Beaker Documentation

Release 1.6.1

Ben Bangert, Mike Bayer

December 13, 2011

CONTENTS

CHAPTER

ONE

CONFIGURATION

Beaker can be configured several different ways, depending on how it's used. The most recommended style is to use a dictionary of preferences that are to be passed to either the `SessionMiddleware` or the `CacheManager`.

Since both Beaker's sessions and caching use the same back-end container storage system, there's some options that are applicable to both of them in addition to session and cache specific configuration.

Most options can be specified as a string (necessary to config options that are setup in INI files), and will be coerced to the appropriate value. Only datetime's and timedelta's cannot be coerced and must be the actual objects.

Frameworks using Beaker usually allow both caching and sessions to be configured in the same spot, Beaker assumes this condition as well and requires options for caching and sessions to be prefixed appropriately.

For example, to configure the `cookie_expires` option for Beaker sessions below, an appropriate entry in a Pylons INI file would be:

```
# Setting cookie_expires = true causes Beaker to omit the
# expires= field from the Set-Cookie: header, signaling the cookie
# should be discarded when the browser closes.
beaker.session.cookie_expires = true
```

Note: When using the options in a framework like Pylons or TurboGears2, these options must be prefixed by `beaker.`, for example in a Pylons INI file:

```
beaker.session.data_dir = %(here)s/data/sessions/data
beaker.session.lock_dir = %(here)s/data/sessions/lock
```

Or when using stand-alone with the `SessionMiddleware`:

```
from beaker.middleware import SessionMiddleware

session_opts = {
    'session.cookie_expires': True
}

app = SomeWSGIAPP()
app = SessionMiddleware(app, session_opts)
```

Or when using the `CacheManager`:

```
from beaker.cache import CacheManager
from beaker.util import parse_cache_config_options
```

```
cache_opts = {
    'cache.type': 'file',
    'cache.data_dir': '/tmp/cache/data',
    'cache.lock_dir': '/tmp/cache/lock'
}

cache = CacheManager(**parse_cache_config_options(cache_opts))
```

Note: When using the CacheManager directly, all dict options must be run through the `beaker.util.parse_cache_config_options()` function to ensure they're valid and of the appropriate type.

1.1 Options For Sessions and Caching

data_dir (optional, string) Used with any back-end that stores its data in physical files, such as the dbm or file-based back-ends. This path should be an absolute path to the directory that stores the files.

lock_dir (required, string) Used with every back-end, to coordinate locking. With caching, this lock file is used to ensure that multiple processes/threads aren't attempting to re-create the same value at the same time (The *Dog-Pile Effect*)

memcache_module (optional, string) One of the names `memcache`, `cmemcache`, `pylibmc`, or `auto`. Default is `auto`. Specifies which memcached client library should be imported when using the ext:memcached backend. If left at its default of `auto`, `pylibmc` is favored first, then `cmemcache`, then `memcache`. New in 1.5.5.

type (required, string) The name of the back-end to use for storing the sessions or cache objects.

Available back-ends supplied with Beaker: `file`, `dbm`, `memory`, `ext:memcached`, `ext:database`, `ext:google`

For sessions, the additional type of `cookie` is available which will store all the session data in the cookie itself. As such, size limitations apply (4096 bytes).

Some of these back-ends require the url option as listed below.

webtest_varname (optional, string) The name of the attribute to use when stashing the session object into the environ for use with WebTest. The name provided here is where the session object will be attached to the WebTest TestApp return value.

url (optional, string) URL is specific to use of either ext:memcached or ext:database. When using one of those types, this option is **required**.

When used with ext:memcached, this should be either a single, or semi-colon separated list of memcached servers:

```
session_opts = {
    'session.type': 'ext:memcached',
    'session.url': '127.0.0.1:11211',
}
```

When used with ext:database, this should be a valid SQLAlchemy database string.

1.2 Session Options

The Session handling takes a variety of additional options relevant to how it stores session id's in cookies, and when using the optional encryption.

auto (optional, bool) When set to True, the session will save itself anytime it is accessed during a request, negating the need to issue the `save()` method.

Defaults to False.

cookie_expires (optional, bool, datetime, timedelta, int) Determines when the cookie used to track the client-side of the session will expire. When set to a boolean value, it will either expire at the end of the browsers session, or never expire.

Setting to a datetime forces a hard ending time for the session (generally used for setting a session to a far off date).

Setting to an integer will result in the cookie being set to expire in that many seconds. I.e. a value of `300` will result in the cookie being set to expire in 300 seconds.

Defaults to never expiring.

cookie_domain (optional, string) What domain the cookie should be set to. When using sub-domains, this should be set to the main domain the cookie should be valid for. For example, if a cookie should be valid under `www.nowhere.com` **and** `files.nowhere.com` then it should be set to `.nowhere.com`.

Defaults to the current domain in its entirety.

Alternatively, the domain can be set dynamically on the session by calling, see *Session Attributes / Keys*.

key (required, string) Name of the cookie key used to save the session under.

secret (required, string) Used with the HMAC to ensure session integrity. This value should ideally be a randomly generated string.

When using in a cluster environment, the secret must be the same on every machine.

secure (optional, bool) Whether or not the session cookie should be marked as secure. When marked as secure, browsers are instructed to not send the cookie over anything other than an SSL connection.

timeout (optional, integer) Seconds until the session is considered invalid, after which it will be ignored and invalidated. This number is based on the time since the session was last accessed, not from when the session was created.

Defaults to never expiring.

1.2.1 Encryption Options

These options should then be used *instead* of the `secret` option listed above.

encrypt_key (required, string) Encryption key to use for the AES cipher. This should be a fairly long randomly generated string.

validate_key (required, string) Validation key used to sign the AES encrypted data.

Note: You may need to install additional libraries to use Beaker's cookie-based session encryption. See the *Encryption* section for more information.

1.3 Cache Options

For caching, options may be directly specified on a per-use basis with the `cache()` decorator, with the rest of these options used as fallback should one of them not be specified in the call.

Only the `lock_dir` option is strictly required, unless using the file-based back-ends as noted with the sessions.

expire (optional, integer) Seconds until the cache is considered old and a new value is created.

1.3.1 Cache Region Options

Starting in Beaker 1.3, cache regions are now supported. These can be thought of as bundles of configuration options to apply, rather than specifying the type and expiration on a per-usage basis.

enabled (optional, bool) Quick toggle to disable or enable caching across an entire application.

This should generally be used when testing an application or in development when caching should be ignored.

Defaults to True.

regions (optional, list, tuple) Names of the regions that are to be configured.

For each region, all of the other cache options are valid and will be read out of the cache options for that key. Options that are not listed under a region will be used globally in the cache unless a region specifies a different value.

For example, to specify two batches of options, one called `long-term`, and one called `short-term`:

```
cache_opts = {
    'cache.data_dir': '/tmp/cache/data',
    'cache.lock_dir': '/tmp/cache/lock'
    'cache.regions': 'short_term, long_term',
    'cache.short_term.type': 'ext:memcached',
    'cache.short_term.url': '127.0.0.1.11211',
    'cache.short_term.expire': '3600',
    'cache.long_term.type': 'file',
    'cache.long_term.expire': '86400',
```

CHAPTER

TWO

SESSIONS

2.1 About

Sessions provide a place to persist data in web applications, Beaker's session system simplifies session implementation details by providing WSGI middleware that handles them.

All cookies are signed with an HMAC signature to prevent tampering by the client.

2.1.1 Lazy-Loading

Only when a session object is actually accessed will the session be loaded from the file-system, preventing performance hits on pages that don't use the session.

2.2 Using

The session object provided by Beaker's `SessionMiddleware` implements a dict-style interface with a few additional object methods. Once the SessionMiddleware is in place, a session object will be made available as `beaker.session` in the WSGI environ.

Getting data out of the session:

```
myvar = session['somekey']
```

Testing for a value:

```
logged_in = 'user_id' in session
```

Adding data to the session:

```
session['name'] = 'Fred Smith'
```

Complete example using a basic WSGI app with sessions:

```
from beaker.middleware import SessionMiddleware

def simple_app(environ, start_response):
    # Get the session object from the environ
    session = environ['beaker.session']

    # Check to see if a value is in the session
```

```
    if 'logged_in' in session:
        user = True
    else:
        user = False

    # Set some other session variable
    session['user_id'] = 10

    start_response('200 OK', [('Content-type', 'text/plain')])
    return ['User is logged in: %s' % user]

# Configure the SessionMiddleware
session_opts = {
    'session.type': 'file',
    'session.cookie_expires': True,
}
wsgi_app = SessionMiddleware(simple_app, session_opts)
```

Note: This example does **not** actually save the session for the next request. Adding the `save()` call explained below is required, or having the session set to auto-save.

2.2.1 Session Attributes / Keys

Sessions have several special attributes that can be used as needed by an application.

- id - Unique 40 char SHA-generated session ID
- last_accessed - The last time the session was accessed before the current access, will be None if the session was just made

There's several special session keys populated as well:

- _accessed_time - Current accessed time of the session, when it was loaded
- _creation_time - When the session was created

2.3 Saving

Sessions can be saved using the `save()` method on the session object:

```
session.save()
```

Warning: Beaker relies on Python's pickle module to pickle data objects for storage in the session. Objects that cannot be pickled should **not** be stored in the session.

This flags a session to be saved, and it will be stored on the chosen back-end at the end of the request.

If it's necessary to immediately save the session to the back-end, the `persist()` method should be used:

```
session.persist()
```

This is not usually the case however, as a session generally should not be saved should something catastrophic happen during a request.

Order Matters: When using the Beaker middleware, you **must call save before the headers are sent to the client**. Since Beaker's middleware watches for when the `start_response` function is called to know that it should add its cookie header, the session must be saved before its called.

Keep in mind that Response objects in popular frameworks (WebOb, Werkzeug, etc.) call start_response immediately, so if you are using one of those objects to handle your Response, you must call .save() before the Response object is called:

```
# this would apply to WebOb and possibly others too
from werkzeug.wrappers import Response

# this will work
def sessions_work(environ, start_response):
    environ['beaker.session']['count'] += 1
    resp = Response('hello')
    environ['beaker.session'].save()
    return resp(environ, start_response)

# this will not work
def sessions_broken(environ, start_response):
    environ['beaker.session']['count'] += 1
    resp = Response('hello')
    retval = resp(environ, start_response)
    environ['beaker.session'].save()
    return retval
```

2.3.1 Auto-save

Saves can be done automatically by setting the `auto` configuration option for sessions. When set, calling the `save()` method is no longer required, and the session will be saved automatically anytime its accessed during a request.

2.4 Deleting

Calling the `delete()` method deletes the session from the back-end storage and sends an expiration on the cookie requesting the browser to clear it:

```
session.delete()
```

This should be used at the end of a request when the session should be deleted and will not be used further in the request.

If a session should be invalidated, and a new session created and used during the request, the `invalidate()` method should be used:

```
session.invalidate()
```

2.4.1 Removing Expired/Old Sessions

Beaker does **not** automatically delete expired or old cookies on any of its back-ends. This task is left up to the developer based on how sessions are being used, and on what back-end.

The database backend records the last accessed time as a column in the database so a script could be run to delete session rows in the database that haven't been used in a long time.

When using the file-based sessions, a script could run to remove files that haven't been touched in a long time, for example (in the session's data dir):

```
find . -mtime +3 -exec rm {} \;
```

2.5 Cookie Domain and Path

In addition to setting a default cookie domain with the *cookie domain setting*, the cookie's domain and path can be set dynamically for a session with the domain and path properties.

These settings will persist as long as the cookie exists, or until changed.

Example:

```
# Setting the session's cookie domain and path
session.domain = '.domain.com'
session.path = '/admin'
```

2.6 Cookie-Based

Session can be stored purely on the client-side using cookie-based sessions. This option can be turned on by setting the session type to `cookie`.

Using cookie-based session carries the limitation of how large a cookie can be (generally 4096 bytes). An exception will be thrown should a session get too large to fit in a cookie, so using cookie-based session should be done carefully and only small bits of data should be stored in them (the users login name, admin status, etc.).

Large cookies can slow down page-loads as they increase latency to every page request since the cookie is sent for every request under that domain. Static content such as images and Javascript should be served off a domain that the cookie is not valid for to prevent this.

Cookie-based sessions scale easily in a clustered environment as there's no need for a shared storage system when different servers handle the same session.

2.6.1 Encryption

In the event that the cookie-based sessions should also be encrypted to prevent the user from being able to decode the data (in addition to not being able to tamper with it), Beaker can use 256-bit AES encryption to secure the contents of the cookie.

Depending on the Python implementation used, Beaker may require an additional library to provide AES encryption.

On CPython (the regular Python), the pycryptopp library or PyCrypto library is required.

On Jython, no additional packages are required, but at least on the Sun JRE, the size of the encryption key is by default limited to 128 bits, which causes generated sessions to be incompatible with those generated in CPython, and vice versa. To overcome this limitation, you need to install the unlimited strength juristiction policy files from Sun:

- Policy files for Java 5
- Policy files for Java 6

CHAPTER

THREE

CACHING

3.1 About

Beaker's caching system was originally based off the Perl Cache::Cache module, which was ported for use in Myghty. Beaker was then extracted from this code, and has been substantially rewritten and modernized.

Several concepts still exist from this origin though. Beaker's caching (and its sessions, though its behind the scenes) utilize the concept of *NamespaceManager*, and *Container* objects to handle storing cached data.

Each back-end utilizes a customized version of each of these objects to handle storing data appropriately depending on the type of the back-end.

The `CacheManager` is responsible for getting the appropriate NamespaceManager, which then stores the cached values. Each namespace corresponds to a single `thing` that should be cached. Usually a single `thing` to be cached might vary slightly depending on parameters, for example a template might need several different copies of itself stored depending on whether a user is logged in or not. Each one of these copies is then `keyed` under the NamespaceManager and stored in a Container.

There are three schemes for using Beaker's caching, the first and more traditional style is the programmatic API. This exposes the namespace's and retrieves a `Cache` object that handles storing keyed values in a NamespaceManager with Container objects.

The more elegant system, introduced in Beaker 1.3, is to use the *cache decorators*, these also support the use of *Cache Regions*.

Introduced in Beaker 1.5 is a more flexible `cache_region()` decorator capable of decorating functions for use with Beaker's *Cache Regions* **before** Beaker has been configured. This makes it possible to easily use Beaker's region caching decorator on functions in the module level.

3.2 Creating the CacheManager Instance

Before using Beaker's caching, an instance of the `CacheManager` class should be created. All of the examples below assume that it has already been created.

Creating the cache instance:

```
from beaker.cache import CacheManager
from beaker.util import parse_cache_config_options

cache_opts = {
    'cache.type': 'file',
    'cache.data_dir': '/tmp/cache/data',
```

```
    'cache.lock_dir': '/tmp/cache/lock'
}

cache = CacheManager(**parse_cache_config_options(cache_opts))
```

Additional configuration options are documented in the *Configuration* section of the Beaker docs.

3.3 Programmatic API

To store data for a cache value, first, a NamespaceManager has to be retrieved to manage the keys for a `thing` to be cached:

```
# Assuming that cache is an already created CacheManager instance
tmpl_cache = cache.get_cache('mytemplate.html', type='dbm', expire=3600)
```

Individual values should be stored using a creation function, which will be called anytime the cache has expired or a new copy needs to be made. The creation function must not accept any arguments as it won't be called with any. Options affecting the created value can be passed in by using closure scope on the creation function:

```
search_param = 'gophers'

def get_results():
    # do something to retrieve data
    data = get_data(search_param)
    return data

# Cache this function, based on the search_param, using the tmpl_cache
# instance from the prior example
results = tmpl_cache.get(key=search_param, createfunc=get_results)
```

3.3.1 Invalidating

All of the values for a particular namespace can be removed by calling the `clear()` method:

```
tmpl_cache.clear()
```

Note that this only clears the key's in the namespace that this particular Cache instance is aware of. Therefore its recommend to manually clear out specific keys in a cache namespace that should be removed:

```
tmpl_cache.remove_value(key=search_param)
```

3.4 Decorator API

When using the decorator API, a namespace does not need to be specified and will instead be created for you with the name of the module + the name of the function that will have its output cached.

Since its possible that multiple functions in the same module might have the same name, additional arguments can be provided to the decorators that will be used in the namespace to prevent multiple functions from caching their values in the same location.

For example:

```
# Assuming that cache is an already created CacheManager instance
@cache.cache('my_search_func', expire=3600)
def get_results(search_param):
    # do something to retrieve data
    data = get_data(search_param)
    return data

results = get_results('gophers')
```

The non-keyword arguments to the `cache()` method are the additional ones used to ensure this function's cache results won't clash with another function in this module called `get_results`.

The cache expire argument is specified as a keyword argument. Other valid arguments to the `get_cache()` method such as `type` can also be passed in.

When using the decorator, the function to cache can have arguments, which will be used as the key was in the *Programmatic API* for the data generated.

Warning: These arguments can **not** be keyword arguments.

3.4.1 Invalidating

Since the `cache()` decorator hides the namespace used, manually removing the key requires the use of the `invalidate()` function. To invalidate the 'gophers' result that the prior example referred to:

```
cache.invalidate(get_results, 'my_search_func', 'gophers')
```

If however, a type was specified for the cached function, the type must also be given to the `invalidate()` function so that it can remove the value from the appropriate back-end.

Example:

```
# Assuming that cache is an already created CacheManager instance
@cache.cache('my_search_func', type="file", expire=3600)
def get_results(search_param):
    # do something to retrieve data
    data = get_data(search_param)
    return data

cache.invalidate(get_results, 'my_search_func', 'gophers', type="file")
```

Note: Both the arguments used to specify the additional namespace info to the cache decorator **and** the arguments sent to the function need to be given to the `region_invalidate()` function so that it can properly locate the namespace and cache key to remove.

3.5 Cache Regions

Rather than having to specify the expiration, or toggle the type used for caching different functions, commonly used cache parameters can be defined as *Cache Regions*. These user-defined regions than may be used with the `region()` decorator rather than passing the configuration.

This can be useful if there are a few common cache schemes used by an application that should be setup in a single place then used as appropriate throughout the application.

Setting up cache region's is documented in the *cache region options* section in *Configuration*.

Assuming a `long_term` and `short_term` region were setup, the `region()` decorator can be used:

```
@cache.region('short_term', 'my_search_func')
def get_results(search_param):
    # do something to retrieve data
    data = get_data(search_param)
    return data

results = get_results('gophers')
```

Or using the `cache_region()` decorator:

```
@cache_region('short_term', 'my_search_func')
def get_results(search_param):
    # do something to retrieve data
    data = get_data(search_param)
    return data

results = get_results('gophers')
```

The only difference with the `cache_region()` decorator is that the cache does not need to be configured when its used. This allows one to decorate functions in a module before the Beaker cache is configured.

3.5.1 Invalidating

Since the `region()` decorator hides the namespace used, manually removing the key requires the use of the `region_invalidate()` function. To invalidate the 'gophers' result that the prior example referred to:

```
cache.region_invalidate(get_results, None, 'my_search_func', 'gophers')
```

Or when using the `cache_region()` decorator, the `beaker.cache.region_invalidate()` function should be used:

```
region_invalidate(get_results, None, 'my_search_func', 'gophers')
```

Note: Both the arguments used to specify the additional namespace info to the cache decorator **and** the arguments sent to the function need to be given to the `region_invalidate()` function so that it can properly locate the namespace and cache key to remove.

CHAPTER

FOUR

CHANGES IN BEAKER

4.1 Release 1.6.1 (10/20/2011)

- Remove stray print statement.
- Include .app for consistency instead of requiring wrap_app.

4.2 Release 1.6 (10/16/2011)

- Fix bug with cache_key length calculation.
- Fix bug with how path was set so that its restored properly and propagated.
- Fix bug with CacheMiddleware clobbering enabled setting.
- Update option for `cookie_expires` so that it can now handle an integer which will be used as the seconds till the cookie expires.
- Merge fix for Issue 31, can now handle unicode cache keys.
- Add `key_length` option for cache regions, and for keyword args passed into the cache system. Cache keys longer than this will be SHA'd.
- added runtime beaker.__version__
- Add `webtest_varname` option to configuration to optionally include the session value in the environ vars when using Beaker with WebTest.
- Defer running of pkg_resources to look for external cache modules until requested. #66
- memcached backend uses pylibmc.ThreadMappedPool to ensure thread-local usage of pylibmc when that library is in use. (#60)
- memcached backend also has `memcache_module` string argument, allows direct specification of the name of which memcache backend to use.
- Basic container/file-based Session support working in Py3K. (#72)
- Further Python 3 fixes
- Added an optimization to the FileNamespaceContainer when used with Session, such that the pickled contents of the file are not read a second time when session.save() is called. (#64)
- Fixed bug whereby CacheManager.invalidate wouldn't work for a function decorated by cache.cache(). (#61)

- cache decorators @cache.cache(), @cache_region() won't include first argument named 'self' or 'cls' as part of the cache key. This allows reasonably safe usage for methods as well as functions. (#55)
- file backend no longer squashes unpickling errors. This was inconsistent behavior versus all the other backends.
- invalidate_corrupt flag on Session now emits a warning. (#52)
- cache.remove_value() removes the value even if it's already marked 'expired' (#42)

4.3 Release 1.5.4 (6/16/2010)

- Fix import error with InvalidCryptoBackendError.
- Fix for domain querying on property.
- Test cleanups
- Fix bug with warnings preventing proper running under Jython.

4.4 Release 1.5.3 (3/2/2010)

- Fix Python 2.4 incompatibility with google import.

4.5 Release 1.5.2 (3/1/2010)

- pkg_resources scanning for additional Beaker back-ends gracefully handles situations where its not present (GAE). Fixes #36.
- Avoid timing attacks on hash comparison.
- Provide abstract base for MemoryNamespaceManager that deals with "dictionaries".
- Added tests for invalidating cache, and fixed bug with function cache when no args are present.
- The SQLAlchemy backends require SQLAlchemy 0.4 or greater (0.6 recommended).
- Rudimental Python 3 support is now available. Simply use Python 3 with Distribute and "python setup.py install" to run 2to3 automatically, or manually run 2to3 on "beaker" and "tests" to convert to a Python 3 version.
- Added support for PyCrypto module to encrypted session, etc. in addition to the existing pycryptopp support.

4.6 Release 1.5.1 (12/17/2009)

- Fix cache namespacing.

4.7 Release 1.5 (11/23/2009)

- Update memcached to default to using pylibmc when available.
- Fix bug when cache value doesn't exist causing has_key to throw an exception rather than return False. Fixes #24.
- Fix bug where getpid under GAE is used improperly to assume it should be a non-string. Fixes #22.
- Add cache_region decorator that works *before* configuration of the cache regions have been completed for use in module-level decorations.
- Fix bug where has_value sees the value before its removed.
- Improved accuracy of "dogpile" checker by removing dependency on "self" attributes, which seem to be slightly unreliable in highly concurrent scenarios.

4.8 Release 1.4.2 (9/25/2009)

- Fix bug where memcached may yank a value after the has_value but before the value can be fetched.
- Fix properties for setting the path. Fixes #15.
- Fix the 'TypeError: argument must be an int, or have a fileno() method' erorr sporadically emitted by FileSynchronizer under moderate load.

4.9 Release 1.4.1 (9/10/2009)

- Fix verification of options to throw an error if a beaker param is an empty string.
- Add CacheManager.invalidate function to easily invalidate cache spaces created by the use of the cache decorator.
- Add CacheManager.region_invalidate function to easily invalidate cache spaces created by the use of the cache_region decorator.
- Fix the InvalidCryptoBackendError exception triggering a TypeError. Patch from dz, fixes #13.

4.10 Release 1.4 (7/24/2009)

- Fix bug with hmac on Python 2.4. Patch from toshio, closes ticket #2133 from the TurboGears2 Trac.
- Fix bug with occasional ValueError from FileNamespaceManager.do_open. Fixes #10.
- Fixed bug with session files being saved despite being new and not saved.
- Fixed bug with CacheMiddleware overwriting configuration with default arguments despite prior setting.
- Fixed bug with SyntaxError not being caught properly in entry point discovery.
- Changed to using BlobProperty for Google Datastore.
- Added domain/path properties to the session. This allows one to dynamically set the cookie's domain and/or path on the fly, which will then be set on the cookie for the session.

- Added support for cookie-based sessions in Jython via the JCE (Java Cryptography Extensions). Patch from Alex Grönholm.
- Update Beaker database extensions to work with SQLAlchemy 0.6 PostgreSQL, and Jython.

4.11 Release 1.3.1 (5/5/2009)

- Added a whole bunch of Sphinx documentation for the updated site.
- Added corresponding remove as an alias to the caches remove_value.
- Fixed cookie session not having an invalidate function.
- Fix bug with CacheMiddleware not using proper function to load configuration options, missing the cache regions.

4.12 Release 1.3 (4/6/2009)

- Added last_accessed attribute to session to indicate the previous time the session was last accessed.
- Added setuptools entry points to dynamically discover additional namespace backends.
- Fixed bug with invalidate and locks, fixes #594.
- Added cache.cache decorator for arbitrary caching.
- Added cache.region decorator to the CacheManager object.
- Added cache regions. Can be provided in a configuration INI type, or by adding in a cache_regions arg to the CacheManager.
- Fix bug with timeout not being saved properly.
- Fix bug with cookie-only sessions sending cookies for new sessions even if they weren't supposed to be saved.
- Fix bug that caused a non-auto accessed session to not record the time it was previously accessed resulting in session timeouts.
- Add function to parse configuration dicts as appropriate for use with the CacheManager.
- The "expiretime" is no longer passed to the memcached backend - since if memcached makes the expired item unavailable at the same time the container expires it, then all actors must block until the new value is available (i.e. breaks the anti-dogpile logic).

4.13 Release 1.2.3 (3/2/2009)

- Fix accessed increment to take place *after* the accessed time is checked to see if it has expired. Fixes #580.
- data_dir/lock_dir parameters are optional to most backends; if not present, mutex-based locking will be used for creation functions
- Adjustments to Container to better account for backends which don't provide read/write locks, such as memcached. As a result, the plain "memory" cache no longer requires read/write mutexing.

4.14 Release 1.2.2 (2/14/2009)

- Fix delete bug reported by andres with session not being deleted.

4.15 Release 1.2.1 (2/09/2009)

- Fix memcached behavior as memcached returns None on nonexistent key fetch which broke invalid session checking.

4.16 Release 1.2 (1/22/2009)

- Updated session to only save to the storage *once* no under any/all conditions rather than every time save() is called.
- Added session.revert() function that reverts the session to the state at the beginning of the request.
- Updated session to store entire session data in a single namespace key, this lets memcached work properly, and makes for more efficient use of the storage system for sessions.

4.17 Release 1.1.3 (12/29/2008)

- Fix the 1.1.2 old cache/session upgrader to handle the has_current_value method.
- Make InvalidCacheBackendError an ImportError.

4.18 Release 1.1.2 (11/24/2008)

- Upgrade Beaker pre-1.1 cache/session values to the new format rather than throwing an exception.

4.19 Release 1.1.1 (11/24/2008)

- Fixed bug in Google extension which passed arguments it should no longer pass to NamespaceManager.
- Fixed bug involving lockfiles left open during cache "value creation" step.

4.20 Release 1.1 (11/16/2008)

- file-based cache will not hold onto cached value once read from file; will create new value if the file is deleted as opposed to re-using what was last read. This allows external removal of files to be used as a cache-invalidation mechanism.
- file-based locking will not unlink lockfiles; this can interfere with the flock() mechanism in the event that a concurrent process is accessing the files.

- Sending "type" and other namespace config arguments to cache.get()/cache.put()/cache.remove_value() is deprecated. The namespace configuration is now preferred at the Cache level, i.e. when you construct a Cache or call cache_manager.get_cache(). This removes the ambiguity of Cache's dictionary interface and has_key() methods, which have no awareness of those arguments.
- the "expiretime" in use is stored in the cache itself, so that it is always available when calling has_key() and other methods. Between this change and the deprecation of 'type', the Cache no longer has any need to store cache configuration in memory per cache key, which in a dynamically-generated key scenario stores an arbitrarily large number of configurations - essentially a memory leak.
- memcache caching has been vastly improved, no longer stores a list of all keys, which along the same theme prevented efficient usage for an arbitrarily large number of keys. The keys() method is now unimplemented, and cache.remove() clears the entire memcache cache across all namespaces. This is what the memcache API provides so it's the best we can do.
- memcache caching passes along "expiretime" to the memcached "time" parameter, so that the cache itself can reduce its size for elements which are expired (memcache seems to manage its size in any case, this is just a hint to improve its operation).
- replaced homegrown ThreadLocal implementation with threading.local, falls back to a 2.3 compat one for python<2.4

4.21 Release 1.0.3 (10/14/2008)

- Fixed os.getpid issue on GAE.
- CookieSession will add '_expires' value to data when an expire time is set, and uses it

4.22 Release 1.0.2 (9/22/2008)

- Fixed bug caused when attempting to invalidate a session that hadn't previously been created.

4.23 Release 1.0.1 (8/19/2008)

- Bug fix for cookie sessions to retain id before clearing values.

4.24 Release 1.0 (8/13/2008)

- Added cookie delete to both cookie only sessions and normal sessions, to help with proxies and such that may determine whether a user is logged in via a cookie. (cookie varies, etc.). Suggested by Felix Schwarz.
- cache.get_value() now uses the given **kwargs in all cases in the same manner as cache.set_value(). This way you can send a new createfunc to cache.get_value() each time and it will be used.

4.25 Release 0.9.5 (6/19/2008)

- Fixed bug in memcached to be tolerant of keys disappearing when memcached expires them.
- Fixed the cache functionality to actually work, previously set_value was ignored if there was already a value set.

4.26 Release 0.9.4 (4/13/2008)

- Adding 'google' backend datastore, available by specifying 'google' as the cache/session type. Note that this takes an optional table_name used to name the model class used.
- SECURITY BUG: Fixed security issue with Beaker not properly removing directory escaping characters from the session ID when un-signed sessions are used. Reported with patch by Felix Schwarz.
- Fixed bug with Beaker not playing well with Registry when its placed above it in the stack. Thanks Wichert Akkerman.

4.27 Release 0.9.3 (2/28/2008)

- Adding 'id' to cookie-based sessions for better compatibility.
- Fixed error with exception still raised for PyCrypto missing.
- WARNING: Session middleware no longer catches Paste HTTP Exceptions, apps are now expected to capture and handle Paste HTTP Exceptions themselves.
- Fixed Python 2.4 compatibility bug in hmac.
- Fixed key lookup bug on cache object to only use the settings for the key lookup. Found by Andrew Stromnov.

4.28 Release 0.9.2 (2/13/2008)

- Added option to make Beaker use a secure cookie.
- Removed CTRCipher as pycryptopp doesn't need it.
- Changed AES to use 256 bit.
- Fixed signing code to use hmac with sha for better signing security.
- Fixed memcached code to use delete_multi on clearing the keys for efficiency and updated key retrieval to properly store and retrieve None values.
- Removing cookie.py and signed cookie middleware, as the environ_key option for session middleware provides a close enough setting.
- Added option to use just cookie-based sessions without requiring encryption.
- Switched encryption requirement from PyCrypto to pycryptopp which uses a proper AES in Counter Mode.

4.29 Release 0.9.1 (2/4/2008)

- Fixed bug in middleware using module that wasn't imported.

4.30 Release 0.9 (12/17/07)

- Fixed bug in memcached replace to actually replace spaces properly.
- Fixed md5 cookie signature to use SHA-1 when available.
- Updated cookie-based session storage to use 256-bit AES-CTR mode with a SHA-1 HMAC signature. Now requires PyCrypto to use for AES scheme.
- WARNING: Moved session and cache middleware to middleware, as per the old deprecation warnings had said was going to happen for 0.8.
- Added cookie-only session storage with RC4 ciphered encryption, requires Python 2.4.
- Add the ability to specify the cookie's domain for sessions.

4.31 Release 0.8.1 (11/15/07)

- Fixed bug in database.py not properly handling missing sqlalchemy library.

4.32 Release 0.8 (10/17/07)

- Fixed bug in prior db update causing session to occasionally not be written back to the db.
- Fixed memcached key error with keys containing spaces. Thanks Jim Musil.
- WARNING: Major change to ext:database to use a single row per namespace. Additionally, there's an accessed and created column present to support easier deletion of old cache/session data. You *will* need to drop any existing tables being used by the ext:database backend.
- Streamline ext:database backend to avoid unnecessary database selects for repeat data.
- Added SQLAlchemy 0.4 support to ext:database backend.

4.33 Release 0.7.5 (08/18/07)

- Fixed data_dir parsing for session string coercions, no longer picks up None as a data_dir.
- Fixed session.get_by_id to lookup recently saved sessions properly, also updates session with creation/access time upon save.
- Add unit tests for get_by_id function. Updated get_by_id to not result in additional session files.
- Added session.get_by_id function to retrieve a session of the given id.

4.34 Release 0.7.4 (07/09/07)

- Fixed issue with Beaker not properly handling arguments as Pylons may pass them in.
- Fixed unit test to catch file removal exception.
- Fixed another bug in synchronization, this one involving reentrant conditions with file synchronization
- If a file open fails due to pickling errors, locks just opened are released unconditionally

4.35 Release 0.7.3 (06/08/07)

- Beaker was not properly parsing input options to session middleware. Thanks to Yannick Gingras and Timothy S for spotting the issue.
- Changed session to only send the cookie header if its a new session and save() was called. Also only creates the session file under these conditions.

4.36 Release 0.7.2 (05/19/07)

- Added deprecation warning for middleware move, relocated middleware to cache and session modules for backwards compatibility.

4.37 Release 0.7.1 05/18/07)

- adjusted synchronization logic to account for Mako/new Cache object's multithreaded usage of Container.

4.38 Release 0.7 (05/18/07)

- WARNING: Cleaned up Cache object based on Mako cache object, this changes the call interface slightly for creating a Cache object directly. The middleware cache object is unaffected from an end-user view. This change also avoids duplicate creations of Cache objects.
- Adding database backend and unit tests.
- Added memcached test, fixed memcached namespace arg passing.
- Fixed session and cache tests, still failing syncdict test. Added doctests for Cache and Session middleware.
- Cleanup of container/cache/container_test
- Namespaces no longer require a context, removed NamespaceContext?
- Logging in container.py uses logging module
- Cleanup of argument passing, use name **kwargs instead of **params for generic kwargs
- Container classes contain a static create_namespace() method, namespaces are accessed from the ContainerContext? via string name + container class alone

- Implemented (but not yet tested) clear() method on Cache, locates all Namespaces used thus far and clears each one based on its keys() collection
- Fixed Cache.clear() method to actually clear the Cache namespace.
- Updated memcached backend to split servers on ';' for multiple memcached backends.
- Merging MyghtyUtils code into Beaker.

4.39 Release 0.6.3 (03/18/2007)

- Added api with customized Session that doesn't require a Myghty request object, just a dict. Updated session to use the new version.
- Removing unicode keys as some dbm backends can't handle unicode keys.
- Adding core files that should've been here.
- More stringent checking for existence of a session.
- Avoid recreating the session object when it's empty.

CHAPTER

FIVE

INDICES AND TABLES

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

5.1 Module Listing

5.1.1 `beaker.cache` – Cache module

This package contains the "front end" classes and functions for Beaker caching.

Included are the `Cache` and `CacheManager` classes, as well as the function decorators `region_decorate()`, `region_invalidate()`.

Module Contents

`beaker.cache.`**`cache_regions`** = {}

Dictionary of 'region' arguments.

A "region" is a string name that refers to a series of cache configuration arguments. An application may have multiple "regions" - one which stores things in a memory cache, one which writes data to files, etc.

The dictionary stores string key names mapped to dictionaries of configuration arguments. Example:

```
from beaker.cache import cache_regions
cache_regions.update({
    'short_term':{
        'expire':'60',
        'type':'memory'
    },
    'long_term':{
        'expire':'1800',
        'type':'dbm',
        'data_dir':'/tmp',
    }
})
```

beaker.cache.**cache_region**(*region*, **args*)

Decorate a function such that its return result is cached, using a "region" to indicate the cache arguments.

Example:

```
from beaker.cache import cache_regions, cache_region

# configure regions
cache_regions.update({
    'short_term':{
        'expire':'60',
        'type':'memory'
    }
})

@cache_region('short_term', 'load_things')
def load(search_term, limit, offset):
    '''Load from a database given a search term, limit, offset.'''
    return database.query(search_term)[offset:offset + limit]
```

The decorator can also be used with object methods. The `self` argument is not part of the cache key. This is based on the actual string name `self` being in the first argument position (new in 1.6):

```
class MyThing(object):
    @cache_region('short_term', 'load_things')
    def load(self, search_term, limit, offset):
        '''Load from a database given a search term, limit, offset.'''
        return database.query(search_term)[offset:offset + limit]
```

Classmethods work as well - use `cls` as the name of the class argument, and place the decorator around the function underneath `@classmethod` (new in 1.6):

```
class MyThing(object):
    @classmethod
    @cache_region('short_term', 'load_things')
    def load(cls, search_term, limit, offset):
        '''Load from a database given a search term, limit, offset.'''
        return database.query(search_term)[offset:offset + limit]
```

Parameters

- **region** – String name of the region corresponding to the desired caching arguments, established in `cache_regions`.
- ***args** – Optional `str()`-compatible arguments which will uniquely identify the key used by this decorated function, in addition to the positional arguments passed to the function itself at call time. This is recommended as it is needed to distinguish between any two functions or methods that have the same name (regardless of parent class or not).

Note: The function being decorated must only be called with positional arguments, and the arguments must support being stringified with `str()`. The concatenation of the `str()` version of each argument, combined with that of the `*args` sent to the decorator, forms the unique cache key.

Note: When a method on a class is decorated, the `self` or `cls` argument in the first position is not included in the "key" used for caching. New in 1.6.

---

beaker.cache.**region_invalidate**(*namespace*, *region*, **args*)

Invalidate a cache region corresponding to a function decorated with `cache_region()`.

Parameters

- **namespace** – The namespace of the cache to invalidate. This is typically a reference to the original function (as returned by the `cache_region()` decorator), where the `cache_region()` decorator applies a "memo" to the function in order to locate the string name of the namespace.
- **region** – String name of the region used with the decorator. This can be `None` in the usual case that the decorated function itself is passed, not the string name of the namespace.
- **args** – Stringifyable arguments that are used to locate the correct key. This consists of the `*args` sent to the `cache_region()` decorator itself, plus the `*args` sent to the function itself at runtime.

Example:

```
from beaker.cache import cache_regions, cache_region, region_invalidate

# configure regions
cache_regions.update({
    'short_term':{
        'expire':'60',
        'type':'memory'
    }
})

@cache_region('short_term', 'load_data')
def load(search_term, limit, offset):
    '''Load from a database given a search term, limit, offset.'''
    return database.query(search_term)[offset:offset + limit]

def invalidate_search(search_term, limit, offset):
    '''Invalidate the cached storage for a given search term, limit, offset.'''
    region_invalidate(load, 'short_term', 'load_data', search_term, limit, offset)
```

Note that when a method on a class is decorated, the first argument `cls` or `self` is not included in the cache key. This means you don't send it to `region_invalidate()`:

```
class MyThing(object):
    @cache_region('short_term', 'some_data')
    def load(self, search_term, limit, offset):
        '''Load from a database given a search term, limit, offset.'''
        return database.query(search_term)[offset:offset + limit]

    def invalidate_search(self, search_term, limit, offset):
        '''Invalidate the cached storage for a given search term, limit, offset.'''
        region_invalidate(self.load, 'short_term', 'some_data', search_term, limit, offse
```

class beaker.cache.**Cache**(*namespace*, *type='memory'*, *expiretime=None*, *starttime=None*, *expire=None*, ***nsargs*)

Front-end to the containment API implementing a data cache.

Parameters

- **namespace** – the namespace of this Cache
- **type** – type of cache to use
- **expire** – seconds to keep cached data
- **expiretime** – seconds to keep cached data (legacy support)
- **starttime** – time when cache was cache was

get(*key*, ***kw*)
Retrieve a cached value from the container

clear()
Clear all the values from the namespace

class beaker.cache.**CacheManager**(***kwargs*)
Initialize a CacheManager object with a set of options

Options should be parsed with the `parse_cache_config_options()` function to ensure only valid options are used.

region(*region*, **args*)
Decorate a function to cache itself using a cache region

The region decorator requires arguments if there are more than two of the same named function, in the same module. This is because the namespace used for the functions cache is based on the functions name and the module.

Example:

```
# Assuming a cache object is available like:
cache = CacheManager(dict_of_config_options)


def populate_things():

    @cache.region('short_term', 'some_data')
    def load(search_term, limit, offset):
        return load_the_data(search_term, limit, offset)

    return load('rabbits', 20, 0)
```

Note: The function being decorated must only be called with positional arguments.

---

region_invalidate(*namespace*, *region*, **args*)
Invalidate a cache region namespace or decorated function

This function only invalidates cache spaces created with the cache_region decorator.

Parameters

- **namespace** – Either the namespace of the result to invalidate, or the cached function
- **region** – The region the function was cached to. If the function was cached to a single region then this argument can be None
- **args** – Arguments that were used to differentiate the cached function as well as the arguments passed to the decorated function

Example:

```
# Assuming a cache object is available like:
cache = CacheManager(dict_of_config_options)

def populate_things(invalidate=False):

    @cache.region('short_term', 'some_data')
    def load(search_term, limit, offset):
        return load_the_data(search_term, limit, offset)

    # If the results should be invalidated first
    if invalidate:
        cache.region_invalidate(load, None, 'some_data',
                                'rabbits', 20, 0)
    return load('rabbits', 20, 0)
```

cache (**args*, ***kwargs*)
Decorate a function to cache itself with supplied parameters

Parameters

- **args** – Used to make the key unique for this function, as in region() above.
- **kwargs** – Parameters to be passed to get_cache(), will override defaults

Example:

```
# Assuming a cache object is available like:
cache = CacheManager(dict_of_config_options)


def populate_things():

    @cache.cache('mycache', expire=15)
    def load(search_term, limit, offset):
        return load_the_data(search_term, limit, offset)

    return load('rabbits', 20, 0)
```

Note: The function being decorated must only be called with positional arguments.

---

invalidate (*func*, **args*, ***kwargs*)
Invalidate a cache decorated function

This function only invalidates cache spaces created with the cache decorator.

Parameters

- **func** – Decorated function to invalidate
- **args** – Used to make the key unique for this function, as in region() above.
- **kwargs** – Parameters that were passed for use by get_cache(), note that this is only required if a `type` was specified for the function

Example:

```
# Assuming a cache object is available like:
cache = CacheManager(dict_of_config_options)
```

```
        def populate_things(invalidate=False):

            @cache.cache('mycache', type="file", expire=15)
            def load(search_term, limit, offset):
                return load_the_data(search_term, limit, offset)

            # If the results should be invalidated first
            if invalidate:
                cache.invalidate(load, 'mycache', 'rabbits', 20, 0, type="file")
            return load('rabbits', 20, 0)
```

5.1.2 beaker.container – Container and Namespace classes

Container and Namespace classes

Module Contents

class beaker.container.**DBMNamespaceManager**(*namespace*, *dbmmodule=None*, *data_dir=None*, *dbm_dir=None*, *lock_dir=None*, *digest_filenames=True*, ***kwargs*)

Bases: beaker.container.OpenResourceNamespaceManager

NamespaceManager that uses dbm files for storage.

class beaker.container.**FileNamespaceManager**(*namespace*, *data_dir=None*, *file_dir=None*, *lock_dir=None*, *digest_filenames=True*, ***kwargs*)

Bases: beaker.container.OpenResourceNamespaceManager

NamespaceManager that uses binary files for storage.

Each namespace is implemented as a single file storing a dictionary of key/value pairs, serialized using the Python pickle module.

class beaker.container.**MemoryNamespaceManager**(*namespace*, ***kwargs*)

Bases: beaker.container.AbstractDictionaryNSManager

NamespaceManager that uses a Python dictionary for storage.

class beaker.container.**NamespaceManager**(*namespace*)

Handles dictionary operations and locking for a namespace of values.

NamespaceManager provides a dictionary-like interface, implementing __getitem__(), __setitem__(), and __contains__(), as well as functions related to lock acquisition.

The implementation for setting and retrieving the namespace data is handled by subclasses.

NamespaceManager may be used alone, or may be accessed by one or more Value objects. Value objects provide per-key services like expiration times and automatic recreation of values.

Multiple NamespaceManagers created with a particular name will all share access to the same underlying datasource and will attempt to synchronize against a common mutex object. The scope of this sharing may be within a single process or across multiple processes, depending on the type of NamespaceManager used.

The NamespaceManager itself is generally threadsafe, except in the case of the DBMNamespaceManager in conjunction with the gdbm dbm implementation.

acquire_read_lock()
Establish a read lock.

This operation is called before a key is read. By default the function does nothing.

acquire_write_lock(*wait=True*, *replace=False*)
Establish a write lock.

This operation is called before a key is written. A return value of `True` indicates the lock has been acquired.

By default the function returns `True` unconditionally.

'replace' is a hint indicating the full contents of the namespace may be safely discarded. Some backends may implement this (i.e. file backend won't unpickle the current contents).

do_remove()
Implement removal of the entire contents of this `NamespaceManager`.

e.g. for a file-based namespace, this would remove all the files.

The front-end to this method is the `NamespaceManager.remove()` method.

get_creation_lock(*key*)
Return a locking object that is used to synchronize multiple threads or processes which wish to generate a new cache value.

This function is typically an instance of `FileSynchronizer`, `ConditionSynchronizer`, or `null_synchronizer`.

The creation lock is only used when a requested value does not exist, or has been expired, and is only used by the `Value` key-management object in conjunction with a "createfunc" value-creation function.

has_key(*key*)
Return `True` if the given key is present in this `Namespace`.

keys()
Return the list of all keys.

This method may not be supported by all `NamespaceManager` implementations.

release_read_lock()
Release a read lock.

This operation is called after a key is read. By default the function does nothing.

release_write_lock()
Release a write lock.

This operation is called after a new value is written. By default this function does nothing.

remove()
Remove the entire contents of this `NamespaceManager`.

e.g. for a file-based namespace, this would remove all the files.

set_value(*key*, *value*, *expiretime=None*)
Sets a value in this `NamespaceManager`.

This is the same as `__setitem__()`, but also allows an expiration time to be passed at the same time.

class `beaker.container.`**OpenResourceNamespaceManager**(*namespace*)
Bases: `beaker.container.NamespaceManager`

A NamespaceManager where read/write operations require opening/ closing of a resource which is possibly mutexed.

class beaker.container.**Value**(*key*, *namespace*, *createfunc=None*, *expiretime=None*, *starttime=None*)

Implements synchronization, expiration, and value-creation logic for a single value stored in a NamespaceManager.

can_have_value()

clear_value()

createfunc

expire_argument

expiretime

get_value()

has_current_value()

has_value()

return true if the container has a value stored.

This is regardless of it being expired or not.

key

namespace

set_value(*value*, *storedtime=None*)

starttime

storedtime

Deprecated Classes

class beaker.container.**Container**

Implements synchronization and value-creation logic for a 'value' stored in a NamespaceManager.

Container and its subclasses are deprecated. The Value class is now used for this purpose.

class beaker.container.**ContainerMeta**(*classname*, *bases*, *dict_*)

Bases: type

class beaker.container.**DBMContainer**

Bases: beaker.container.Container

class beaker.container.**FileContainer**

Bases: beaker.container.Container

class beaker.container.**MemoryContainer**

Bases: beaker.container.Container

5.1.3 beaker.middleware – Middleware classes

Module Contents

class beaker.middleware.**CacheMiddleware**(*app*, *config=None*, *environ_key='beaker.cache'*, ***kwargs*)

Initialize the Cache Middleware

The Cache middleware will make a CacheManager instance available every request under the `environ['beaker.cache']` key by default. The location in environ can be changed by setting `environ_key`.

config dict All settings should be prefixed by 'cache.'. This method of passing variables is intended for Paste and other setups that accumulate multiple component settings in a single dictionary. If config contains *no cache. prefixed args*, then *all* of the config options will be used to intialize the Cache objects.

environ_key Location where the Cache instance will keyed in the WSGI environ

****kwargs** All keyword arguments are assumed to be cache settings and will override any settings found in `config`

class `beaker.middleware.`**SessionMiddleware**(*wrap_app*, *config=None*, *environ_key='beaker.session'*, ***kwargs*)

Initialize the Session Middleware

The Session middleware will make a lazy session instance available every request under the `environ['beaker.session']` key by default. The location in environ can be changed by setting `environ_key`.

config dict All settings should be prefixed by 'session.'. This method of passing variables is intended for Paste and other setups that accumulate multiple component settings in a single dictionary. If config contains *no cache. prefixed args*, then *all* of the config options will be used to intialize the Cache objects.

environ_key Location where the Session instance will keyed in the WSGI environ

****kwargs** All keyword arguments are assumed to be session settings and will override any settings found in `config`

5.1.4 `beaker.session` – Session classes

Module Contents

class `beaker.session.`**CookieSession**(*request*, *key='beaker.session.id'*, *timeout=None*, *cookie_expires=True*, *cookie_domain=None*, *encrypt_key=None*, *validate_key=None*, *secure=False*, *httponly=False*, ***kwargs*)

Pure cookie-based session

Options recognized when using cookie-based sessions are slightly more restricted than general sessions.

key The name the cookie should be set to.

timeout How long session data is considered valid. This is used regardless of the cookie being present or not to determine whether session data is still valid.

encrypt_key The key to use for the session encryption, if not provided the session will not be encrypted.

validate_key The key used to sign the encrypted session

cookie_domain Domain to use for the cookie.

secure Whether or not the cookie should only be sent over SSL.

httponly Whether or not the cookie should only be accessible by the browser not by JavaScript.

save(*accessed_only=False*)
Saves the data for this session to persistent storage

expire()
Delete the 'expires' attribute on this Session, if any.

delete()
Delete the cookie, and clear the session

invalidate()
Clear the contents and start a new session

class beaker.session.**Session**(*request*, *id=None*, *invalidate_corrupt=False*, *use_cookies=True*, *type=None*, *data_dir=None*, *key='beaker.session.id'*, *timeout=None*, *cookie_expires=True*, *cookie_domain=None*, *secret=None*, *secure=False*, *namespace_class=None*, *httponly=False*, ***namespace_args*)
Session object that uses container package for storage.

key The name the cookie should be set to.

timeout How long session data is considered valid. This is used regardless of the cookie being present or not to determine whether session data is still valid.

cookie_domain Domain to use for the cookie.

secure Whether or not the cookie should only be sent over SSL.

httponly Whether or not the cookie should only be accessible by the browser not by JavaScript.

save(*accessed_only=False*)
Saves the data for this session to persistent storage

If accessed_only is True, then only the original data loaded at the beginning of the request will be saved, with the updated last accessed time.

revert()
Revert the session to its original state from its first access in the request

lock()
Locks this session against other processes/threads. This is automatic when load/save is called.

use with caution and always with a corresponding 'unlock' inside a "finally:" block, as a stray lock typically cannot be unlocked without shutting down the whole application.

unlock()
Unlocks this session against other processes/threads. This is automatic when load/save is called.

use with caution and always within a "finally:" block, as a stray lock typically cannot be unlocked without shutting down the whole application.

delete()
Deletes the session from the persistent storage, and sends an expired cookie out

invalidate()
Invalidates this session, creates a new session id, returns to the is_new state

class beaker.session.**SessionObject**(*environ*, ***params*)
Session proxy/lazy creator

This object proxies access to the actual session object, so that in the case that the session hasn't been used before, it will be setup. This avoid creating and loading the session from persistent storage unless its actually used during the request.

persist()
Persist the session to the storage

If its set to autosave, then the entire session will be saved regardless of if save() has been called. Otherwise, just the accessed time will be updated if save() was not called, or the session will be saved if save() was called.

get_by_id(*id*)
Loads a session given a session ID

accessed()
Returns whether or not the session has been accessed

class beaker.session.**SignedCookie**(*secret*, *input=None*)
Extends python cookie to give digital signature support

5.1.5 beaker.synchronization – Synchronization classes

Synchronization functions.

File- and mutex-based mutual exclusion synchronizers are provided, as well as a name-based mutex which locks within an application based on a string name.

Module Contents

class beaker.synchronization.**ConditionSynchronizer**(*identifier*)
a synchronizer using a Condition.

class beaker.synchronization.**FileSynchronizer**(*identifier*, *lock_dir*)
A synchronizer which locks using flock().

class beaker.synchronization.**NameLock**(*identifier=None*, *reentrant=False*)
a proxy for an RLock object that is stored in a name based registry.

Multiple threads can get a reference to the same RLock based on the name alone, and synchronize operations related to that name.

class beaker.synchronization.**null_synchronizer**
A 'null' synchronizer, which provides the SynchronizerImpl interface without any locking.

class beaker.synchronization.**SynchronizerImpl**
Base class for a synchronization object that allows multiple readers, single writers.

5.1.6 beaker.util – Beaker Utilities

Beaker utilities

Module Contents

beaker.util.**encoded_path**(*root*, *identifiers*, *extension='.enc'*, *depth=3*, *digest_filenames=True*)
Generate a unique file-accessible path from the given list of identifiers starting at the given root directory.

beaker.util.**func_namespace**(*func*)
Generates a unique namespace for a function

class beaker.util.**SyncDict**
An efficient/threadsafe singleton map algorithm, a.k.a. "get a value based on this key, and create if not found or not valid" paradigm:

exists && isvalid ? get : create

Designed to work with weakref dictionaries to expect items to asynchronously disappear from the dictionary.

Use python 2.3.3 or greater ! a major bug was just fixed in Nov. 2003 that was driving me nuts with garbage collection/weakrefs in this section.

class beaker.util.**ThreadLocal**
stores a value on a per-thread basis

beaker.util.**verify_directory**(*dir*)
verifies and creates a directory. tries to ignore collisions with other threads and processes.

beaker.util.**parse_cache_config_options**(*config*, *include_defaults=True*)
Parse configuration options and validate for use with the CacheManager

5.1.7 beaker.ext.database – Database Container and NameSpace Manager classes

Module Contents

class beaker.ext.database.**DatabaseContainer**

class beaker.ext.database.**DatabaseNamespaceManager**(*namespace*, *url=None*, *sa_opts=None*, *optimistic=False*, *table_name='beaker_cache'*, *data_dir=None*, *lock_dir=None*, ***params*)
Creates a database namespace manager

url SQLAlchemy compliant db url

sa_opts A dictionary of SQLAlchemy keyword options to initialize the engine with.

optimistic Use optimistic session locking, note that this will result in an additional select when updating a cache value to compare version numbers.

table_name The table name to use in the database for the cache.

5.1.8 beaker.ext.google – Google Container and NameSpace Manager classes

Module Contents

class beaker.ext.google.**GoogleContainer**

class beaker.ext.google.**GoogleNamespaceManager**(*namespace*, *table_name='beaker_cache'*, ***params*)
Creates a datastore namespace manager

5.1.9 beaker.ext.memcached – Memcached Container and NameSpace Manager classes

Module Contents

class beaker.ext.memcached.**MemcachedContainer**
Bases: beaker.container.Container

Container class which invokes MemcacheNamespaceManager.

class beaker.ext.memcached.**MemcachedNamespaceManager**(*namespace*, *url*, *memcache_module='auto'*, *data_dir=None*, *lock_dir=None*, ***kw*)
Bases: beaker.container.NamespaceManager

Provides the NamespaceManager API over a memcache client library.

class beaker.ext.memcached.**PyLibMCNamespaceManager**(**arg*, ***kw*)
Bases: beaker.ext.memcached.MemcachedNamespaceManager

Provide thread-local support for pylibmc.

5.1.10 beaker.ext.sqla – SqlAlchemy Container and NameSpace Manager classes

Module Contents

beaker.ext.sqla.**make_cache_table**(*metadata*, *table_name='beaker_cache'*)
Return a Table object suitable for storing cached values for the namespace manager. Do not create the table.

class beaker.ext.sqla.**SqlaContainer**

class beaker.ext.sqla.**SqlaNamespaceManager**(*namespace*, *bind*, *table*, *data_dir=None*, *lock_dir=None*, ***kwargs*)
Create a namespace manager for use with a database table via SQLAlchemy.

bind SQLAlchemy Engine or Connection object

table SQLAlchemy Table object in which to store namespace data. This should usually be something created by make_cache_table.

5.1.11 beaker.crypto.pbkdf2 – PKCS#5 v2.0 Password-Based Key Derivation classes

Module Contents

beaker.crypto.pbkdf2.**crypt**(*word*, *salt=None*, *iterations=None*)
PBKDF2-based unix crypt(3) replacement.

The number of iterations specified in the salt overrides the 'iterations' parameter.

The effective hash length is 192 bits.

class beaker.crypto.pbkdf2.**PBKDF2**(*passphrase*, *salt*, *iterations=1000*, *digestmodule=<module 'Crypto.Hash.SHA' from '/usr/lib/python2.7/dist-packages/Crypto/Hash/SHA.pyc'>*, *macmodule=<module 'Crypto.Hash.HMAC' from '/usr/lib/python2.7/dist-packages/Crypto/Hash/HMAC.pyc'>*)

PBKDF2.py : PKCS#5 v2.0 Password-Based Key Derivation

This implementation takes a passphrase and a salt (and optionally an iteration count, a digest module, and a MAC module) and provides a file-like object from which an arbitrarily-sized key can be read.

If the passphrase and/or salt are unicode objects, they are encoded as UTF-8 before they are processed.

The idea behind PBKDF2 is to derive a cryptographic key from a passphrase and a salt.

PBKDF2 may also be used as a strong salted password hash. The 'crypt' function is provided for that purpose.

Remember: Keys generated using PBKDF2 are only as strong as the passphrases they are derived from.

close()
Close the stream.

hexread(*octets*)
Read the specified number of octets. Return them as hexadecimal.

Note that len(obj.hexread(n)) == 2*n.

read(*bytes*)
Read the specified number of key bytes.