Repozitory Documentation

Release 1.2

Shane Hathaway

September 01, 2012

CONTENTS

Repozitory is a library for archiving documents and recovering old versions. It is designed to work in tandem with a primary document storage mechanism such as ZODB. Repozitory stores the document versions in a PostgreSQL or SQLite database using SQLAlchemy. Repozitory was built for KARL, an open source online collaboration system, but Repozitory is intended to be useful for any Python project that stores user-editable documents. Repozitory does not require ZODB.

Using Repozitory, an application can support document versioning without burdening its own database schema with support for multiple document versions. Repozitory provides a stable database schema that rarely needs to change when applications change.

The version control model implemented by Repozitory is designed to be simple for end users, so it is not as elaborate as version control systems designed for software developers. In Repozitory, each document is versioned independently. The attributes of containers are not normally included in Repozitory, but Repozitory tracks the contents of containers in order to provide an undelete facility.

CHAPTER

ONE

USAGE

Install Repozitory using Setuptools:

```
easy_install repozitory
```

The main class in Repozitory is `Archive`. Its constructor requires one parameter, an `EngineParams` object (or any object that has `db_string` and `kwargs` attributes). Here is one way for applications to create an `Archive` instance:

```
from repozitory.archive import EngineParams
from repozitory.archive import Archive

params = EngineParams('sqlite:///', echo=True)
archive = Archive(params)
```

`Archive` objects can be stored in ZODB or any other database that pickles objects, so if your application is based on ZODB, then you probably want to store the Archive object as an attribute of some root object.

1.1 Archiving a document

To archive a document, applications call the `archive` method of the Archive, passing an object that provides the `IObjectVersion` interface. The `IObjectVersion` interface specifies the following attributes.

- **`docid`** A unique 32 bit integer document identifier.
- **`title`** The document title as a Unicode string. May be left blank.
- **`description`** The document description as a Unicode string. May be left blank.
- **`created`** The `datetime` when the document was created. Should be in UTC, so applications should use `datetime.datetime.utcnow()`.
- **`modified`** The `datetime` when the document was last modified. Should be in UTC, so applications should use `datetime.datetime.utcnow()`.
- **`path`** The location of the document as a Unicode string, such as '/some/path/mydoc'. May be left blank, but the path improves the convenience of the relational database, so applications should provide it when possible.
- **`attrs`** The content and other attributes of the document as a JSON-compatible Python dict. The content of this attribute will be encoded as JSON in the relational database. May be set to None. It is not appropriate to include binary streams in the `attrs` attribute; store binary streams using `blobs` instead.
- **`blobs`** A mapping of binary large objects to store with this document. This attribute allows applications to store images and other binary streams of arbitrary size. May be set to None.

Each key in the mapping is a Unicode string. Each value is either a filename or an open file object (such as a StringIO). Open file objects must be seekable.

Repozitory automatically de-duplicates binary streams using MD5 and SHA-256 hashes, so even if many versions of a document (or many documents) use a single large image, Repozitory will store only one copy of that image, saving storage space.

- **`user`** The user who last changed the document, as a Unicode string. Applications should normally store user IDs rather than user names in this attribute.
- **`comment`** The user's comment relating to this version of the document, if any. May be None.
- **`klass` (optional)** The Python class of the document being stored. Repozitory will verify that the class is importable (exists in the scope of some module), since importing the class is often useful for recovery purposes. If this attribute is not provided, Repozitory will try to determine the class automatically.

Repozitory integrates with the `transaction` package, so the results of calling `archive` will not be committed until you call `transaction.commit`. Here is an example of how applications might use the `archive` method.

```
from cStringIO import StringIO
import datetime
import transaction
from repozitory.archive import EngineParams
from repozitory.archive import Archive


class MyDocument(object):
    def __init__(self, docid, title, description, text, image_data):
        self.docid = docid
        self.title = title
        self.description = description
        self.created = datetime.datetime.utcnow()
        self.text = text
        self.image_data = image_data


class MyDocumentVersion(object):
    # Implements IObjectVersion
    def __init__(self, doc, user, comment=None):
        # assert isinstance(doc, MyDocument)
        self.docid = doc.docid
        self.title = doc.title
        self.description = doc.description
        self.created = doc.created
        self.modified = datetime.datetime.utcnow()
        self.path = u'/doc/%d' % doc.docid
        self.attrs = {'text': doc.text}
        self.blobs = {'image': StringIO(doc.image_data)}
        self.user = user
        self.comment = comment
        self.klass = object


d = MyDocument(
    docid=5,
    title=u'The Life of Brain',
    description=(
        u'Brian is born on the original Christmas, in the stable '
        u'next door. He spends his life being mistaken for a messiah.'
    ),
```

```
    text=u'blah blah',
    image_data=(
        'GIF89a\x01\x00\x01\x00\x80\x00\x00\xff\xff\xff\xff\xff\xff!'
        '\xf9\x04\x01\x00\x00\x01\x00,\x00\x00\x00\x00\x01\x00\x01'
        '\x00\x00\x02\x02D\x01\x00;'
    ),
)
params = EngineParams('sqlite:///', echo=False)
archive = Archive(params)
archive.archive(MyDocumentVersion(d, '123', 'Test!'))
d.title = u'The Life of Brian'
archive.archive(MyDocumentVersion(d, '123', 'Corrected title'))
transaction.commit()
```

Again, don't forget to call `transaction.commit`! If you are building a web application with a WSGI pipeline, the best way to call `transaction.commit` is to include a WSGI component such as `repoze.tm2` in your pipeline.

1.2 Reading a document's history

The `history` method of an `Archive` object provides a complete list of versions of a particular document. Pass the document's `docid`. If you want only the current version of the document, add the parameter `only_current=True`. The `history` method returns the most recent version first.

Each item in the history list provides the `IObjectVersion` interface described above, as well as `IObjectHistoryRecord`. If a document contained blobs, those blobs will be provided in the history as open file objects.

The attributes provided by `IObjectHistoryRecord` are:

- **`version_num`** The version number of the document; starts with 1 and increases automatically each time the `archive` method is called.
- **`derived_from_version`** The version number this version was based on. Set to None if this is the first version of the document.

  This is normally (version_num - 1), but that changes if users revert documents to older versions.
- **`current_version`** The current version of the document. Every history record has the same value for `current_version`.
- **`archive_time`** The `datetime` when the version was archived.

  This attribute is controlled by Repozitory, not by the application, so it may be different from the `modified` value.

The following example shows how to read a document's history.

```
>>> h = archive.history(5)
>>> len(h)
2
>>> h[0].title
u'The Life of Brian'
>>> h[0].blobs.keys()
[u'image']
>>> len(h[0].blobs['image'].read())
43
>>> h[0].version_num
2
>>> h[0].derived_from_version
```

```
1
>>> h[0].current_version
2
```

1.3 Reverting

To revert a document, the application should call the `history` method, select an historical state to revert to, and change the corresponding document to match the historical state.

Once the document has been reverted, the application should call the `reverted` method of the archive, passing the `docid` and the `version_num` that the user chose to revert to. After calling `reverted`, it may be a good idea to call `archive` again immediately, causing the reverted version to appear at the top of the version history (but with a new version number).

Continuing the example:

```
>>> h = archive.history(5)
>>> len(h)
2
>>> h[0].title
u'The Life of Brian'
>>> h[1].title
u'The Life of Brain'
>>> d.title = h[1].title
>>> archive.reverted(5, h[1].version_num)
>>> h = archive.history(5, only_current=True)
>>> len(h)
1
>>> h[0].title
u'The Life of Brain'
```

1.4 Undeleting

Repozitory provides a per-container undelete (a.k.a. trash) facility. To use it, call the `archive_container` method after every change to a document container (where a document container is a folder, wiki, or other place where documents are stored). Pass a `container` object, which must provide the `IContainerVersion` interface, and a `user` string, which identifies the user who made the change. Objects that provide the `IContainerVersion` interface must have the following attributes:

- **`container_id`** The 32 bit integer ID of the container. Distinct from `docid`, but it would be wise to avoid a clash between the `docid` and `container_id` spaces.
- **`path`** The location of the container as a Unicode string, such as '/some/path'. May be left blank, but the path improves the convenience of the relational database, so applications should provide it when possible.
- **`map`** The current contents of the container as a mapping of document name (a Unicode string) to document ID. May be an empty mapping.
- **`ns_map`** Additional contents of the container as a mapping where each key is a namespace (a non-empty Unicode string) and each value is a mapping of document name to document ID. This is useful when the container has multiple document namespaces. May be an empty mapping.

When you call the `archive_container` method, Repozitory will detect changes to the container and make records of any deletions and undeletions. Note that `archive_container` does not keep a history; it only updates the record

of the current container contents. If your application needs to keep a history of the container itself, use the `archive` method and make sure the `container_id` and `docid` spaces do not clash.

Continuing the example:

```
class MyContainer(object):
    def __init__(self, container_id, contents):
        self.container_id = container_id
        self.contents = contents

    def __getitem__(self, name):
        return self.contents[name]

class MyContainerVersion(object):
    # Implements IContainerVersion
    def __init__(self, container):
        # assert isinstance(container, MyContainer)
        self.container_id = container.container_id
        self.path = '/container/%d' % container.container_id
        self.map = dict((name, doc.docid)
            for (name, doc) in container.contents.items())
        self.ns_map = {}

c = MyContainer(6, {'movie': d})
archive.archive_container(MyContainerVersion(c), '123')
transaction.commit()
```

Now let's say the user has deleted the single item from the container. The application should record the change using `archive_container`:

```
del c.contents['movie']
archive.archive_container(MyContainerVersion(c), '123')
transaction.commit()
```

The application can use the `container_contents` method of the archive to get the current state of the container and list the documents deleted from the container. The `container_contents` method returns an object that provides `IContainerVersion` as well as `IContainerRecord`, which provides the `deleted` attribute. The `deleted` attribute is a list of objects that provide `IDeletedItem`.

```
>>> cc = archive.container_contents(6)
>>> cc.container_id
6
>>> cc.path
u'/container/6'
>>> cc.map
{}
>>> cc.ns_map
{}
>>> len(cc.deleted)
1
>>> cc.deleted[0].docid
5
>>> cc.deleted[0].name
u'movie'
>>> cc.deleted[0].deleted_by
u'123'
>>> cc.deleted[0].deleted_time is not None
True
>>> cc.deleted[0].new_container_ids is None
```

```
True
```

Note that the new_container_ids attribute in the example above is None, implying the document was deleted, not moved. Let's move the document to a new container.

```
>>> new_container = MyContainer(7, {'movie': d})
>>> archive.archive_container(MyContainerVersion(new_container), '123')
>>> transaction.commit()
>>> new_cc = archive.container_contents(7)
>>> new_cc.container_id
7
>>> new_cc.map
{u'movie': 5}
>>> len(new_cc.deleted)
0
>>> cc = archive.container_contents(6)
>>> cc.container_id
6
>>> cc.map
{}
>>> len(cc.deleted)
1
>>> cc.deleted[0].name
u'movie'
>>> cc.deleted[0].new_container_ids
[7]
```

The result of the container_contents method now shows that the document has been moved to container 7. The application could use this information to redirect users accessing the document in the old container (from a bookmark or a stale search result) to the new document location.

The application can also restore the deleted document by adding it back to the container. In this example, we already have the document as `d`, but in order to get the document to restore, applications normally have to call `history(docid, only_current=True)` and turn the result into a document object.

```
>>> c.contents['movie'] = d
>>> archive.archive_container(MyContainerVersion(c), '123')
>>> transaction.commit()
>>> cc = archive.container_contents(6)
>>> cc.map
{u'movie': 5}
>>> len(cc.deleted)
0
```

As shown in the example, Repozitory removes restored documents from the deleted list.

CHAPTER

TWO

INTERFACE DOCUMENTATION

2.1 IArchive

interface `repozitory.interfaces.`**`IArchive`**

A utility that archives objects.

`archive`(*obj*)

Add a version to the archive of an object.

The obj parameter must provide the IObjectVersion interface. The object does not need to have been in the archive previously.

Returns the new version number.

`iter_hierarchy`(*top_container_id*, *max_depth=None*, *follow_deleted=False*, *follow_moved=False*)

Iterate over IContainerRecords in a hierarchy.

This is more efficient than traversing the hierarchy by calling the container_contents method repeatedly, because this method minimizes the number of calls to the database. Yields an IContainerRecord for each container in the hierarchy.

Set the max_depth parameter to limit the depth of containers to include. When max_depth is 0, only the top container is included; when depth is 1, its children are included, and so on. Set max_depth to None (the default) to include containers of arbitrary depth.

If follow_deleted is true, this method will also include deleted containers and descendants of deleted containers in the results.

If follow_moved is true, this method will also include moved containers and descendants of moved containers in the results.

If no container exists by the given top_container_id, this method returns an empty mapping.

NB: This method assumes that container_ids are also docids. (Most other methods make no such assumption.)

`get_version`(*docid*, *version_num*)

Return a specific IObjectHistoryRecord for a document.

`archive_container`(*container*, *user*)

Update the archive of a container.

The container parameter must provide the IContainerVersion interface. The container does not need to have been in the archive previously. The user parameter (a string) specifies who made the change.

Note that this method does not keep a record of container versions. It only records the contents of the container and tracks deletions, to allow for undeletion of contained objects.

Returns None.

container_contents(*container_id*)
Returns the contents of a container as IContainerRecord.

filter_container_ids(*container_ids*)
Returns which of the specified container IDs exist in the archive.

Returns a sequence containing a subset of the provided container_ids.

shred(*docids=()*, *container_ids=()*)
Delete the specified objects and containers permanently.

The containers to shred must not contain any objects (exempting the objects to be shredded), or a ValueError will be raised.

Returns None.

which_contain_deleted(*container_ids*, *max_depth=None*)
Returns the subset of container_ids that have had something deleted.

This is useful for building a hierarchical trash UI that allows users to visit only containers that have had something deleted. All descendant containers are examined (unless max_depth limits the search.)

Returns a sequence containing a subset of the provided container_ids.

NB: This method assumes that container_ids are also docids. (Most other methods make no such assumption.)

reverted(*docid*, *version_num*)
Tell the database that an object has been reverted.

history(*docid*, *only_current=False*)
Get the history of an object.

Returns a list of objects that provide IObjectHistoryRecord. The most recent version number is listed first.

If only_current is true, only the current history record is returned in the list. (The most current history record might not be the most recent version number if the object has been reverted.)

2.2 IObjectVersion

interface repozitory.interfaces.**IObjectVersion**
Extends: repozitory.interfaces.IDCDescriptiveProperties, repozitory.interfaces.IDCTimes

The content of an object for version control.

Note that the following attributes are required, as specified by the base interfaces:

title description created modified

The title and description should be unicode.

comment
The comment linked to the version; may be None.

docid
The docid of the object as an integer.

blobs
A map of binary large objects linked to this state. May be None.

Each key is a unicode string. Each value is either a filename or an open file object (such as a StringIO). Open file objects must be seekable.

user
The user who made the change. (A string)

attrs
The attributes to store as a JSON-encodeable dictionary.

May be None.

path
The path of the object as a Unicode string.

Used only as metadata. May be left empty.

klass
Optional: the class of the object.

To detect the class automatically, do not provide this attribute, or set it to None.

2.3 IObjectHistoryRecord

interface repozitory.interfaces.**IObjectHistoryRecord**
Extends: repozitory.interfaces.IObjectVersion

An historical record of an object version.

The IArchive.history() method returns objects that provide this interface. All blobs returned by the history() method are open files (not filenames).

version_num
The version number of the object; starts with 1.

archive_time
The datetime in UTC when the version was archived.

May be different from the modified attribute. The modified attribute is set by the application, whereas the archive_time attribute is set by repozitory automatically.

current_version
The current version number of the object.

derived_from_version
The version number this version was based on.

None if the object was created in this version.

2.4 IContainerVersion

interface repozitory.interfaces.**IContainerVersion**
The contents of a container for version control.

ns_map
Namespaced container items, as {ns: {name: docid}}.

All namespaces and names must be non-empty strings and all referenced objects must already exist in the archive.

map
The current items in the container, as {name: docid}.

All names must be non-empty strings and all referenced objects must already exist in the archive.

container_id
The ID of the container as an integer.

path
The path of the container as a Unicode string.

Used only as metadata. May be left empty.

2.5 IContainerRecord

interface repozitory.interfaces.**IContainerRecord**
Extends: repozitory.interfaces.IContainerVersion

Provides the current and deleted contents of a container.

deleted
The deleted items in the container as a list of IDeletedItem.

The most recently deleted items are listed first.

A item name may appear more than once in the list if it has referred to different docids at different times. A docid will never appear more than once in the list, since adding an object to a container causes the corresponding docid to be removed from the deleted list for that container.

2.6 IDeletedItem

interface repozitory.interfaces.**IDeletedItem**
A record of an item deleted from a container.

docid
The docid of the deleted object as an integer.

name
The object's former name within the container.

deleted_time
When the object was deleted (a UTC datetime).

moved
True if this item was moved rather than deleted.

True when new_container_ids is non-empty.

namespace
The object's former namespace in the container.

Empty if the object was not in a namespace.

deleted_by
Who deleted the object (a string).

new_container_ids
Container(s) where the object now exists.

Empty or None if the object is not currently in any container.

CHAPTER

THREE

COMPARISON WITH ZOPEVERSIONCONTROL

Repozitory and the older ZopeVersionControl product perform a similar function but do it differently. Both serve as an archive of document versions, but ZopeVersionControl performs its work by copying complete ZODB objects to and from a ZODB-based archive, while Repozitory expects the application to translate data when copying data to and from the archive.

ZopeVersionControl was designed to minimize the amount of code required to integrate version control into an existing ZODB application, but in practice, it turned out that debugging applications that use ZopeVersionControl is often painful. Applications failed to correctly distinguish between leaf objects and objects with branches, causing ZopeVersionControl to either version too many objects or not enough.

Repozitory is less ambitious. Repozitory requires more integration code than ZopeVersionControl requires, but the integration code is likely to be more straightforward and easier to debug.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

r