repoze.tm Documentation

Release 2.0

Repoze Developers

June 03, 2016

Contents

CHAPTER 1

Overview

repoze.tm2 is WSGI middleware which uses the ZODB package's transaction manager to wrap a call to its pipeline children inside a transaction.

Note: repoze.tm2 is equivalent to the repoze.tm package (it was forked from repoze.tm), except it has a dependency only on the transaction package rather than a dependency on the entire ZODB3 package (ZODB3 3.8 ships with the transaction package right now). It is an error to install both repoze.tm and repoze.tm2 into the same environment, as they provide the same entry points and import points.

CHAPTER 2

Behavior

When this middleware is present in the WSGI pipeline, a new transaction will be started once a WSGI request makes it to the middleware. If any downstream application raises an exception, the transaction will be aborted, otherwise the transaction will be committed. Any "data managers" participating in the transaction will be aborted or committed respectively. A ZODB "connection" is an example of a data manager.

Since this is a tiny wrapper around the ZODB transaction module, and the ZODB transaction module is "thread-safe" (in the sense that its default policy is to create a new transaction for each thread), it should be fine to use in either multiprocess or multithread environments.

CHAPTER 3

Purpose and Usage

The ZODB transaction manager is a completely generic transaction manager. It can be used independently of the actual "object database" part of ZODB. One of the purposes of creating `repoze.tm` was to allow for systems other than Zope to make use of two-phase commit transactions in a WSGI context.

Let's pretend we have an existing system that places data into a relational database when someone submits a form. The system has been running for a while, and our code handles the database commit and rollback for us explicitly; if the form processing succeeds, our code commits the database transaction. If it fails, our code rolls back the database transaction. Everything works fine.

Now our customer asks us if we can also place data into another separate relational database when the form is submitted as well as continuing to place data in the original database. We need to put data in both databases, and if we want to ensure that no records exist in one that don't exist in the other as a result of a form submission, we're going to need to do a pretty complicated commit and rollback dance in each place in our code which needs to write to both data stores. We can't just blindly commit one, then commit the other, because the second commit may fail and we'll be left with "orphan" data in the first, and we'll either need to clean it up manually or leave it there to trip over later.

A transaction manager helps us ensure that no data is committed to either database unless both participating data stores can commit. Once the transaction manager determines that both data stores are willing to commit, it will commit them both in very quick succession, so that there is only a minimal chance that the second data store will fail to commit. If it does, the system will raise an error that makes it impossible to begin another transaction until the system restarts, so the damage is minimized. In practice, this error almost never occurs unless the code that interfaces the database to the transaction manager has a bug.

CHAPTER 4

Adding `repoze.tm2` To Your WSGI Pipeline

Via `PasteDeploy` .INI configuration:

```
[pipeline:main]
 pipeline =
         egg:repoze.tm2#tm
         myapp
```

Via Python:

```
from otherplace import mywsgiapp

from repoze.tm import TM
new_wsgiapp = TM(mywsgiapp)
```

CHAPTER 5

Using A Commit Veto

If you'd like to veto commits based on the status code returned by the downstream application, use a commit veto callback.

First, define the callback somewhere in your application:

```
def commit_veto(environ, status, headers):
    for header_name, header_value in headers:
        if header_name.lower() == 'x-tm':
            if header_value.lower() == 'commit':
                return False
            return True
    for bad in ('4', '5'):
        if status.startswith(bad):
            return True
    return False
```

Then configure it into your middleware.

Via Python:

```
from otherplace import mywsgiapp
from my.package import commit_veto

from repoze.tm import TM
new_wsgiapp = TM(mywsgiapp, commit_veto=commit_veto)
```

Via PasteDeploy:

```
[filter:tm]
commit_veto = my.package:commit_veto
```

In the PasteDeploy example, the path is a Python dotted name, where the dots separate module and package names, and the colon separates a module from its contents. In the above example, the code would be implemented as a "commit_veto" function which lives in the "package" submodule of the "my" package.

A variant of the commit veto implementation shown above as an example is actually present in the `repoze.tm2` package as `repoze.tm.default_commit_veto`. It's fairly general, so you needn't implement one yourself. Instead just use it.

Via Python:

```
from otherplace import mywsgiapp
from repoze.tm import default_commit_veto
```

```
from repoze.tm import TM
new_wsgiapp = TM(mywsgiapp, commit_veto=default_commit_veto)
```

Via PasteDeploy:

```
[filter:tm]
commit_veto = repoze.tm:default_commit_veto
```

API documentation for `default_commit_veto` exists at `repoze.tm.default_commit_veto()`.

CHAPTER 6

Mocking Up A Data Manager

The piece of code you need to write in order to participate in ZODB transactions is called a 'data manager'. It is typically a class. Here's the interface that you need to implement in the code for a data manager:

```
class IDataManager(zope.interface.Interface):
    """Objects that manage transactional storage.

    These objects may manage data for other objects, or they
    may manage non-object storages, such as relational
    databases.  For example, a ZODB.Connection.

    Note that when some data is modified, that data's data
    manager should join a transaction so that data can be
    committed when the user commits the transaction.  """

    transaction_manager = zope.interface.Attribute(
        """The transaction manager (TM) used by this data
        manager.

        This is a public attribute, intended for read-only
        use.  The value is an instance of ITransactionManager,
        typically set by the data manager's constructor.  """
        )

    def abort(transaction):
        """Abort a transaction and forget all changes.

        Abort must be called outside of a two-phase commit.

        Abort is called by the transaction manager to abort transactions
        that are not yet in a two-phase commit.
        """

    # Two-phase commit protocol.  These methods are called by
    # the ITransaction object associated with the transaction
    # being committed.  The sequence of calls normally follows
    # this regular expression: tpc_begin commit tpc_vote
    # (tpc_finish | tpc_abort)

    def tpc_begin(transaction):

        """Begin commit of a transaction, starting the
        two-phase commit.
```

```
        transaction is the ITransaction instance associated with the
        transaction being committed.
        """

    def commit(transaction):

        """Commit modifications to registered objects.

        Save changes to be made persistent if the transaction
        commits (if tpc_finish is called later).  If tpc_abort
        is called later, changes must not persist.

        This includes conflict detection and handling.  If no
        conflicts or errors occur, the data manager should be
        prepared to make the changes persist when tpc_finish
        is called.  """

    def tpc_vote(transaction):
        """Verify that a data manager can commit the transaction.

        This is the last chance for a data manager to vote 'no'.  A
        data manager votes 'no' by raising an exception.

        transaction is the ITransaction instance associated with the
        transaction being committed.
        """

    def tpc_finish(transaction):

        """Indicate confirmation that the transaction is done.

        Make all changes to objects modified by this
        transaction persist.

        transaction is the ITransaction instance associated
        with the transaction being committed.

        This should never fail.  If this raises an exception,
        the database is not expected to maintain consistency;
        it's a serious error.  """

    def tpc_abort(transaction):

        """Abort a transaction.

        This is called by a transaction manager to end a
        two-phase commit on the data manager.  Abandon all
        changes to objects modified by this transaction.

        transaction is the ITransaction instance associated
        with the transaction being committed.

        This should never fail.
        """

    def sortKey():

        """Return a key to use for ordering registered
```

```
        DataManagers.

        ZODB uses a global sort order to prevent deadlock when
        it commits transactions involving multiple resource
        managers.  The resource manager must define a
        sortKey() method that provides a global ordering for
        resource managers.  """
        # Alternate version:
        #"""Return a consistent sort key for this connection.
        # #This allows ordering multiple connections that use
        the same storage in #a consistent manner. This is
        unique for the lifetime of a connection, #which is
        good enough to avoid ZEO deadlocks.  #"""
```

Let's implement a mock data manager. Our mock data manager will write data to a file if the transaction commits. It will not write data to a file if the transaction aborts:

```
class MockDataManager:

    transaction_manager = None

    def __init__(self, data, path):
        self.data = data
        self.path = path

    def abort(self, transaction):
        pass

    def tpc_begin(self, transaction):
        pass

    def commit(self, transaction):
        import tempfile
        self.tempfn = tempfile.mktemp()
        temp = open(self.tempfn, 'wb')
        temp.write(self.data)
        temp.flush()
        temp.close()

    def tpc_vote(self, transaction):
        import os
        if not os.path.exists(self.tempfn):
            raise ValueError('%s doesnt exist' % self.tempfn)
        if os.path.exists(self.path):
            raise ValueError('file already exists')

    def tpc_finish(self, transaction):
        import os
        os.rename(self.tempfn, self.path)

    def tpc_abort(self, transaction):
        import os
        try:
            os.remove(self.tempfn)
        except OSError:
            pass
```

We can create a datamanager and join it into the currently running transaction:

```
dm = MockDataManager('heres the data',  '/tmp/file')
import transaction
t = transaction.get()
t.join(dm)
```

When the transaction commits, a file will be placed in '/tmp/file' containing 'heres the data'. If the transaction aborts, no file will be created.

If more than one data manager is joined to the transaction, all of them must be willing to commit or the entire transaction is aborted and none of them commit. If you can imagine creating two of the mock data managers we've made within application code, if one has a problem during "tpc_vote", neither will actually write a file to the ultimate location, and thus your application consistency is maintained.

CHAPTER 7

Integrating Your Data Manager With `repoze.tm2`

The `repoze.tm2` transaction management machinery has an implicit policy. When it is in the WSGI pipeline, a transaction is started when the middleware is invoked. Thus, in your application code, calling "import transaction; transaction.get()" will return the transaction object created by the `repoze.tm2` middleware. You needn't call t.commit() or t.abort() within your application code. You only need to call t.join, to register your data manager with the transaction. `repoze.tm2` will abort the transaction if an exception is raised by your application code or lower middleware before it returns a WSGI response. If your application or lower middleware raises an exception, the transaction is aborted.

CHAPTER 8

Cleanup

When the repoze.tm2 middleware is in the WSGI pipeline, a boolean key is present in the environment (repoze.tm.active). A utility function named repoze.tm.isActive() can be imported and passed the WSGI environment to check for activation:

```
from repoze.tm import isActive
tm_active = isActive(wsgi_environment)
```

If an application needs to perform an action after a transaction ends, the repoze.tm.after_end registry may be used to register a callback. This object is an instance fo the repoze.tm.AfterEnd class. The repoze.tm.AfterEnd.register() method accepts a callback (accepting no arguments) and a transaction instance:

```
from repoze.tm import after_end
import transaction
t = transaction.get() # the current transaction
def func():
    pass # close a connection, etc
after_end.register(func, t)
```

"after_end" callbacks should only be registered when the transaction manager is active, or a memory leak will result (registration cleanup happens only on transaction commit or abort, which is managed by repoze.tm2 while in the pipeline).

CHAPTER 9

Further Documentation

Many database adapters written for Zope (e.g. for Postgres, MySQL, etc) use this transaction manager, so it should be possible to take a look in these places to see how to implement a more real-world transaction-aware database connector that uses this module in non-Zope applications:

- http://www.zodb.org/en/latest/documentation/guide/transactions.html
- http://mysql-python.sourceforge.net/ (ZMySQLDA)
- http://www.initd.org/svn/psycopg/psycopg2/trunk/ (ZPsycoPGDA)

CHAPTER 10

Contacting

The repoze-dev maillist should be used for communications about this software.

Report bugs on Github: https://github.com/repoze/repoze.tm2/issues

Fork it on Github: https://github.com/repoze/repoze.tm2/

CHAPTER 11

API Docs

11.1 repoze.tm2 API

after_end

CHAPTER 12

Change Logs

12.1 repoze.tm2 Change History

12.2 Changelog

12.2.1 2.1.1 (unreleased)

- TBD

12.2.2 2.1 (2016-06-03)

- Add support for Python 3.4, 3.5 and PyPy3.
- Drop support for Python 2.6 and 3.2.
- Add support for testing under Travis.

12.2.3 2.0 (2013-06-26)

- Avoid swallowing the original exception while aborting the transaction in middleware. See PR #3.

12.2.4 2.0b1 (2013-04-05)

- Middleware is now a generator, to deal appropriately with application iterators which are themselves not lists.
- Convert use of deprecated failIf/failUnless to assertFalse/assertTrue.
- Add support for testing under supported Pythons using Tox.
- Add explicit support for Python 3.2 ad 3.3.
- Drop support for Python 2.4, 2.5.

12.2.5 1.0 (2012-03-24)

- Run OOTB under Python 2.4 / 2.5 (pin 'transaction' dependency to a supported version when running under 2.4 / 2.5).

12.2.6 1.0b2 (2011-07-18)

- A new header `X-Tm` is now honored by the `default_commit_veto` commit veto hook. If this header exists in the headerlist, its value must be a string. If its value is `commit`, the transaction will be committed regardless of the status code or the value of `X-Tm-Abort`. If the value of the `X-Tm` header is `abort` (or any other string value except `commit`), the transaction will be aborted regardless of the status code or the value of `X-Tm-Abort`.
- Use of the `X-Tm-Abort` header is now deprecated. Instead use the `X-Tm` header with a value of `abort` instead.
- Add API docs section.

12.2.7 1.0b1 (2011-01-19)

- Added `repoze.tm.default_commit_veto` commit veto hook. This commit veto hook aborts for 4XX and 5XX response codes, or if there's a header named `X-Tm-Abort` in the headerlist and allows a commit otherwise.
- Documented commit veto hook.

12.2.8 1.0a5 (2009-09-07)

- Don't commit after aborting if the transaction was doomed or if the commit veto aborted.
- Don't use "real" transaction module in tests.
- 100% test coverage.

12.2.9 1.0a4 (2009-01-06)

- RESTify CHANGES, move docs in README.txt into Sphinx.
- Remove `setup.cfg` (all dependencies available via PyPI).
- Synchronization point with `repoze.tm` (0.9).

12.2.10 1.0a3 (2008-08-03)

Allow `commit_veto` hook to be specified within Paste config, ala:

```
[filter:tm]
use = repoze.tm:make_tm
commit_veto = some.package:myfunction
```

`myfunction` should take three args: environ, status, headers and should return True if the txn should be aborted, False if it should be committed.

Initial PyPI release.

12.2.11 1.0a2 (2008-07-15)

- Provide "commit_veto" hook point (contributed by Alberto Valverde).
- Point easy_install at http://dist.repoze.org/tm2/dev/simple via setup.cfg.

12.2.12 1.0a1 (2008-01-09)

- Fork point: we've created repoze.tm2, which is repoze.tm that has a dependency only on the 'transaction' package instead of all of ZODB.
- Better documentation for non-Zope usage in README.txt.

12.2.13 0.8 (2007-10-11)

- Relaxed requirement for ZODB 3.7.2, since we might need to use the package with other verions. Note that the tests which depend on transaction having "doom" semantics don't work with 3.7.2, anyway.

12.2.14 0.7 (2007-09-25)

- Depend on PyPI release of ZODB 3.7.2. Upgrade to this by doing bin/easy_install -U 'ZODB3 >= 3.7.1, < 3.8.0a' if necessary.

12.2.15 0.6 (2007-09-21)

- after_end.register and after_end.unregister must now be passed a transaction object rather than a WSGI environment to avoid the possibility that the WSGI environment used by a child participating in transaction management won't be the same one used by the repoze.tm package.
- repoze.tm now inserts a key into the WSGI environment (`repoze.tm.active`) if it's active in the WSGI pipeline. An API function, repoze.tm:isActive can be called with a single argument, the WSGI environment, to check if the middleware is active.

12.2.16 0.5 (2007-09-18)

- Depend on rerolled ZODB 3.7.1 instead of zopelib.
- Add license and copyright, change trove classifiers.

12.2.17 0.4 (2007-09-17)

- Depend on zopelib rather than ZODB 3.8.0b3 distribution, because the ZODB distribution pulls in various packages (zope.interface and ZEO most notably) that are incompatible with stock Zope 2.10.4 apps and older sandboxes. We'll need to revisit this.

12.2.18 0.3 (2007-09-14)

- Provide limited compatibility for older transaction package versions which don't support the 'transaction.isDoomed' API.

12.2.19 0.2 (2007-09-13)

- Provide after_end API for registering callbacks at transaction end.

12.2.20 0.1 (2007-09-10)

- Initial Release

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

A