def deferToQueue(self, func, *args, **kw):
        """
        Dispatches I{callable(*args, **kw)} as a task via the like-named
        method of my asynchronous queue, returning a C{Deferred} to
        its eventual result.

        Scheduling of the call is impacted by the I{niceness} keyword
        that can be included in I{**kw}. As with UNIX niceness, the
        value should be an integer where 0 is normal scheduling,
        negative numbers are higher priority, and positive numbers are
        lower priority.

        @keyword niceness: Scheduling niceness, an integer between -20
          and 20, with lower numbers having higher scheduling priority
          as in UNIX C{nice} and C{renice}.
        """
        return self.waitUntilRunning().addCallback(
            lambda _: self.q.call(func, *args, **kw))


__all__ = ['transact', 'AccessBroker', 'SA']
```

errors.py

```python
# sAsync:
# An enhancement to the SQLAlchemy package that provides persistent
# item-value stores, arrays, and dictionaries, and an access broker for
# conveniently managing database access, table setup, and
# transactions. Everything can be run in an asynchronous fashion using
# the Twisted framework and its deferred processing capabilities.
#
# Copyright (C) 2006, 2015 by Edwin A. Suominen, http://edsuom.com/sAsync
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Errors relating to database access
"""

class AsyncError(Exception):
    """
    The requested action is incompatible with asynchronous operations.
    """

class TransactionError(Exception):
    """
    An exception was raised while trying to run a transaction.
    """

class DatabaseError(Exception):
    """
    A problem occured when trying to access the database.
    """

class TransactIterationError(Exception):
    """
    An attempt to access a transaction result as an iterator
    """
```

items.py

```
# sAsync:
# An enhancement to the SQLAlchemy package that provides persistent
# item-value stores, arrays, and dictionaries, and an access broker for
# conveniently managing database access, table setup, and
# transactions. Everything can be run in an asynchronous fashion using
# the Twisted framework and its deferred processing capabilities.
#
# Copyright (C) 2006, 2015 by Edwin A. Suominen, http://edsuom.com/sAsync
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Dictionary-like objects with behind-the-scenes database persistence.

L{Items} provides a public interface for non-blocking database access
to persistently stored name:value items within a uniquely-identified
group, e.g., for a persistent dictionary using L{pdict.PersistentDict}.
"""

# Imports
from twisted.internet import defer
import sqlalchemy as SA

from database import transact, AccessBroker


NICENESS_WRITE = 6


class Missing:
    """
    An instance of me is returned as the value of a missing item.
    """
    def __init__(self, group, name):
        self.group, self.name = group, name


class Transactor(AccessBroker):
    """
    I do the hands-on work of non-blocking database access for the
    persistence of C{name:value} items within a uniquely-identified
    group, e.g., for a persistent dictionary using
    L{pdict.PersistentDict}.

    My methods return Twisted C{Deferred} instances to the results of
    their database accesses rather than forcing the client code to
    block while the database access is being completed.
    """
    def __init__(self, ID, *url, **kw):
        """
        Instantiates me for the items of a particular group uniquely identified
        by the supplied integer I{ID}, optionally using a particular database
        connection to I{url} with any supplied keywords.
        """
        if not isinstance(ID, int):
            raise TypeError("Item IDs must be integers")
        self.groupID = ID
        if url:
            AccessBroker.__init__(self, url[0], **kw)
        else:
            AccessBroker.__init__(self)

    def startup(self):
        """
        Startup method, automatically called before the first transaction.
        """
        return self.table(
            'sasync_items',
            SA.Column('group_id', SA.Integer, primary_key=True),
            SA.Column('name', SA.String(40), primary_key=True),
            SA.Column('value', SA.PickleType, nullable=False)
            )

    @transact
    def load(self, name):
        """
        Item load transaction
        """
        items = self.sasync_items
        if not self.s('load'):
            self.s(
                [items.c.value],
                SA.and_(items.c.group_id == self.groupID,
                        items.c.name == SA.bindparam('name')))
        row = self.s().execute(name=name).fetchone()
        if row:
            return row['value']
        return Missing(self.groupID, name)

    @transact
    def loadAll(self):
        """
        Load all my items, returing a C{name:value} dict
        """
        items = self.sasync_items
        if not self.s('load_all'):
            self.s(
                [items.c.name, items.c.value],
                items.c.group_id == self.groupID)
        rows = self.s().execute().fetchall()
        result = {}
        for row in rows:
            result[row['name']] = row['value']
        return result

    @transact
    def update(self, name, value):
        """
        Item overwrite (entry update) transaction
        """
        items = self.sasync_items
        u = items.update(
            SA.and_(items.c.group_id == self.groupID,
                    items.c.name == name))
        u.execute(value=value)

    @transact
    def insert(self, name, value):
        """
        Item add (entry insert) transaction
        """
        self.sasync_items.insert().execute(
            group_id=self.groupID, name=name, value=value)

    @transact
    def delete(self, *names):
        """
        Item(s) delete transaction
        """
        items = self.sasync_items
        self.sasync_items.delete(
            SA.and_(items.c.group_id == self.groupID,
                    items.c.name.in_(names))).execute()

    @transact
    def names(self):
        """
        All item names loading transaction
        """
        items = self.sasync_items
        if not self.s('names'):
            self.s(
                [items.c.name],
                items.c.group_id == self.groupID)
        return [str(x[0]) for x in self.s().execute().fetchall()]


class Items(object):
    """
    I provide a public interface for non-blocking database access to
    persistently stored name:value items within a uniquely identified
    group, e.g., for a persistent dictionary using
    L{pdict.PersistentDict}.

    Before you use any instance of me, you must specify the parameters
    for creating an C{SQLAlchemy} database engine. A single argument
    is used, which specifies a connection to a database via an
    RFC-1738 url. In addition, the following keyword options can be
    employed, which are listed in the API docs for L{sasync} and
    L{database.AccessBroker}.

    You can set an engine globally, for all instances of me via the
    L{sasync.engine} package-level function, or via the
    L{AccessBroker.engine} class method. Alternatively, you can
    specify an engine for one particular instance by supplying the
    parameters to my constructor.
    """
    def __init__(self, ID, *url, **kw):
        """
        Instantiates me for the items of a particular group uniquely
        identified by the supplied hashable I{ID}.

        In addition to any engine-specifying keywords supplied, the following
        are particular to this constructor:

        @param ID: A hashable object that is used as my unique identifier.

        @keyword nameType: A C{type} object defining the type that each name
            will be coerced to after being loaded as a string from the
            database.
        """
        try:
            self.groupID = hash(ID)
        except:
            raise TypeError("Item IDs must be hashable")
        self.nameType = kw.pop('nameType', str)
        if url:
            self.t = Transactor(self.groupID, url[0], **kw)
        else:
            self.t = Transactor(self.groupID)
        for name in ('waitUntilRunning', 'callWhenRunning', 'shutdown'):
            setattr(self, name, getattr(self.t, name))

    def write(self, funcName, name, value, niceness=None):
        """
        Performs a database write transaction, returning a deferred to its
        completion.
        """
        func = getattr(self.t, funcName)
        if niceness is None:
            niceness = NICENESS_WRITE
        return self.callWhenRunning(func, name, value, niceness=niceness)

    def load(self, name):
        """
        Loads item I{name} from the database, returning a deferred to the
        loaded value. A L{Missing} object represents the value of a missing
        item.
        """
        return self.callWhenRunning(self.t.load, name)

    @defer.inlineCallbacks
    def loadAll(self):
        """
        Loads all items in my group from the database, returning a
        deferred to a dict of the loaded values. The keys of the dict
        are coerced to the type of my I{nameType} attribute.
        """
        newDict = {}
        yield self.waitUntilRunning()
        valueDict = yield self.t.loadAll()
        for name, value in valueDict.iteritems():
            key = self.nameType(name)
            newDict[key] = value
        defer.returnValue(newDict)

    def update(self, name, value):
        """
        Updates the database entry for item I{name} = I{value}, returning a
        deferred that fires when the transaction is done.
        """
        return self.write('update', name, value)

    def insert(self, name, value):
        """
        Inserts a database entry for item I{name} = I{value}, returning a
        deferred that fires when the transaction is done.
        """
        return self.write('insert', name, value)

    def delete(self, *names):
        """
        Deletes the database entries for the items having the supplied
        I{*names}, returning a deferred that fires when the transaction is
        done.
        """
        return self.t.delete(*names)

    def names(self):
        """
        Returns a deferred that fires with a list of the names of all items
        currently defined in my group.
        """
        def gotNames(names):
            return [self.nameType(x) for x in names]

        d = self.t.names()
        d.addCallback(gotNames)
        return d


__all__ = ['Missing', 'Items']
```

parray.py

```
# sAsync:
# An enhancement to the SQLAlchemy package that provides persistent
# item-value stores, arrays, and dictionaries, and an access broker for
# conveniently managing database access, table setup, and
# transactions. Everything can be run in an asynchronous fashion using
# the Twisted framework and its deferred processing capabilities.
#
# Copyright (C) 2006, 2015 by Edwin A. Suominen, http://edsuom.com/sAsync
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Persistent Three-dimensional array objects.
"""

# Imports
from twisted.internet import defer, reactor
import sqlalchemy as SA

from asynqueue import DeferredTracker
from database import transact, AccessBroker


NICENESS_WRITE = 6


class Transactor(AccessBroker):
    """
    I do the hands-on work of (potentially) non-blocking database access for
    the persistence of array elements within a uniquely-identified group.

    My methods return Twisted deferred instances to the results of their
    database accesses rather than forcing the client code to block while the
    database access is being completed.

    """
    def __init__(self, ID, *url, **kw):
        """
        Instantiates me for a three-dimensional array of elements within a
        particular group uniquely identified by the supplied integer I{ID},
        using a database connection to I{url}.
        """
        if not isinstance(ID, int):
            raise TypeError("Item IDs must be integers")
        self.groupID = ID
        self.dt = DeferredTracker()
        super(Transactor, self).__init__(*url[:1], **kw)

    def startup(self):
        """
        You can run my transaction methods when the deferred returned from
        this method fires, and not before.
        """
        return self.table(
            'sasync_array',
            SA.Column('group_id', SA.Integer),
            SA.Column('x', SA.Integer),
            SA.Column('y', SA.Integer),
            SA.Column('z', SA.Integer),
            SA.Column('value', SA.PickleType, nullable=False),
            unique_elements=['group_id', 'x', 'y', 'z']
            )

    @transact
    def load(self, x, y, z):
        """
        Element load transaction
        """
        array = self.sasync_array
        if not self.s('load'):
            self.s(
                [array.c.value],
                SA.and_(array.c.group_id == self.groupID,
                        array.c.x == SA.bindparam('x'),
                        array.c.y == SA.bindparam('y'),
                        array.c.z == SA.bindparam('z'))
                )
        rows = self.s().execute(x=hash(x), y=hash(y), z=hash(z)).first()
        return rows['value'] if rows else None

    @transact
    def update(self, x, y, z, value):
        """
        Element overwrite (entry update) transaction
        """
        elements = self.sasync_array
        u = elements.update(
            SA.and_(elements.c.group_id == self.groupID,
                    elements.c.x == hash(x),
                    elements.c.y == hash(y),
                    elements.c.z == hash(z))
            )
        u.execute(value=value)

    @transact
    def insert(self, x, y, z, value):
        """
        Element add (entry insert) transaction
        """
        self.sasync_array.insert().execute(
            group_id=self.groupID,
            x=hash(x), y=hash(y), z=hash(z), value=value)

    @transact
    def delete(self, x, y, z):
        """
        Element delete transaction
        """
        elements = self.sasync_array
        self.sasync_array.delete(
            SA.and_(elements.c.group_id == self.groupID,
                    elements.c.x == hash(x),
                    elements.c.y == hash(y),
                    elements.c.z == hash(z))
            ).execute()

    @transact
    def clear(self):
        """
        Transaction to clear all elements (B{Use with care!})
        """
        elements = self.sasync_array
        self.sasync_array.delete(
            elements.c.group_id == self.groupID).execute()


class PersistentArray(object):
    """
    I am a three-dimensional array of Python objects, addressable by any
    three-way combination of hashable Python objects. You can use me as a
    two-dimensional array by simply using some constant, e.g., C{None} when
    supplying an address for my third dimension.
    """
    search = None

    def __init__(self, ID, *url, **kw):
        """
        Constructor, with a URL and any engine-specifying keywords
        supplied if a particular engine is to be used for this
        instance. The following additional keyword is particular to
        this constructor:

        """
        try:
            self.ID = hash(ID)
        except:
            raise TypeError("Item IDs must be hashable")
        self.dt = DeferredTracker()
        self.t = Transactor(self.ID, *url[:1], **kw)
        for name in ('waitUntilRunning', 'callWhenRunning', 'shutdown'):
            setattr(self, name, getattr(self.t, name))
        self.dt.put(self.waitUntilRunning())

    def write(self, funcName, *args, **kw):
        """
        Performs a database write transaction, returning a deferred to its
        completion.
        """
        func = getattr(self.t, funcName)
        kw = {'niceness':kw.get('niceness', NICENESS_WRITE)}
        return self.callWhenRunning(func, *args, **kw)

    def get(self, x, y, z):
        """
        Retrieves an element (x,y,z) from the database.
        """
        return self.dt.deferToAll().addCallback(
            lambda _: self.t.load(x, y, z))

    def set(self, x, y, z, value):
        """
        Persists the supplied I{value} of element (x,y,z) to the database,
        inserting or updating a row as appropriate.
        """
        def loaded(loadedValue):
            if loadedValue is None:
                return self.write("insert", x, y, z, value)
            return self.write("update", x, y, z, value)

        d = self.callWhenRunning(self.t.load, x, y, z).addCallback(loaded)
        self.dt.put(d)
        return d

    def delete(self, x, y, z):
        """
        Deletes the database row for element (x,y,z).
        """
        d = self.write("delete", x, y, z)
        self.dt.put(d)
        return d

    def clear(self):
        """
        Deletes the entire group of database rows for B{all} of my
        elements. B{Use with care!}
        """
        d =self.write("clear", niceness=0)
        self.dt.put(d)
        return d


__all__ = ['PersistentArray']
```

pdict.py

```python
# sAsync:
# An enhancement to the SQLAlchemy package that provides persistent
# item-value stores, arrays, and dictionaries, and an access broker for
# conveniently managing database access, table setup, and
# transactions. Everything can be run in an asynchronous fashion using
# the Twisted framework and its deferred processing capabilities.
#
# Copyright (C) 2006, 2015 by Edwin A. Suominen, http://edsuom.com/sAsync
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Dictionary-like objects with behind-the-scenes database persistence.

B{Caution:} This module has a few test failures and may need some
debugging.
"""

from collections import MutableMapping

from twisted.internet import defer

from asynqueue import DeferredTracker

import items
from errors import AsyncError


class PersistentDictBase(MutableMapping, object):
    """
    I am a base class for a database-persistent dictionary-like object
    uniquely identified by the hashable constructor argument I{ID}.

    Before you use any instance of me, you must specify the parameters
    for creating an SQLAlchemy database engine. A single argument is
    used, which specifies a connection to a database via an RFC-1738
    url. In addition, the following keyword options can be employed,
    which are listed in the API docs for L{sasync} and
    L{sasync.database.AccessBroker}.
    You can set an engine globally, for all instances of me, via the
    L{sasync.engine} package-level function, or via the
    L{AccessBroker.engine} class method. Alternatively, you can
    specify an engine for one particular instance by supplying the
    parameters to my constructor.

    In my default mode of operation, both read and write item accesses
    occur asynchronously and return deferreds. However, you can put me
    into B{load} mode by calling my L{preload} method. At that point,
    all my items will be accessed synchronously as with any other
    dictionary. No other deferreds will be returned from any item
    access. Lazy writing will still be done, but behind the scenes and
    with no API access to write completions.

    B{IMPORTANT}: As with all C{sAsync} data store objects, make sure
    you call my L{shutdown} method for an instance of me that you're
    done with before allowing that instance to be deleted.

    @ivar isPreloadMode: Boolean flag that indicates if I am operating in
        preload mode.
    """
    def __init__(self, ID, *url, **kw):
        """
        Instantiates me with an item store keyed to the supplied hashable
        I{ID}.

        In addition to any engine-specifying keywords supplied, the following
        are particular to this constructor:

        @param ID: A hashable object that is used as my unique identifier.

        """
        try:
            self.ID = hash(ID)
        except:
            raise TypeError("Item IDs must be hashable")
        # In-memory Caches
        self.data, self.keyCache = {}, {}
        # For tracking lazy writes
        self.writeTracker = DeferredTracker()
        # My very own persistent items store
        if url:
            self.i = items.Items(self.ID, url[0], **kw)
        else:
            self.i = items.Items(self.ID)
        self.isPreloadMode = False

    def waitUntilRunning(self):
        return self.i.waitUntilRunning()

    def preload(self):
        """
        This method preloads all my items from the database (which may take a
        while), returning a C{Deferred} that fires when everything's ready and
        I've completed the transition into B{preload} mode.
        """
        d = self.loadAll()
        d.addCallback(lambda _: setattr(self, 'isPreloadMode', True))
        return d

    def shutdown(self, *null):
        """
        Shuts down my database L{AccessBroker} and its synchronous task
        queue.
        """
        return self.writeTracker.deferToAll().addCallback(
            lambda _: self.i.shutdown())

    @defer.inlineCallbacks
    def loadAll(self, *null):
        """
        Loads all items from the database, setting my in-memory dict and key
        cache accordingly.
        """
        yield self.deferToWrites()
        items = yield self.i.loadAll()
        self.data.clear()
        self.data.update(items)
        self.keyCache = dict.fromkeys(items.keys(), True)
        defer.returnValue(self.data)

    def deferToWrites(self, lastOnly=False):
        """
        @see: L{DeferredTracker.deferToAll}
        """
        if lastOnly:
            return self.writeTracker.deferToLast()
        return self.writeTracker.deferToAll()


class PersistentDict(PersistentDictBase):
    """
    I am a database-persistent dictionary-like object with memory caching of
    items and lazy writing.

    Getting, setting, or deleting my items returns C{Deferred} objects
    of the Twisted asynchronous framework that fire when the
    underlying database accesses are completed. Returning a deferred
    value avoids forcing the client code to block while the real value
    is being read from the database.

    @ivar data: The in-memory dictionary that each instance of me uses
      to cache values for a given ID.
    """

    #--- Core dict operations -------------------------------------------------

    def __getitem__(self, name):
        """
        Returns a C{Deferred} to the value of item I{name} or the value itself
        if in preload mode.
        The value is only loaded from the database if it isn't already in the
        in-memory dictionary.
        """
        def valueLoaded(value):
            if isinstance(value, items.Missing):
                raise KeyError(
                    "No item '%s' in the database" % name)
            self.data[name] = value
            self.keyCache.setdefault(name, False)
            return value

        if name in self.data:
            value = self.data[name]
            if self.isPreloadMode:
                return value
            else:
                return defer.succeed(value)
        elif self.isPreloadMode:
            raise KeyError(
                "No item '%s' in the database" % name)
        else:
            return self.i.load(name).addCallback(valueLoaded)

    def __setitem__(self, name, value):
        """
        Sets item I{name} to I{value}, saving it to the database if there
        isn't already an in-memory dictionary item with that exact value.
        """
        def valueLoaded(loadedValue):
            if isinstance(loadedValue, items.Missing):
                # Item isn't in the database, so insert it
                d = self.i.insert(name, value)
            else:
                # Update current value of item in the database
                d = self.i.update(name, value)
            d.addCallback(done)
            return d
        def done(x):
            d2.callback(None)
            return x

        oldValue = self.data.get(name, None)
        self.data[name] = value
        self.keyCache.setdefault(name, False)
        # Everything from here on is just lazy writing
        if oldValue is None:
            # We're writing an item that hasn't been loaded from the database
            # yet
            if self.isPreloadMode:
                # If it hasn't been loaded yet, in preload mode, it ain't there
                d = self.i.insert(name, value)
            else:
                # Not in preload mode, so it may be in the database
                # but not yet loaded. Note that we need to also track
                # a Deferred for whichever transaction is done next.
                d2 = defer.Deferred()
                self.writeTracker.put(d2)
                d = self.i.load(name).addCallback(valueLoaded)
        else:
            # There's already a value in the in-memory dictionary, update it
            d = self.i.update(name, value)
        self.writeTracker.put(d)

    def __delitem__(self, name):
        """
        Deletes item I{name}, removing its entry from both the in-memory
        dictionary and the database
        """
        if name in self.data:
            del self.data[name]
            self.keyCache.pop(name, None)
            d = self.i.delete(name)
            self.writeTracker.put(d)
        else:
            raise KeyError(name)

    def __contains__(self, key):
        """
        Indicates if I contain item I{key}.

        In I{preload} mode, returns C{True} if the item is present in my
        in-memory dictionary and C{False} if not.

        In normal mode, returns an immediate C{Deferred} firing with C{True}
        without a transaction if the item is already present in my in-memory
        dictionary. If it isn't, tries to load the item (it will probably be
        requested soon anyhow) and returns a C{Deferred} that will ultimately
        fire with C{True} unless the load resulted in a L{Missing} object. In
        that case, deletes the loaded C{Missing} object from my in-memory
        dictionary and fires the deferred with C{False}.

        Using the C{<key> in <dict>} Python construct doesn't seem to work in
        normal mode. Use L{has_key} instead.
        """
        if self.isPreloadMode:
            return self.data.__contains__(key)
        elif key in self.data or key in self.keyCache:
            return defer.succeed(True)
        else:
            d = self.i.load(key)
            d.addCallback(lambda value: not isinstance(value, items.Missing))
            return d

    def keys(self):
        """
        Returns an immediate or deferred list of the names of all my items in
        the database.
        """
        def gotKeyList(keyList):
            self.keyCache = dict.fromkeys(keyList, True)
            return keyList
        if self.isPreloadMode:
            return self.data.keys()
        if True in self.keyCache.values():
            # The key cache is valid as long as it has entries (=True)
            # that were retrieved from preloading or a previous call
            # of this method. The __setitem__ method will add new keys
            # to the cache, but that doesn't initialize it.
            keys = self.keyCache.keys()
            return defer.succeed(keys)
        # Empty or invalid key cache, load and cache a list of keys
        return self.i.names().addCallback(gotKeyList)

    #--- Replacement dict methods as needed -----------------------------------

    def has_key(self, key):
        """
        Returns an immediate or deferred Boolean indicating whether the key is
        present.
        """
        return self.__contains__(key)

    def clear(self):
        """
        Clears the in-memory dictionary of all items and deletes all their
        database entries.
        """
        self.keyCache.clear()
        self.data.clear()
        d = self.writeTracker.deferToAll()
        d.addCallback(lambda _: self.i.names())
        d.addCallback(lambda names: self.i.delete(*names))
        self.writeTracker.put(d)
        return d

    def __iter__(self):
        """
        B{Only for preload mode}: Iterate over all my keys.
        """
        return iter(self.iterkeys)

    def iteritems(self):
        """
        B{Only for preload mode}: Iterate over all my items.
        """
        if self.isPreloadMode:
            for item in self.data.iteritems():
                yield item
        else:
            raise AsyncError("Can't iterate asynchronously")

    def iterkeys(self):
        """
        B{Only for preload mode}: Iterate over all my keys.
        """
        if self.isPreloadMode:
            for key in self.data.iterkeys():
                yield key
        else:
            raise AsyncError("Can't iterate asynchronously")

    def itervalues(self):
        """
        B{Only for preload mode}: Iterate over all my values.
        """
        if self.isPreloadMode:
            for value in self.data.itervalues():
                yield values
        else:
            raise AsyncError("Can't iterate asynchronously")

    def __len__(self):
        """
        Returns an immediate or deferred integer with my length, i.e.,
        the number of keys.
        """
        if self.isPreloadMode:
            return len(self.data)
        return self.loadAll().addCallback(lambda x: len(x))

    def items(self):
        """
        Returns an immediate or deferred sequence of (name, value) tuples
        representing all my items.
        """
        if self.isPreloadMode:
            return self.data.items()
        return self.loadAll().addCallback(lambda x: x.items())

    def values(self):
        """
        Returns an immediate or deferred sequence of all my values.
        """
        if self.isPreloadMode:
            return self.data.values()
        return self.loadAll().addCallback(lambda x: x.values())

    def get(self, *args):
        """
        Returns an immediate or deferred value of the value for the key
        specified as the first argument, or a default value if specified as an
        optional second argument. If the item is not present and no default
        value is supplied, raises the appropriate exception.
        """
        def gotItem(loadedValue, key, defaultValue):
            if isinstance(loadedValue, items.Missing):
                return defaultValue
            self.data[key] = loadedValue
            return loadedValue

        key = args[0]
        if len(args) == 1:
            return self[key]
        defaultValue = args[1]
        if self.isPreloadMode:
            if self.has_key(key):
                return self[key]
            return defaultValue
        d = self.i.load(key)
        d.addCallback(gotItem, key, defaultValue)
        return d

    def setdefault(self, key, value):
        """
        Sets my item specified by I{key} to I{value} if it doesn't exist
        already.  Returns an immediate or deferred reference to the item's
        value after its new value (if any) is set.
        """
        def gotItem(loadedValue):
            if isinstance(loadedValue, items.Missing):
                self.__setitem__(key, value)
                d = self.writeTracker.deferToLast()
                d.addCallback(lambda _: value)
                return d
            self.data[key] = loadedValue
            return loadedValue

        if self.isPreloadMode:
            if key in self.data:
                return self.data[key]
            self.__setitem__(key, value)
            return value
        if key in self.data:
            return defer.succeed(self.data[key])
        return self.i.load(key).addCallback(gotItem)

    def copy(self):
        """
        Returns an immediate or deferred copy of me that is a conventional
        (non-persisted) dictionary.
        """
        if self.isPreloadMode:
            return self.data.copy()
        return self.loadAll()
```

queue.py

```python
# sAsync:
# An enhancement to the SQLAlchemy package that provides persistent
# item-value stores, arrays, and dictionaries, and an access broker for
# conveniently managing database access, table setup, and
# transactions. Everything can be run in an asynchronous fashion using
# the Twisted framework and its deferred processing capabilities.
#
# Copyright (C) 2006, 2015 by Edwin A. Suominen, http://edsuom.com/sAsync
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Queuing for asynchronous database transactions via C{SQLAlchemy}.

"""

import logging

from twisted.internet import defer

import asynqueue

import sqlalchemy as SA
from sqlalchemy import pool


class Factory(object):
    """
    I generate L{asynqueue.ThreadQueue} objects, a unique one for each
    call to me with a unique url-kw combination.
    """
    globalQueue = None

    def __init__(self):
        self.queues = {}

    @staticmethod
    def newQueue(url, **kw):
        """
        Returns a C{Deferred} that fires with a new L{asynqueue.ThreadQueue}
        that has a new SQLAlchemy engine attached as its I{engine}
        attribute.
        """
        def getEngine():
            # Add a NullHandler to avoid "No handlers could be
            # found for logger sqlalchemy.pool." messages
            logging.getLogger(
                "sqlalchemy.pool").addHandler(logging.NullHandler())
            # Now create the engine
            return SA.create_engine(url, **kw)
        def gotEngine(engine):
            q.engine = engine
            return q
        # Iterators are always returned as raw because ResultProxy
        # objects are iterators but sAsync is smarter at handling them
        # than AsynQueue.
        q = asynqueue.ThreadQueue(
            raw=True,
            verbose=kw.pop('verbose', False),
            spew=kw.pop('spew', False),
            returnFailure=kw.pop('returnFailure', False))
        return q.call(getEngine).addCallback(gotEngine)

    @classmethod
    def getGlobal(cls):
        """
        Returns a deferred reference to the global queue, assuming one has
        been defined with L{setGlobal}.

        Calling this method, or L{get} without a url argument, is the
        only approved way to get a reference to the global queue.
        """
        if cls.globalQueue:
            return defer.succeed(cls.globalQueue)
        d = defer.Deferred()
        if hasattr(cls, 'd') and not cls.d.called:
            cls.d.chainDeferred(d)
        else:
            cls.d = d
        return d

    @classmethod
    def setGlobal(cls, url, **kw):
        """
        Sets up a global queue and engine, storing as the default and
        returning a deferred reference to it.

        Calling this method is the only approved way to set the global
        queue.
        """
        def gotQueue(q):
            del cls.d
            cls.globalQueue = q
            return q
        cls.d = cls.newQueue(url, **kw).addCallback(gotQueue)
        return cls.d

    def kill(self, q):
        """
        Removes the supplied queue object from my local queue cache and
        shuts down the queue. Returns a C{Deferred} that fires when
        the removal and shutdown are done.

        Has no effect on the global queue.
        """
        for key, value in self.queues.iteritems():
            if value == q:
                # Found it. Delete and quit looking.
                del self.queues[key]
                break
        if q == self.globalQueue:
            # We can't kill the global queue
            return defer.succeed(None)
        # Shut 'er down
        return q.shutdown()

    def __call__(self, *url, **kw):
        """
        Returns a C{Deferred} that fires with an L{asynqueue.ThreadQueue}
        that has an C{SQLAlchemy} engine attached to it, constructed
        with the supplied url and any keywords. The engine can be
        referenced via the queue's I{engine} attribute.

        If a queue has already been constructed with the same url-kw
        parameters, that same one is returned. Otherwise, a new one is
        constructed and saved for a repeat call.

        If there is no url argument, the global default queue will be
        returned. There must be one for that to work, of course.

        Separate instances of me can have separate queues for the
        exact same url-kw parameters. But all instances share the same
        global queue.
        """
        def gotQueue(q):
            self.queues[key] = q
            return q

        if url:
            url = url[0]
            key = hash((url, tuple(kw.items())))
            if key in self.queues:
                return defer.succeed(self.queues[key])
            return self.newQueue(url, **kw).addCallback(gotQueue)
        return self.getGlobal()
```

selex.py

```
# sAsync:
# An enhancement to the SQLAlchemy package that provides persistent
# item-value stores, arrays, and dictionaries, and an access broker for
# conveniently managing database access, table setup, and
# transactions. Everything can be run in an asynchronous fashion using
# the Twisted framework and its deferred processing capabilities.
#
# Copyright (C) 2006, 2015 by Edwin A. Suominen, http://edsuom.com/sAsync
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Convenient select-object usage for asynchronous database
transactions via C{SQLAlchemy}.
"""

from twisted.internet import defer

import sqlalchemy as SA


class SelectAndResultHolder(object):
    """
    I am yielded by L{database.AccessBroker.selectorator} to let you
    work on (1) a C{select} of the provided columns or (2) an object
    produced by a callable and any args for it, then call me for its
    result.

    Provide my constructor with a reference to the
    L{database.AccessBroker} and the args, plus any keywords you want
    added to the call.

    Everything is cleaned up via my L{close} method after the "loop"
    ends.
    """
    def __init__(self, broker, *args, **kw):
        self.broker = broker
        if callable(args[0]):
            self._sObject = args[0](*args[1:])
        else:
            self._sObject = SA.select(args)
        self.kw = kw

    def _wrapper(self, *args, **kw):
        """
        Replaces the C{select} object with the result of a method of it that
        you obtained as an attribute of me. Henceforth my attributes
        shall be those of the replacement object.
        """
        self._sObject = getattr(self._sObject, self._methodName)(*args, **kw)

    def __getattr__(self, name):
        """
        Access an attribute of my C{select} object (or a replacement obtained
        via a method call) as if it were my own. If the attribute is
        callable, wrap it in my magic object-replacement wrapper
        method.
        """
        obj = getattr(self._sObject, name)
        if callable(obj):
            self._methodName = name
            return self._wrapper
        return obj

    def __call__(self, *args, **kw):
        """
        Executes the C{select} object, with any supplied args and keywords.

        If you call this from within a transaction already, the
        nesting will be dealt with appropriately and you will get an
        immediate C{ResultProxy}. Otherwise, you'll get a deferred that
        fires with the result, with row iteration coolness.

        As with any transaction, you can disable such behavior and get
        either the raw C{ResultProxy} (with I{raw}) or a list of rows
        (with I{asList}). Those transaction keywords can get supplied
        to my constructor or to this call, if it doesn't itself occur
        from inside a transaction.
        """
        kw.update(self.kw)
        self.result = self.broker.execute(self._sObject, *args, **kw)
        return self.result

    def close(self):
        """
        Closes the C{ResultProxy} if possible.
        """
        def closer(rp):
            rp.close()
            return rp

        result = getattr(self, 'result', None)
        if isinstance(result, defer.Deferred):
            result.addCallback(closer)
        elif callable(getattr(result, 'close', None)):
            result.close()
```