python-camlistore

Release dev

September 05, 2016

Contents

`python-camlistore` is a Python client library for Camlistore, which aims to be "your personal storage system for life".

At present this client library is pretty experimental, much like Camlistore itself. It provides a basic interface to the low-level blob storage system and to parts of the search system, but with an interface that will probably change as Camlistore evolves and as the client interface design is refined.

Contents:

CHAPTER 1

Getting Started

1.1 Install The Module

This is a standard Python module and can be installed using `pip` as usual:

- `pip install camlistore`

1.2 Connect To Camlistore

Before this library will be of any use you will need a Camlistore server running. See the Camlistore docs for more details.

Once you have a server running you can connect to it using *`camlistore.connect()`*. For example, if you have your Camlistore server running on localhost:

```
import camlistore

conn = camlistore.connect("http://localhost:3179/")
```

This function will contact the specified URL to make sure it looks like a valid Camlistore server and discover some details about its configuration. The `conn` return value is then a *`camlistore.Connection`* object, configured and ready to access the server.

1.3 Try Writing A Blob!

To test if we've connected successfully, we can try some simple calls to write a blob and retrieve it again:

```
blobref = conn.blobs.put(camlistore.Blob("Hello, Camlistore!"))
hello_blob = conn.blobs.get(blobref)
print hello_blob.data
```

If things are working as expected, this should print out `Hello, Camlistore!`, having successfully written that string into the store and retrieved it again. You're now ready to proceed to the following sections to learn more about the blob store and search interfaces.

This program will fail if the connection is not configured properly. For example, it may fail if the Camlistore server requires authentication, since our example does not account for that.

1.4 Connection Interface Reference

camlistore.**connect**(*base_url*)

Create a connection to the Camlistore instance at the given base URL.

This function implements the Camlistore discovery protocol to recognize a server and automatically determine which features are available, ultimately instantiating and returning a `Connection` object.

For now we assume an unauthenticated connection, which is generally only possible when connecting via `localhost`. In future this function will be extended with some options for configuring authentication.

class camlistore.**Connection**(*http_session=None*, *blob_root=None*, *search_root=None*, *sign_root=None*)

Represents a logical connection to a camlistore server.

Most callers should not instantiate this directly, but should instead use `connect()`, which implements the Camlistore server discovery protocol to auto-configure an instance of this class.

Note that this does not imply a TCP or any other kind of socket connection, but merely some persistent state that will be used when making requests to the server. In particular, several consecutive requests via the same connection may be executed via a single keep-alive HTTP connection, reducing round-trip time.

CHAPTER 2

Accessing the Blob Store

The lowest-level interface in Camlistore is the raw blob store, which provides a mechanism to store and retrieve immutable objects. All other Camlistore functionality is built on this base layer.

Blob store functionality is accessed via `camlistore.Connection.blobs`, which is a pre-configured instance of *camlistore.blobclient.BlobClient*.

class `camlistore.blobclient.`**BlobClient**(*http_session*, *base_url*)
Low-level interface to Camlistore's blob store interface.

The blob store is the lowest-level Camlistore API and provides only for inserting and retrieving immutable, content-addressed blobs.

All of the functionality of Camlistore builds on this abstraction, but most use-cases are better served by the *search* interface, which can be accessed via `camlistore.Connection.searcher`.

Callers should not instantiate this class directly. Instead, call *camlistore.connect()* to obtain a *camlistore.Connection* object and access `camlistore.Connection.blobs`.

blob_exists(*blobref*)
Determine if a blob exists with the given blobref.

Returns *True* if the blobref is known to the server, or *False* if it is not.

To more efficiently test the presence of many blobs at once, it's better to use `get_size_multi()`; known blobs will have a size, while unknown blobs will indicate `None`.

enumerate()
Enumerate all of the blobs on the server, in blobref order.

Returns an iterable over all of the blobs. The underlying server interface returns the resultset in chunks, so beginning iteration will cause one request but continued iteration may cause followup requests to retrieve additional chunks.

Most applications do not need to enumerate all blobs and can instead use the facilities provided by the search interface. The enumeration interface exists primarily to enable the Camlistore indexer to build its search index, but may be useful for other alternative index implementations.

get(*blobref*)
Get the data for a blob, given its blobref.

Returns a *camlistore.Blob* instance describing the blob, or raises *camlistore.exceptions.NotFoundError* if the given blobref is not known to the server.

get_size(*blobref*)
Get the size of a blob, given its blobref.

Returns the size of the blob as an `int` in bytes, or raises *camlistore.exceptions.NotFoundError* if the given blobref is not known to the server.

get_size_multi(**blobrefs*)
Get the size of several blobs at once, given their blobrefs.

This is a batch version of `get_size()`, returning a mapping object whose keys are the request blobrefs and whose values are either the size of each corresponding blob or `None` if the blobref is not known to the server.

put(*blob*)
Write a single blob into the store.

The blob must be given as a *camlistore.Blob* instance. Returns the blobref of the created blob, which is guaranteed to match *blob.blobref* of the given blob.

This function will first check with the server to see if it has the given blob, so it is not necessary for the caller to check for the existence of the blob before uploading.

When writing many blobs at once – a more common occurence than just one in most applications – it is more efficient to use `put_multi()`, since it is able to batch-upload blobs and reduce the number of round-trips required to complete the operation.

put_multi(**blobs*)
Upload several blobs to the store.

This is a batch version of `put()`, uploading several blobs at once and returning a list of their blobrefs in the same order as they were provided in the arguments.

At present this method does *not* correctly handle the protocol restriction that only 32MB of data can be uploaded at once, so this function will fail if that limit is exceeded. It is intended that this will be fixed in a future version.

class camlistore.**Blob**(*data*, *hash_func_name='sha1'*, *blobref=None*)
Represents a blob.

A blob is really just a raw string of bytes, but this class exists to provide a convenient interface to make a blob and find its blobref and size.

Although blobs are not mutable, instances of this class *are*. Mutating instances of this class (by assigning to `Blob.data` or `Blob.hash_func_name`) will change the blob's blobref, causing it to be a different blob as far as Camlistore is concerned, although it remains the same object as far as Python is concerned.

Most callers should not pass a `blobref` argument to the initializer, since it can be computed automatically from the other arguments. If one *is* provided, it *must* match the provided data or else the *camlistore.exceptions.HashMismatchError* exception will be raised, allowing callers to check for a hash mismatch as a side-effect. If a blobref is provided, its hash function overrides the value passed in as `hash_func_name`.

blobref
The blobref of this blob.

This value will change each time either `data` or `hash_func_name` is modified, so callers should be careful about caching this value in a local variable if modifications are expected.

data
The raw blob data, as a `str`.

Assigning to this property will change `blobref`, and effectively create a new blob as far as the server is concerned.

hash_func_name
The name of the hash function to use for this blob's blobref.

This must always be the name of a function that is supported by both the local `hashlib` *and* the Camlistore server. `"sha1"` is currently a safe choice for compatibility, and is thus the default. `"sha256"` will also work with the implementations available at the time of writing.

Assigning to this property will change `blobref`, and effectively create a new blob as far as the server is concerned.

size
The size of the blob data, in bytes.

class `camlistore.blobclient.`**BlobMeta**(*blobref*, *size=None*, *blob_client=None*)
Metadata about a blob.

This is essentially a *camlistore.Blob* object without the blob's data, for situations where we have the identity of a blob but have not yet retrieved it.

Callers should not instantiate this class directly. It's intended only to be used as the return value of methods on `BlobClient`.

get_data()
Retrieve the blob described by this object.

This will call to the server to obtain the given blob, with the same behavior as `BlobClient.get()`.

CHAPTER 3

Accessing the Search Interface

Most interesting functionality of Camlistore is implemented in terms of the search interface, which is provided by Camlistore's indexer. It visits all of the stored blobs and discovers higher-level relationships between them based on its understanding of various schemas.

Search functionality is accessed via `camlistore.Connection.searcher`, which is a pre-configured instance of *camlistore.searchclient.SearchClient*.

class `camlistore.searchclient.`**`SearchClient`**(*http_session*, *base_url*)

Low-level interface to Camlistore indexer search operations.

The indexer component visits all blobs in the store and infers connections between them based on its knowledge of certain schema formats.

In particular, the indexer is responsible for tracking all of the modification claims for a permanode and providing its flattened attribute map for any given point in time.

However, the indexer also has special knowledge of the conventions around storage of filesystems and can thus be a more convenient interface for filesystem traversal than the raw blob interface.

Callers should not instantiate this class directly. Instead, call *camlistore.connect()* to obtain a *camlistore.Connection* object and access `camlistore.Connection.searcher`.

`describe_blob`(*blobref*)

Request a description of a particular blob, returning a `BlobDescription` object.

The "description" of a blob is the indexer's record of the blob, so it contains only the subset of information retained by the indexer. The level of detail in the returned object will thus depend on what the indexer knows about the given object.

`get_claims_for_permanode`(*blobref*)

Get the claims for a particular permanode, as an iterable of `ClaimMeta`.

The concept of "claims" is what allows a permanode to appear mutable even though the underlying storage is immutable. The indexer processes each of the valid claims on a given permanode to produce an aggregated set of its attributes for a given point in time.

Most callers should prefer to use `describe_blob()` instead, since that returns the flattened result of processing all attributes, rather than requiring the client to process the claims itself.

`query`(*expression*)

Run a query against the index, returning an iterable of `SearchResult`.

The given expression is just passed on verbatim to the underlying query interface.

Query constraints are not yet supported.

3.1 Get Blob Descriptions

One important feature of search interface is its ability to obtain the current state of a mutable object, or even its state at a particular point in time.

camlistore.searchclient.SearchClient.describe_blob() takes a blobref and returns a *camlistore.searchclient.BlobDescription* object that provides access to the index metadata for the given blob, as well as efficient access to descriptions of related objects.

class camlistore.searchclient.**BlobDescription**(*searcher*, *raw_dict*, *other_raw_dicts={}*)
Represents the indexer's description of a blob, from `SearchClient.describe_blob()`.

blobref
The blobref of the blob being described.

describe_another(*blobref*)
Obtain a description of another related blob.

When asked for a description, the indexer also returns descriptions of some of the objects related to the requested object, such as the files in a directory.

This interface allows callers to retrieve related objects while possibly making use of that already-retrieved data, falling back on a new call to the indexer if the requested blob was not already described.

Since this method sometimes uses data retrieved earlier, it may return stale data. If the latest data is absolutely required, prefer to call directly `SearchClient.describe_blob()`.

size
The indexer's idea of the size of the blob.

type
The indexer's idea of the type of the blob.

3.2 Execute Search Queries

The other main capability of the search interface is querying the store to find objects fitting certain criteria.

camlistore.searchclient.SearchClient.query is the interface to this functionality, returning an iterable of *camlistore.searchclient.SearchResult* objects.

class camlistore.searchclient.**SearchResult**(*blobref*)
Represents a search result from `SearchClient.query()`.

3.3 Access Raw Permanode Claims

The mechanism by which Camlistore implements mutable objects is via *permanodes* which act as an immitable persistent "name" for a mutable object, and *claims* which describe mutations of permanodes.

Most callers will use the flattened list of permanode attributes provided by *camlistore.searchclient.SearchClient.describe_blob()*, but it is also possible to access the raw claim list for a permanode via *camlistore.searchclient.SearchClient.get_claims_for_permanode()*, which returns an iterable of *camlistore.searchclient.ClaimMeta* objects.

class camlistore.searchclient.**ClaimMeta**(*raw_dict*)
Description of a claim.

A claim is a description of a mutation against a permanode. The indexer aggregates claims to decide the state of a permanode for a given point in time.

The `type` attribute represents the kind of mutation, and a different subset of the other attributes will be populated depending on that type.

attr
: For claims that mutate attributes, the name of the attribute that this claim mutates, as a string.

blobref
: The blobref of the underlying claim object.

permanode_blobref
: The blobref of the permanode to which this claim applies.

signer_blobref
: The blobref of the public key of the party that made this claim, against which the claim's signature was verified.

target_blobref
: For claim types that have target blobs, the blobref of the claim's target.

time
: The time at which the claim was made, as a :py:class:datetime.datetime:. The timestamps of claims are used to order them and to allow the indexer to decide the state of a permanode on any given date, by filtering later permanodes.

type
: The type of mutation being performed by this claim.

value
: For claims that mutate attributes, the value applied to the mutation.

CHAPTER 4

Error Types

Operations can result in various different errors of the types listed below.

exception camlistore.exceptions.**ConnectionError**
There was some kind of error while establishing an initial connection to a Camlistore server.

exception camlistore.exceptions.**HashMismatchError**
There was a mismatch between an expected hash value an an actual hash value.

exception camlistore.exceptions.**NotCamliServerError**
When attempting to connect to a Camlistore server it was determined that the given resource does not implement the Camlistore protocol, and is thus assumed not to be a Camlistore server.

exception camlistore.exceptions.**NotFoundError**
The requested object was not found on the server.

exception camlistore.exceptions.**ServerError**
The server returned an unexpected error in response to some operation.

exception camlistore.exceptions.**ServerFeatureUnavailableError**
The server does not implement the requested feature.

This can occur if e.g. a particular server is running a blob store but is not running an indexer, and a caller tries to use search features.

CHAPTER 5

Indices and tables

- genindex
- search

C