Couchbase Client Library: Python 1.0-Beta

Couchbase

Couchbase Client Library: Python 1.0-Beta

Abstract

This is the manual for the 1.0-beta version of the Couchbase Python client library, which is compatible with Couchbase Server 2.0+.

This manual provides a reference to the key features and best practice for using the Python Couchbase Client Library (`couchbase`). The content constitutes a reference to the core API, not a complete guide to the entire API functionality.

External Community Resources.

Python Client Library
SDK Forum

Note. The following document is still in production, and is not considered complete or exhaustive.

Last document update: 24 Jun 2013 16:03; *Document built*: 24 Jun 2013 16:03.

Documentation Availability and Formats. This documentation is available *online: HTML Online* . For other documentation from Couchbase, see Couchbase Documentation Library

Contact: editors@couchbase.com or couchbase.com

Table of Contents

1. Getting Started 1
    1.1. Download and Installation 1
    1.2. Hello Couchbase 1
    1.3. Working With Documents 3
        1.3.1. Storing Documents 3
        1.3.2. Getting Documents By Key 6
        1.3.3. Getting Documents by Querying Views 7
        1.3.4. Encoding and Serialization 8
2. Tutorial 11
    2.1. Quickstart 11
    2.2. Preparations 11
        2.2.1. Project Setup 12
        2.2.2. Preparing the Views 12
        2.2.3. Structure of the Flask Application 13
    2.3. The Welcome Page 14
    2.4. Managing Beers 16
        2.4.1. Showing Beers 16
        2.4.2. Deleting Beers 18
        2.4.3. Displaying Beers 19
        2.4.4. Editing Beers 20
        2.4.5. Creating Beers 22
        2.4.6. Searching Beers 22
    2.5. Managing Breweries 24
    2.6. Wrapping Up 24
        2.6.1. Food For Thought 24
3. Using the APIs 25
    3.1. Connecting 25
        3.1.1. Multiple Nodes 25
        3.1.2. Timeouts 25
        3.1.3. SASL Buckets 25
        3.1.4. Threads 25
    3.2. API Return Values 26
    3.3. Storing Data 26
        3.3.1. Setting Values 26
        3.3.2. Arithmetic and Counter Operations 26
        3.3.3. Append and Prepend Operations 27
        3.3.4. Expiration Operations 28
    3.4. Deleting Data 28
    3.5. Retrieving Data 28
    3.6. Locking Data/Ensuring Consistency 29
        3.6.1. Opportunistic Locking 29
        3.6.2. Pessimistic Locking 30
    3.7. Working With Views 31
        3.7.1. Common View Parameters 32
        3.7.2. Pagination 33
    3.8. Design Document Management 33
4. Advanced Usage 35
    4.1. Batched (Bulk) Operations 35
        4.1.1. Exceptions in Batched Operations 35
    4.2. Using With and Without Threads 35
    4.3. Custom Encodings/Conversions 36

4.3.1. Formats and Flags 36
4.3.2. Custom `Transcoder` Objects 36
4.3.3. Bypassing Conversion 37
4.4. Logging and Debugging 37
4.4.1. Components 37
4.4.2. Exception Handling 38
4.4.3. Application Crashes 39
5. Contributing 41
5.1. General Information 41
5.1.1. Compiling Python From Source 41
5.1.2. Running Tests 42
5.1.3. Building Docs 42
5.2. Source Style Guidelines 42
A. Release Notes 45
A.1. Release Notes for Couchbase Client Library Python 0.8.0 Beta (1 September 2012) 45
A.2. Release Notes for Couchbase Client Library Python 0.7.1 Beta (6 August 2012) 45
A.3. Release Notes for Couchbase Client Library Python 0.7.0 Beta (6 August 2012) 45
B. Licenses 48
B.1. Documentation License 48

List of Examples

1.1. CRUD Example 5

Chapter 1. Getting Started

The following chapters will demonstrate how to get started quickly using Couchbase with the Python SDK. We'll first show how to install the SDK and then demonstrate how it can be used to perform some simple operations.

1.1. Download and Installation

Follow and install these packages to get started wih using the Python SDK

1. Get, Install, and Start Couchbase server. Come back here when you are done.

2. Get and install the C library. Note that for Windows users, starting with version 1.0 Beta, the C library is bundled with the Python SDK, so you may skip this step.

3. Check your Python version. It should be at least version 2.6 (Python versions 3.x are supported as well). To check your python version:

```
shell> python -V
python 2.6.6.
```

4. Install the Python SDK. The easiest way to do this is via the `pip` tool.

   Simply invoke

```
shell> pip install couchbase --quiet
```

   If all went well, you should not see any errors printed to the screen.

   Alternatively, you may also manually download one of the packages at PyPi

5. Verify your Python SDK is available and working

```
shell> python -c 'import couchbase'
```

If this does not print any errors or exceptions, your Python SDK is properly installed!

1.2. Hello Couchbase

To follow the tradition of programming tutorials, we'll start with "Hello Couchbase". Note that this example expects you to have installed the "beer-sample" bucket (which is provided with the default install).

hello-couchbase.py.

```
from couchbase import Couchbase
from couchbase.exceptions import CouchbaseError

c = Couchbase.connect(bucket='beer-sample', host='localhost')

try:
    beer = c.get("aass_brewery-juleol")

except CouchbaseError as e:
    print "Couldn't retrieve value for key", e
    # Rethrow the exception, making the application exit
    raise

doc = beer.value

# Because Python 2.x will complain if an ASCII format string is used
# with Unicode format values, we make the format string unicode as
```

```
# well.

print unicode("{name}, ABV: {abv}").format(name=doc['name'], abv=doc['abv'])

doc['comment'] = "Random beer from Norway"

try:
    result = c.replace("aass_brewery-juleol", doc)
    print result

except CouchbaseError as e:
    print "Couldn't replace key"
    raise
```

While this code should be simple, we'll explain each step in greater detail:

- **Connecting**

  The `Couchbase.connect` class method constructs a new `couchbase.connection.Connection` object. This object represents a connection to a single bucket within the cluster. Arguments passed to `connect` are passed to the constructor (see API documentation on the `Connection` object for more details and options).

  A bucket represents a logical namespace for a key. All keys must be unique within a single bucket, but multiple buckets can have keys with the same names (and they will not conflict). A new connection object must be created for each bucket you wish to interact with in your application. Here we are creating one connection to the `beer-sample` bucket.

  The constructor is passed the bucket name (which is `beer-sample`), and a node on the cluster to connect to. Note that you can pass any node that is a member of the cluster. In this case, I'm using my local cluster instance.

- **Retrieving Data**

  The `get` method retrieves the value for the key requested. If the key exists, a `Result` object is returned containing the value of the key as well as additional metadata. To get the actual value of the object, you can access the `Result` object's `value` property.

  Note that if the key does not exist on the server, an exception of type `CouchbaseError` is thrown. This exception object can be caught and examined or printed to see more details about why the operation failed. See the API documentation for more details.

  Note that we treat the `value` as a `dict` object. As a documented oriented database, values stored to the server are considered to be JSON by default, and when retrieve from the server are interpreted to be JSON (and unserialized into a Python dict). It is possible to use other formats than the default JSON, however. The `set` methods accept a `format` keyword argument which indicates the conversion type to be used. The default is `couchbase.FMT_JSON`, but you may also use `couchbase.FMT_BYTES`, `couchbase.FMT_UTF8`, or `couchbase.FMT_PICKLE` instead. If none of these are sufficient, you may even write your own custom `Transcoder` object to handle conversion on your own.

- **Storing Data**

  To store documents in the server, you can use one of the `set` family of methods. Here we use `replace` which enforces the constraint that a previous value of the document must already exist. This can be thought of as an *update* operation in terms of *C.R.U.D.* (Create, Read, Update, Delete).

  The storage methods also return a `Result` object containing metadata about the value stored.

Now we're ready to run our first Couchbase Program:

```
shell> python hello-couchbase.py
```

```
Juleøl, ABV: 5.9
OperationResult<RC=0x0, Key=aass_brewery-juleol, CAS=0x428e4317cfe60000>
```

The first line outputs the *name* field of the document, and the second line outputs the `Result` object of the replace operation.

1.3. Working With Documents

A document in Couchbase server consists of a *key*, *value*, and *metadata*. We will explain the following briefly

- **Key**

  A key is a unique identifier for your data. Each document must have its unique key. The key can be any valid unicode string.

- **Value**

  The value is your own application data which exists under the key. The format of the value can be anything. By default, only JSON-serializable object are supported (that is, Python `str`, `unicode`, `dict`, `list`, `tuple`, `int`, `long`, `float`, `bool`, and `None` types) - in short, anything that the standard `json.dumps` will accept. The reason JSON is the default format is for the ability to later query the database based on value contents, as will be explained later.

  Note that it is possible to also store arbitrary Python objects using the `FMT_PICKLE` value for the `format` option.

- **Metadata**

  This contains information concerning the format of the value (e.g. whether it's JSON, Pickle, or something else). It also contains revision information - such as the *CAS*, which we'll read about later.

You can *store* documents by providing the unique *key* under which the document will be stored, and the *value* which contains the actual document. You can *retrieve* documents either by directly specifying the unique *key* under which the document was stored, or by querying *views* which will retrieve information about documents based on specific *criteria* - which will yield the documents that match it.

1.3.1. Storing Documents

This section provides a bit more insight in how to store documents. This is a prerequisite to demonstrate how to retrieve documents (as there must be something to retrieve)

> **Note**
>
> There are additional storage methods beyond those described here, which are covered in the Advanced section. These include manipulating numeric counters, setting expiration times for documents, and appending/prepending to existing values.

The `Connection` object has three different store operations which conform to the *CRUD* model:

- `set(key, value)`

  This stores the document `value` under the key `key`. If the key did not previously exist, it is created. If the key already exists, its existing value is overwritten with the new contents of `value`.

- `add(key, value)`

  This stores the document `value` under the key `key`, but only if `key` does *not already exist*. If `key` already exists, an exception is thrown.

- `replace(key, value)`

This is the inverse of `add`. This will set the contents of `key` to `value`, but only if the *key already exists*. If the key does not already exist, an exception is thrown.

- `delete(key)`

  Deletes the key `key` from the bucket. Future attempts to access this key via `get` will raise an exception until something is stored again for this key using one of the `set` methods.

```
from couchbase import Couchbase
from couchbase.exceptions import CouchbaseError

key = "demo_key"
value = "demo_value"

# We use the 'default' bucket.
c = Couchbase.connect(bucket='default', host='localhost')

print "Setting key {0} with value {1}".format(key, value)
result = c.set(key, value)
print "...", result

print ""
print "Getting value for key {0}".format(key)
result = c.get(key)
print "...", result

print ""
print "Creating new key {0} with value 'new_value'".format(key)
print "This will fail as '{0}' already exists".format(key)
try:
    c.add(key, "another value")
except CouchbaseError as e:
    print e

print "Replacing existing key {0} with new value".format(key)
result = c.replace(key, "new value")
print "...", "result"

print ""
print "Getting new value for key {0}".format(key)
result = c.get(key)
print "...", result

print ""
print "Deleting key", key
result = c.delete(key)
print "...", result

print ""
print "Getting value for key {0}. This will fail as it has been deleted".format(key)
try:
    c.get(key)
except CouchbaseError as e:
    print e

print ""
print "Creating new key {0} with value 'added_value'".format(key)
result = c.add(key, "added_value")
print "...", result

print "Getting the new value"
Setting key demo_key with value demo_value
... OperationResult<RC=0x0, Key=demo_key, CAS=0x3222e0f096e80000>

Getting value for key demo_key
... ValueResult<RC=0x0, Key=demo_key, Value=u'demo_value', CAS=0x3222e0f096e80000, Flags=0x0>

Creating new key demo_key with value 'new_value'
This will fail as 'demo_key' already exists
<Key=u'demo_key', RC=0xC[Key exists (with a different CAS value)], Operational Error, Results=1, C Source=(src/multire
Replacing existing key demo_key with new value
... result

Getting new value for key demo_key
... ValueResult<RC=0x0, Key=demo_key, Value=u'new value', CAS=0xbff8f2f096e80000, Flags=0x0>

Deleting key demo_key
... OperationResult<RC=0x0, Key=demo_key, CAS=0xc0f8f2f096e80000>

Getting value for key demo_key. This will fail as it has been deleted
<Key=u'demo_key', RC=0xD[No such key], Operational Error, Results=1, C Source=(src/multiresult.c,147)>

Creating new key demo_key with value 'added_value'
... OperationResult<RC=0x0, Key=demo_key, CAS=0x366a05f196e80000>
Getting the new value
... ValueResult<RC=0x0, Key=demo_key, Value=u'added_value', CAS=0x366a05f196e80000, Flags=0x0>
```

1.3.2. Getting Documents By Key

Couchbase allows two ways to fetch your documents: You can retrieve a document by its *key*, or you can retrieve a set of documents which match some constraint using Views. Since views are more complex, we'll first demonstrate getting documents by their keys.

To get a single document, simply supply the key as the first argument to the `get` method. It will return a `Result` object on success which can then be used to extract the value.

Getting A Single Document.

```
client.store("my list", [])
result = client.get("my list")
doc = result.value
```

To get multiple documents, you may use the more efficient `get_multi` method. It is passed an iterable sequence of keys, and returns a dict-like object (this is actually a dict subclass called `MultiResult`) with the keys passed to `get_multi` as keys, and the values being a `Result` object for the result of each key.

Getting Multiple Documents.

```
client.set_multi({
    'sheep_counting' : ['first sheep', 'second sheep'],
    'famous_sheep' : {'sherry lewis' : 'Lamb Chops'}
})

keys = ('sheep_counting', 'famous_sheep')
results = client.get_multi(keys)
for key, result in results.items():
    doc = result.value
```

Error Handling

Note that if a document does not exist, a `couchbase.exceptions.NotFoundError` (which is a subclass of `couchbase.exceptions.CouchbaseError` is thrown).

You can change this behavior by using the `quiet` keyword parameter and setting it to true (to suppress exceptions for a specific `get` call) or by setting the `Connection.quiet` property on the `Connection` object (which will supress exceptions on `get` for subsequent calls).

When using `quiet`, you can still determine if a key was retrieved successfuly by examining the `success` property of the value object

Passing `quiet` to `get`.

```
result = client.get("non-exist-key", quiet=True)
if result.success:
    print "Got document OK"
else:
    print ("Couldn't retrieve document. "
           "Result was received with code"), result.rc
```

Or

Setting `quiet` in the constructor.

```
client = Couchbase.connect(bucket='default', quiet=True)
result = client.get("non-exist-key")
if result.success:
    print "Got document OK"
else:
    print "Couldn't retrieve document"
```

The `rc` property of the `Result` object contains the error code received on failure (on success, its value is `0`). You can also obtain the exception class which would have been thrown by using

```
>>> CouchbaseError.rc_to_exctype(result.rc)
<class 'couchbase.exceptions.NotFoundError'>
```

This class method is passed an error code and produces the appropriate exception class.

Note that on `get_multi` with the quiet option enabled, you can immediately determine if all the keys were fetched successfully or not by examining the returned `MultiResult` 's `all_ok` property.

```
results = client.get_multi(("i exist", "but i don't"), quiet=True)
if not results.all_ok:
    print "Couldn't get all keys"
```

1.3.3. Getting Documents by Querying Views

In addition to fetching documents by keys, you may also employ *Views* to retrieve information using secondary indexes. This guide gets you started on how to use them from the Python SDK. If you want to learn more about views, see the chapter in the Couchbase Server 2.0 documentation

First, create your view definition using the web UI (though you may also do this directly from the Python SDK, as will be shown later).

You can then query the view results by calling the `query` method on the `Connection` object. Simply pass it the design and view name.

```
view_results = client.query("beer", "brewery_beers")
for result in view_results:
    print "Mapped key: %r" % (result.key,)
    print "Emitted value: %r" % (result.value,)
    print "Document ID: %s" % (result.docid,)
```

The `query` method returns a `couchbase.views.iterator.View` object which is an iterator. You may simply iterate over it to retrieve the results for the query. Each object yielded is a `ViewRow` which is a simple object containing the key, value, document ID, and optionally the document itself for each of the results returned by the view.

In addition to passing the design and view name, the `query` method accepts additional keyword arguments which control the behavior of the results returned. You may thus use it like so:

```
results = client.query("beer", "brewery_beers", opt1=value1, opt2=value2, ...)
for result in results:
    # do something with result..
```

Here are some of the available parameters for the `query` method. A full listing may be found in the API documentation.

- `include_docs`

  This boolean parameter indicates whether the corresponding document should be retrieved for each row fetched. If this is true, the `doc` property of the `ViewRow` object yielded by the iterator returned by `query` will contain a `Result` object containing the document for the key.

- `reduce`

  This boolean parameter indicates whether the server should also pass the results to the view's `reduce` function. An exception is raised if the view does not have a `reduce` method defined.

- `limit`

  This numeric parameter indicates the maximum amount of results to fetch from the query. This is handy if your query can produce a lot of results

- `descending`

This boolean parameter indicates that the results should be returned in reversed (descending) order.

- `stale`

  This boolean parameter can be used to control the tradeoff between performance and freshness of data.

- `debug`

  This boolean parameter will also fetch low-level debugging information from the view engine.

- `streaming`

  This boolean parameter indicates whether the view results should be decoded in a *streaming* manner. When enabled, the iterator will internally fetch chunks of the response as required.

As this is less efficient than fetching all results at once, it is disabled by default, but can be very useful if you have a large dataset as it prevents the entire view from being buffered in memory.

```
results = client.query("beer", "brewery_beers",
                       include_docs=True, limit=5)

for result in results:
    print "key is %r" % (result.key)
    doc = result.doc.value
    if doc['type'] == "beer":
        print "Got a beer. It's got %0.2f ABV" % (doc['abv'],)
```

1.3.4. Encoding and Serialization

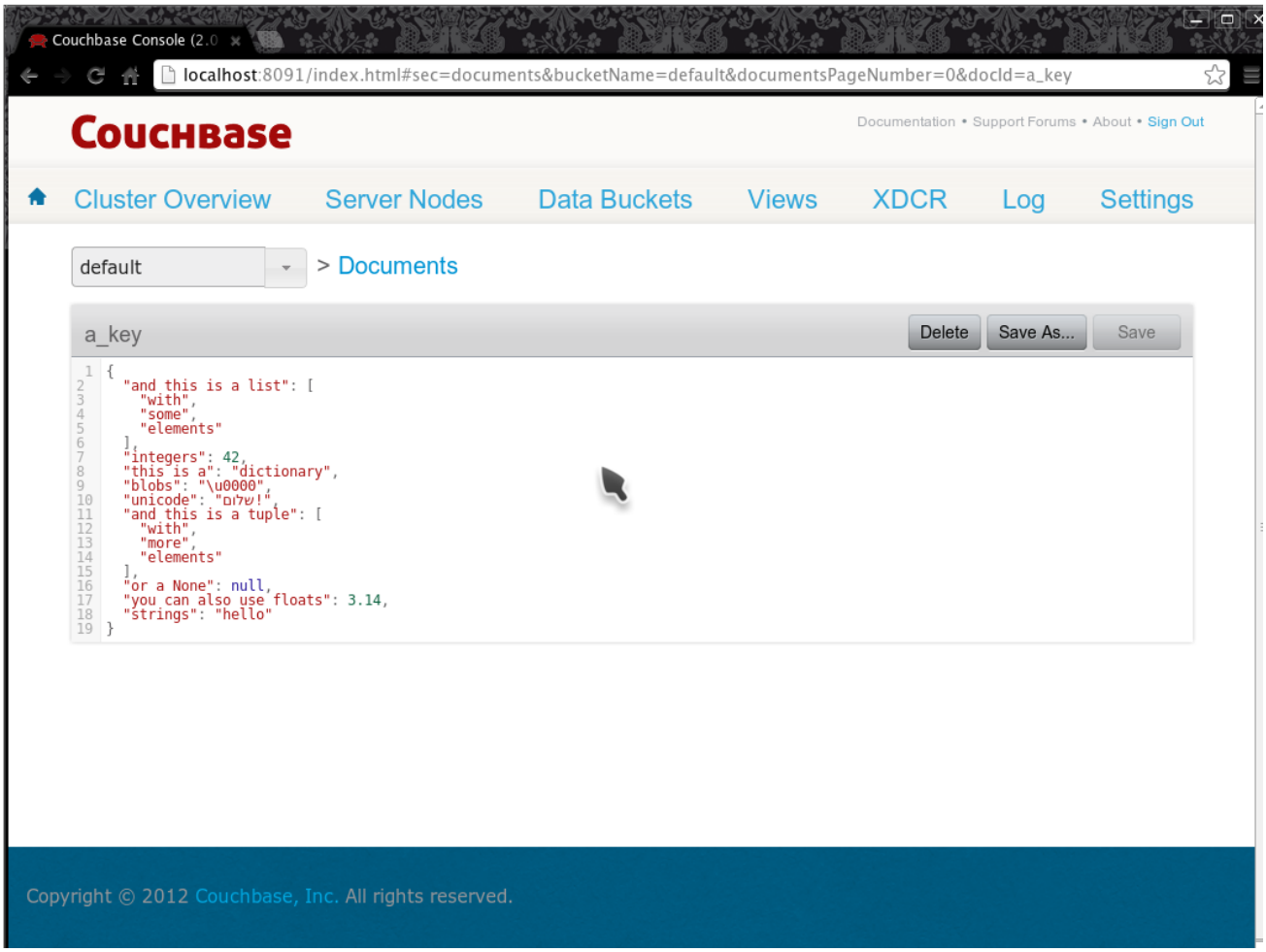
The default encoding format for the Python SDK is JSON. This means you can pass any valid object which is accepted by the standard `json.dumps` library function and you will receive it back when you retrieve it.

```
# -*- coding: utf-8 -*-

import pprint
from couchbase import Couchbase

client = Couchbase.connect(bucket='default', host='localhost')
value = {
    "this is a" : "dictionary",
    "and this is a list" : ["with", "some", "elements"],
    "and this is a tuple" : ("with", "more", "elements"),
    "you can also use floats" : 3.14,
    "integers" : 42,
    "strings" : "hello",
    "unicode" : "####!",
    "blobs" : "\x00",
    "or a None" : None
}

client.set("a_key", value)
result = client.get("a_key")
pprint.pprint(result.value)
print result.value['unicode']
```

Which then prints

```
{u'and this is a list': [u'with', u'some', u'elements'],
 u'and this is a tuple': [u'with', u'more', u'elements'],
 u'blobs': u'\x00',
 u'integers': 42,
 u'or a None': None,
 u'strings': u'hello',
 u'this is a': u'dictionary',
 u'unicode': u'\u05e9\u05dc\u05d5\u05dd!',
 u'you can also use floats': 3.14}
####!
```

If you navigate to the document browser for the bucket in the Web UI (go to `localhost:8091` in your browser, type in your administrative credentials, go over to the *Data Buckets* pane, and click on the *Documents* button for the `default` bucket. Then in the text input box, type in the ID for the document you just created (in this case, it's `a_key`)), you'll see it show up and recognized by the document browser). This means it can now be indexed and queried against using views.



1.3.4.1. Other Formats

While JSON is the default format, it might be useful to utilize other formats. For example, if you wish to store complex custom Python objects and classes and don't require that they be indexed with views, you can use the `pickle` serialization format. This allows you to store types that will not be accepted by JSON:

```
import pprint

from couchbase import Couchbase, FMT_PICKLE

c = Couchbase.connect(bucket='default')
c.set("a_python_object", object(), format=FMT_PICKLE)
c.set("a_python_set", set([1,2,3]), format=FMT_PICKLE)

pprint.pprint(c.get("a_python_object").value)
pprint.pprint(c.get("a_python_set").value)
```

Outputs:

```
<object object at 0x7fa7d0ad80e0>
set([1, 2, 3])
```

You can also store arbitrary strings of bytes by using `FMT_BYTES`

Note

In Python 2 (2.6 and above) `bytes` and `str` are the same type; however in Python 3, a `str` is a string with an encoding (i.e. Python 2's `unicode`) while `bytes` is a sequence of bytes which must be explicitly converted in order to be used with text operations.

```
import pprint

from couchbase import Couchbase, FMT_BYTES

c = Couchbase.connect(bucket='default')
c.set("blob", b"\x01\x02\x03\x04", format=FMT_BYTES)
pprint.pprint(c.get("blob").value)
```

Outputs

```
b'\x01\x02\x03\x04'
```

Or use `FMT_UTF8` to store a `unicode` object represented as *UTF-8*

Note

While JSON is also capable of storing strings and Unicode, the JSON specification mandates that all strings begin and end with a quote (`"`). This uses up needless space and costs extra processing power in "decoding" and "encoding" your JSON string. Therefore you can save on performance by using `FMT_UTF8` for simple strings

Note

It is possible to encode your data in other encodings other than *UTF-8*. However since the view engine operates using *UTF-8*, we select this as the default. If you need a different encoding, consider using the `Transcoder` interface.

```
from couchbase import Couchbase, FMT_UTF8

c = Couchbase.connect(bucket='default')
c.set("EXCALIBUR", u"\u03EE", format=FMT_UTF8)
print c.get("EXCALIBUR")
```

Outputs

```
#
```

Setting The Default Format

You can set the default format for the value type you use most by setting the `default_format` property on the connection object, either during construction or afterwards:

```
c = Couchbase.connect(bucket='default', default_format=FMT_UTF8)
```

Or

```
c.default_format = FMT_PICKLE
```

Chapter 2. Tutorial

In this chapter we will build on the foundations of the *Getting Started* guide, and build a simple web application on top of it. Make sure to have the `beer-sample` bucket installed, as we'll be using it -; the sample application will allow you to edit and manage various beers and breweries.

Note

The sample application is not entirely complete, and there are some features which remain to be implemented. Implementing them is an exercise for the reader.

The full source code is available through couchbaselabs on GitHub

Note that the sample application provides more content then described in this tutorial but it should be simple to navigate while reading this tutorial.

2.1. Quickstart

- Ensure you have *Flask* installed. You can either install it via your distribution or use `pip install flask`.
- Download Couchbase Server and install it. Make sure to install the `beer-sample` dataset when you run the wizard, because this tutorial application will work with it.
- Clone the repository and `cd` into the directory

```
shell> git clone git://github.com/couchbaselabs/beersample-python
Cloning into 'beersample-python'
# ...
shell> cd beersample-python
```

- Some views need to be set up. You can do this manually via the Web UI, or invoke the `design_setup.py` script in the sample repository.

  In the `beer` design document, create a view called `by_name`

```
function (doc, meta) {
    if (doc.type && doc.type == "beer") {
        emit(doc.name, null);
    }
}
```

  Create a design document called `brewery`. Add a view called `by_name`

```
function (doc, meta) {
    if (doc.type == "brewery") {
        emit(doc.name, null);
    }
}
```

- Finally, invoke the `beer.py` script

```
shell> python beer.py
 * Running on http://0.0.0.0:5000/
 * Restarting with reloader
```

- Navigate to `localhost:5000` and enjoy the application!

2.2. Preparations

In this section we'll talk a bit about setting up your directory layout and adding some views in the server before we start dealing with the Python SDK and Flask itself.

2.2.1. Project Setup

Create your project directory, we will call it `beer`

```
shell> mkdir beer
shell> cd beer
shell> mkdir templates
shell> mkdir templates/beer
shell> mkdir templates/brewery
shell> mkdir static
shell> mkdir static/js
shell> mkdir static/css
```

Showing your directory contents should display something like this

```
shell> find . -type d
./static
./static/js
./static/css
./static/html
./templates
./templates/brewery
./templates/beer
```

In order to make the application look pretty, we're incorporating jQuery and Twitter Bootstrap. You can either download the libraries and put it in their appropriate `css` and `js` directories (under `static`), or clone the project repository and use it from there. Either way, make sure to have the following files in place:

- `static/css/beersample.css`
- `static/css/bootstrap.min.css` (the minified twitter bootstrap library)
- `static/css/bootstrap-responsive.min.css` (the minified responsive layout classes from bootstrap)
- `static/js/beersample.js`
- `static/js/jquery.min.js` (the jQuery javascript library)

From here on, you should have a barebones web application configured that has all the dependencies included. We'll now move on and configure the `beer-sample` bucket the way we need it.

2.2.2. Preparing the Views

The beer-sample bucket comes with a small set of views already predefined, but to make our application function correctly we need some more. This is also a very good chance to explore the view management possibilities inside the Web-UI.

Since we want to list beers and breweries by their name, we need to define one view for each. Head over to the Web-UI and click on the *Views* menu. Select `beer-sample` from the dropdown list to switch to the correct bucket. Now click on *Development Views* and then *Create Development View* to define your first view. You need to give it the name of both the design document and the actual view. Insert the following names:

- Design Document Name: `_design/dev_beer`
- View Name: `by_name`

The next step is to define the `map` and (optional) `reduce` functions. In our examples, we won't use the reduce functions at all but you can play around and see what happens. Insert the following map function (that's JavaScript) and click `Save`.

```
function (doc, meta) {
  if(doc.type && doc.type == "beer") {
    emit(doc.name, null);
```

```
  }
}
```

Every map function takes the full document (`doc`) and its associated metadata (`meta`) as the arguments. You are then free to inspect this data and emit a result when you want to have it in your index. In our case we emit the name of the beer (`doc.name`) when the document both has a `type` field and the `type` is `beer`. We don't need to emit a value - that's we we are using `null` here. It's always advisable to keep the index as small as possible. Resist the urge to include the full document through `emit(meta.id, doc)`, because it will increase the size of your view indexes. If you need to access the full document (or large parts), then use the `include_docs` in the `query` method, which will return `ViewRow` objects together with their documents. You can also call `cb.get(row.docid)` as well, to get the individual doc for a single row. The resulting retrieval of the document may be slightly out of sync with your view, but it will be fast and efficient.

Now we need to do (nearly) the same for our breweries. Since you already know how to do this, here is all the information you need to create it:

- Design Document Name: `_design/dev_brewery`
- View Name: `by_name`
- Map Function:

```
function (doc, meta) {
  if(doc.type && doc.type == "brewery") {
    emit(doc.name, null);
  }
}
```

The final step that you need to do is to push the design documents in production. While the design documents are in development, the index is only applied on the local node. Since we want to have the index on the whole dataset, click the *Publish* button on both design documents (and accept any info popup that warns you from overriding the old one).

For more information about using views for indexing and querying from Couchbase Server, here are some useful resources:

- General Information: Couchbase Server Manual: Views and Indexes.
- Sample Patterns: to see examples and patterns you can use for views, see Couchbase Views, Sample Patterns.
- Timestamp Pattern: many developers frequently ask about extracting information based on date or time. To find out more, see Couchbase Views, Sample Patterns.

2.2.3. Structure of the Flask Application

We'll be showing bits and pieces of the web app as it pertains to specific sections. The entire file is less than 300 lines long, and can be inspected by looking into the `beer.py` file in the repository.

First, our imports

We need some extra imports to be able to handle exceptions properly and let us build better view queries.

beer.py (imports).

```
from collections import namedtuple
import json

from flask import Flask, request, redirect, abort, render_template

from couchbase import Couchbase
```

```
from couchbase.exceptions import KeyExistsError
from couchbase.views.iterator import RowProcessor
from couchbase.views.params import UNSPEC, Query
```

Then, we want to set some constants for our application.

beer.py (configuration).

```
DATABASE = 'beer-sample'
HOST = 'localhost'
ENTRIES_PER_PAGE = 30
```

The `ENTRIES_PER_PAGE` variable is used later on to configure how many beers and breweries to show in the search results.

Now, we're ready to create our `Flask` application instance

beer.py (creating the application).

```
app = Flask(__name__, static_url_path='')
app.config.from_object(__name__)
```

The first line creates a new Flask application. The first argument is the module in which the application is defined. Since we're only using a single file as our application, we can use `__name__` which expands to the name of the current file being executed (minus the `.py` suffix).

The second argument instructs Flask to treat unrouted URLs as being requests for files located in the `static` directory we created earlier. This will allow our templates to load the required `.js` and `.css` files.

The second line creates a configuration object for our `app`. The argument is the name of the module to scan for configuration directives. Flask scans this module for variable names in `UPPER_CASE` and places them in the `app.config` dictionary.

Then, define a function to give us a database connection

beer.py (generating a Connection object).

```
def connect_db():
    return Couchbase.connect(
        bucket=app.config['DATABASE'],
        host=app.config['HOST'])

db = connect_db()
```

You already know how to connect to a Couchbase cluster, we'll skip the explanation here.

The second line sets the module-level `db` variable to be the `Connection` object. While in larger applications this is probably not a good idea, since this is a simple app, we can get away with it.

2.3. The Welcome Page

The first route we will implement is that of the `welcome` page, i.e. the page which is displayed when someone will go to the root of your site. Since there is no Couchbase interaction involved, we just tell Flask to render the template.

beer.py (welcome page).

```
@app.route('/')
def welcome():
    return render_template('welcome.html')

app.add_url_rule('/welcome', view_func=welcome)
```

The `welcome.html` is actually a *Jinja* template inside the `templates` directory. Its contents is displayed here:

templates/welcome.html.

```
{% extends "layout.html" %}
{% block body %}
<div class="span6">
  <div class="span12">
    <h4>Browse all Beers</h4>
    <a href="/beers" class="btn btn-warning">Show me all beers</a>
    <hr />
  </div>
  <div class="span12">
    <h4>Browse all Breweries</h4>
    <a href="/breweries" class="btn btn-info">Take me to the breweries</a>
  </div>
</div>
<div class="span6">
<div class="span6">
  <div class="span12">
    <h4>About this App</h4>
    <p>Welcome to Couchbase!</p>
    <p>This application helps you to get started on application
        development with Couchbase. It shows how to create, update and
        delete documents and how to work with JSON documents.</p>
  </div>
</div>

{% endblock %}
```

The template simply provides some links to the brewery and beer pages (which are shown later).

An interesting thing about this template is that it "inherits" from the common `layout.html` template. All pages in the beer app will have a common header and footer to them — with only their `body` differing. Here we will show the `layout.html` template.

templates/layout.html.

```
<!DOCTYPE HTML>

<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Couchbase Python Beer Sample</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="The Couchbase Java Beer-Sample App">
    <meta name="author" content="Couchbase, Inc. 2013">

    <link href="/css/bootstrap.min.css" rel="stylesheet">
    <link href="/css/beersample.css" rel="stylesheet">
    <link href="/css/bootstrap-responsive.min.css" rel="stylesheet">

    <!-- HTML5 shim, for IE6-8 support of HTML5 elements -->
    <!--[if lt IE 9]>
      <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->
  </head>

  <body>
    <div class="container-narrow">
      <div class="masthead">
        <ul class="nav nav-pills pull-right">
          <li><a href="/welcome">Home</a></li>
          <li><a href="/beers">Beers</a></li>
          <li><a href="/breweries">Breweries</a></li>
        </ul>
        <h2 class="muted">Couchbase Beer Sample</h2>
      </div>
      <hr>
      <div class="row-fluid">
        <div class="span12">
            {% block body %}{% endblock %}
        </div>
```

```
      </div>
      <hr>
      <div class="footer">
        <p>&copy; Couchbase, Inc. 2013</p>
      </div>
    </div>
    <script src="/js/jquery.min.js"></script>
    <script src="/js/bootstrap.min.js"></script>
    <script src="/js/beersample.js"></script>
  </body>
</html>
```

If you start your app now, you should be able to navigate to `localhost:5000` and see the welcome page. You'll get a 404 if you try to visit any links though - this is because we haven't implemented them yet. Let's do that now!

2.4. Managing Beers

In this section we'll show the construction of the webapp in respect to managing beers. We'll be able to list, inspect, edit, create, search, and delete beers.

2.4.1. Showing Beers

Now we're finally getting into the cooler stuff of this tutorial.

First, we'll implement several classes for our pages to use.

beer.py (custom Beer row class and processing).

```
class Beer(object):
    def __init__(self, id, name, doc=None):
        self.id = id
        self.name = name
        self.brewery = None
        self.doc = doc

    def __getattr__(self, name):
        if not self.doc:
            return ""
        return self.doc.get(name, "")


class BeerListRowProcessor(object):
    """
    This is the row processor for listing all beers (with their brewery IDs).
    """
    def handle_rows(self, rows, connection, include_docs):
        ret = []
        by_docids = {}

        for r in rows:
            b = Beer(r['id'], r['key'])
            ret.append(b)
            by_docids[b.id] = b

        keys_to_fetch = [ x.id for x in ret ]
        docs = connection.get_multi(keys_to_fetch, quiet=True)

        for beer_id, doc in docs.items():
            if not doc.success:
                ret.remove(beer)
                continue

            beer = by_docids[beer_id]
            beer.brewery_id = doc.value['brewery_id']

        return ret
```

First, we declare a simple `Beer` object. This isn't too fancy and we could've probably just used a simple `dict` - however it allows us to demonstrate the use of the `RowProcessor` interface (defined next).

In the beer listing page, we want to display each beer along with a link to the brewery that produces it. However, we've defined the `beer/by_name` view to only return the name of the beer. In order to obtain the brewery we need to fetch each beer document and examine it. The document will contain the Brewery ID which we can then use later on.

The `BeerListRowProcessor` is an implementation of the `RowProcessor` interface which operates on the returned view rows.

For each raw JSON row, it creates a new `Beer` object; the first argument is the document ID - which is used to provide a link to display more information about the beer. The second is the name of the beer itself which we use in the beer list on the webpage.

We also create a local variable called `by_docids` - this will allow us to get a `Beer` object by its document ID- for reasons we will soon see.

After we've created all the beers, we create a list of document IDs to fetch by using list comprehension. We pass this list to `get_multi` (passing `quiet=True`, as there may be some inconsistencies between view indexes and the actual documents).

While we could have made this simpler by performing an individual `get` on each `beer.id`, this would have been less efficient in terms of network usage.

Now that we have the beer documents, it's time to set each beer's `brewery_id` to its relevant value.

We first check to see that each document was successful in being retrieved; then we look up the corresponding `Beer` object by getting it from the `by_docids` dictionary using the `beer_id` as the key.

Then, we extract the `brewery_id` field from the document and place it into the `Beer` object.

Finally, we return the list of populated beers. The `View` object (returned by the `query` function) will now yield results from it as we iterate over it.

Before we forget, let's put this all together:

beer.py (showing beer listings).

```
@app.route('/beers')
def beers():
    rp = BeerListRowProcessor()
    rows = db.query("beer", "by_name",
                    limit=ENTRIES_PER_PAGE,
                    row_processor=rp)

    return render_template('beer/index.html', results=rows)
```

We tell flask to route requests to `/beers` to this function. We create an instance of the `BeerListRowProcessor` function we just defined above.

We then execute a view query using the `query` method; passing it the name of the design and view (`beer` and `by_name`, respsectively).

We set the `limit` directive to the aforementioned `ENTRIES_PER_PAGE` directive; so as not to flood a single webpage with many results.

We finally tell the `query` method to use our own `BeerListRowProcessor` for processing the results.

We then direct the template engine to render the `beer/index.html` template, setting the template variable `rows` to the iterable returned by the `query` function.

Here is the `beer/index.html` template:

templates/beer/index.html.

```
{% extends "layout.html" %}
{% block body %}

<h3>Browse Beers</h3>
<form class="navbar-search pull-left">
    <input id="beer-search" type="text" class="search-query" placeholder="Search for Beers">
</form>


<table id="beer-table" class="table table-striped">
    <thead>
        <tr>
            <th>Name</th>
            <th>Brewery</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        {% for beer in results %}
        <tr>
            <td><a href="/beers/show/{{beer.id}}">{{beer.name}}</a></td>
            <td><a href="/breweries/show/{{beer.brewery_id}}">To Brewery</a></td>
            <td>
                <a class="btn btn-small btn-warning" href="/beers/edit/{{beer.id}}">Edit</a>
                <a class="btn btn-small btn-danger" href="/beers/delete/{{beer.id}}">Delete</a>
            </td>
        </tr>
        {% endfor %}
    </tbody>
</table>

<div>
    <a class="btn btn-small btn-success" href="/beers/create">Add Beer</a>
</div>

{% endblock %}
```

We're using *Jinja*`{% for %}` blocks to iterate and emit a fragment of HTML for each `Beer` object returned by the query.

If you navigate to `localhost:5000/beers`, you'll see a listing of beers now. Each beer will have an `To Brewery`, `Edit`, and `Delete` button.

On the bottom of the page, you can also see a button `Add Beer` which will allow you to define new beers.

Let's implement the `Delete` button next!

2.4.2. Deleting Beers

Due to the simplicity of Couchbase and Flask, we can implement a single method to delete both beers and breweries.

beer.py (deleting a beer).

```
@app.route('<otype>/delete/<id>')
def delete_object(otype, id):
    try:
        db.delete(id)
        return redirect('/welcome')

    except NotFoundError:
        return "No such {0} '{1}'".format(otype, id), 404
```

Here we tell flask to route any URL which has as its second component the string `delete` to this method. The paths in `<angle brackets>` are routing tokens which flask passes to the handler as arguments. This means that URLs such as `/beers/delete/foobar`, `/foo/delete/whatever` etc. are all routed to here.

When we get an ID, we try to delete it by using the `delete` method. We use a `try` block. If successful, we redirect to the welcome page; but if the key does not exist, we return with an error message and a `404` status code.

You can now access this page by going to `localhost:5000/beers/delete/nonexistent` and get a 404. Or you can delete a beer by clicking on one of the `Delete` buttons in the `/beers` page!

2.4.3. Displaying Beers

Here we will demonstrate how you can display the beers. In this case, we display a page showing all the fields and values of a given beer.

beer.py (showing a single beer).

```
@app.route('/beers/show/<beer_id>')
def show_beer(beer_id):
    doc = db.get(beer_id, quiet=True)
    if not doc.success:
        return "No such beer {0}".format(beer_id), 404


    return render_template(
        'beer/show.html',
        beer=Beer(beer_id, doc.value['name'], doc.value))
```

Like for the `delete` action, we first check to see that the beer exists. We are passed the beer ID as the last part of the URL - this is passed to us as the `beer_id`.

In order to display the information for the given beer ID, we simply call the connection's `get` method with the `beer_id` argument. We also pass the `quiet` parameter so that we don't receive an exception if the beer does not exist.

We then check to see that the `success` property of the returned `Result` object is true. If it isn't we return an HTTP `404` error.

If the beer exists, we construct a new `Beer` object; passing it the ID and the `name` field within the value dictionary.

We then pass this beer to the `templates/beer/show.html` template which we'll show here:

templates/beer/show.html.

```
{% extends "layout.html" %}
{% block body %}

{% set display = beer.doc %}
{% set brewery_id = display['brewery_id'] %}

<h3>Show Details for Beer "{{beer.name}}"</h3>
<table class="table table-striped">
    <tbody>
        <tr>
            <td><strong>brewery_id</strong></td>
            <td><a href="/breweries/show/{{brewery_id}}">{{brewery_id}}</a></td>
        </tr>
        {% for k, v in display.items() if not k == "brewery_id" %}
        <tr>
            <td><strong>{{k}}</strong></td>
            <td>{{v}}</td>
        </tr>
        {% endfor %}
    </tbody>
</table>

<a class="btn btn-medium btn-warning"
    href="/beers/edit/{{beer.id}}">Edit</a>
<a class="btn btn-medium btn-danger"
    href="/beers/delete/{{beer.id}}">Delete</a>

{% endblock %}
```

Here we make the `display` variable in a special `{% set %}` directive. This makes dealing with the rest of the code simpler.

The next thing we do is extract the `brewery_id`, and create a special entry with a link pointing to the page to display the actual brewery.

The next thing we do is iterate over the rest of the fields (omitting the brewery ID); printing out the key and value of each.

Finally, we provide links at the bottom to `Edit` and `Delete` the beer.

2.4.4. Editing Beers

beer.py (beer edit page).

```
def normalize_beer_fields(form):
    doc = {}
    for k, v in form.items():
        name_base, fieldname = k.split('_', 1)
        if name_base != 'beer':
            continue

        doc[fieldname] = v

    if not 'name' in doc or not doc['name']:
        return (None, ("Must have name", 400))

    if not 'brewery_id' in doc or not doc['brewery_id']:
        return (None, ("Must have brewery ID", 400))

    if not db.get(doc['brewery_id'], quiet=True).success:
        return (None,
                ("Brewery ID {0} not found".format(doc['brewery_id']), 400))

    return doc, None

@app.route('/beers/edit/<beer>', methods=['GET'])
def edit_beer_display(beer):
    bdoc = db.get(beer, quiet=True)
    if not bdoc.success:
        return "No Such Beer", 404

    return render_template('beer/edit.html',
                           beer=Beer(beer, bdoc.value['name'], bdoc.value),
                           posturl='/beers/edit/' + beer,
                           is_create=False)


@app.route('/beers/edit/<beer>', methods=['POST'])
def edit_beer_submit(beer):
    doc, err = normalize_beer_fields(request.form)

    if not doc:
        return err

    db.set(beer, doc)
    return redirect('/beers/show/' + beer)
```

We define two handlers for editing. The first is the `GET` method for `/beers/edit/<beer>` which displays a nice HTML form in which we can use to edit it. It passes the template the `Beer` object, a boolean parameter indicating that this is *not* a new beer (as the same template is also used for the `Create Beer` form), and finally the URL to `POST` to when the form is submitted.

The second is the `POST` handler which validates the input. The post handler calls the `normalize_beer_fields` function.

This function converts the form fields into properly formed names for the beer document; then it checks to see that the beer has a valid `name`. It then checks to see that a `brewery_id` was specified and that it indeed exists. Once these checks have passed, it returns a tuple of (`doc`, `None`).

The `POST` handler checks to see that the second element of the tuple is false - if it isn't, then it's an error code, and the first element becomes the error message.

Otherwise, the first element becomes the document.

It then sets the document in Couchbase using the `set` method.

The template is rather wordy as we enumerate all the possible fields with a nice description :)

templates/beer/edit.html.

```
{% extends "layout.html" %}
{% block body %}

{% if is_create %}
<h3>Create Beer</h3>
{% else %}
<h3>Editing {{beer.name}}</h3>
{% endif %}

<form method="post" action="">
  <fieldset>
    <legend>General Info</legend>
    <div class="span12">
      <div class="span6">
        <label>Name</label>
        <input type="text" name="beer_name" placeholder="The name of the beer." value="{{beer.name}}">

        <label>Description</label>
        <input type="text" name="beer_description" placeholder="A short description." value="{{beer.description}}">
      </div>
      <div class="span6">
        <label>Style</label>
        <input type="text" name="beer_style" placeholder="Bitter? Sweet? Hoppy?" value="{{beer.style}}">

        <label>Category</label>
        <input type="text" name="beer_category" placeholder="Ale? Stout? Lager?" value="{{beer.category}}">
      </div>
    </div>
  </fieldset>
  <fieldset>
    <legend>Details</legend>
    <div class="span12">
      <div class="span6">
        <label>Alcohol (ABV)</label>
        <input type="text" name="beer_abv" placeholder="The beer's ABV" value="{{beer.abv}}">

        <label>Biterness (IBU)</label>
        <input type="text" name="beer_ibu" placeholder="The beer's IBU" value="{{beer.ibu}}">
      </div>
      <div class="span6">
        <label>Beer Color (SRM)</label>
        <input type="text" name="beer_srm" placeholder="The beer's SRM" value="{{beer.srm}}">

        <label>Universal Product Code (UPC)</label>
        <input type="text" name="beer_upc" placeholder="The beer's UPC" value="{{beer.upc}}">
      </div>
    </div>
  </fieldset>
  <fieldset>
    <legend>Brewery</legend>
    <div class="span12">
      <div class="span6">
        <label>Brewery</label>
        <input type="text" name="beer_brewery_id" placeholder="The brewery" value="{{beer.brewery_id}}">
      </div>
    </div>
  </fieldset>
  <div class="form-actions">
      <button type="submit" class="btn btn-primary">Save changes</button>
  </div>
</form>

{% endblock %}
```

The template first checks the `is_create` variable - if it's `False`, then we're editing an existing beer, and the caption is filled with that name. Otherwise, it's titled as `Create Beer`.

2.4.5. Creating Beers

This is largely the same as editing beers:

beer.py (create beer page).

```
@app.route('/beers/create')
def create_beer_display():
    return render_template('beer/edit.html', beer=Beer('', ''), is_create=True)


@app.route('/beers/create', methods=['POST'])
def create_beer_submit():
    doc, err = normalize_beer_fields(request.form)

    if not doc:
        return err

    id = '{0}-{1}'.format(doc['brewery_id'],
                          doc['name'].replace(' ', '_').lower())

    try:
        db.add(id, doc)
        return redirect('/beers/show/' + id)

    except KeyExistsError:
        return "Beer already exists!", 400
```

Here we display the same form as the one for editing beers, except we set the `is_create` parameter to True, and pass an empty `Beer` object - needed because the template still tries to populate the form fields with *existing* values.

In the `POST` handler, we call `normalize_beer_field` as above when editing beers.

Since we're creating a *new* beer, we use the `add` method instead. This will raise an exception if the beer already exists. We catch this and display it to the user.

If all things went well, the user is redirected to the beer display page for the newly created beer.

2.4.6. Searching Beers

In the beer listing page above, you may have noticed a search box at the top. We can use it to dynamically filter our table based on user input. We'll use *Javascript* at the client layer to perform the querying and filtering, and views with range queries at the server (flask) layer to return the results.

Before we implement the Python-level search method, we need to put the following in the `static/js/beersample.js` file (if it's not there already) to listen on searchbox changes and update the table with the resulting JSON (which will be returned from the search method):

static/js/beersample.js (snippet).

```
$(document).ready(function() {

    /**
     * AJAX Beer Search Filter
     */
    $("#beer-search").keyup(function() {
       var content = $("#beer-search").val();
       if(content.length >= 0) {
           $.getJSON("/beers/search", {"value": content}, function(data) {
               $("#beer-table tbody tr").remove();
               for(var i=0;i<data.length;i++) {
                   var html = "<tr>";
                   html += "<td><a href=\"/beers/show/"+data[i].id+"\">"+data[i].name+"</a></td>";
                   html += "<td><a href=\"/breweries/show/"+data[i].brewery+"\">To Brewery</a></td>";
                   html += "<td>";
                   html += "<a class=\"btn btn-small btn-warning\" href=\"/beers/edit/"+data[i].id+"\">Edit</a>\n";
```

```
                    html += "<a class=\"btn btn-small btn-danger\" href=\"/beers/delete/"+data[i].id+"\">Delete</a>";
                    html += "</td>";
                    html += "</tr>";
                    $("#beer-table tbody").append(html);
                }
            });
        }
    });
});
```

The code waits for keyup events on the search field, and if they happen, it issues an *AJAX* query on the search function within the app. The search handler computes the result (using views) and returns it as JSON. The JavaScript then clears the table, iterates over the results, and creates new rows.

The search handler looks like this:

beer.py (ajax search response).

```
def return_search_json(ret):
    response = app.make_response(json.dumps(ret))
    response.headers['Content-Type'] = 'application/json'
    return response

@app.route('/beers/search')
def beer_search():
    value = request.args.get('value')
    q = Query()
    q.mapkey_range = [value, value + Query.STRING_RANGE_END]
    q.limit = ENTRIES_PER_PAGE

    ret = []

    rp = BeerListRowProcessor()
    res = db.query("beer", "by_name",
                   row_processor=rp,
                   query=q,
                   include_docs=True)

    for beer in res:
        ret.append({'id' : beer.id,
                    'name' : beer.name,
                    'brewery' : beer.brewery_id})

    return return_search_json(ret)
```

The `beer_search` function first extracts the user input by examining the query string from the request.

It then creates a `Query` object; the `Query` object then has its `mapkey_range` property set to a list of two elements; the first is the user input, and the second is the user input with the magic `STRING_RANGE_END` string appended to it. This form of range indicates that all keys which start with the user input (`value`) will be returned. If we just provided a single element, results would also contain matches which are lexically "greater" than the user input; if we just provided the same value for the second and first elements, only items which matched the string exactly would be returned.

The special `STRING_RANGE_END` is actually a `u"\uEFF"` UTF-8 character, which for the view engine means "end here". You need to get used to it a bit, but it’s actually very neat and efficient.

We re-use our `BeerListRowProcessor` class to filter the results here (as the data required is the same as that of the beer listing (`beer/index.html`) page.

However we need to return a JSON array of

```
{ "id" : "beer_id", "name" : "beer_name", "brewery" : "the_brewery_id" }
```

so we need to convert the rows into JSON first. This is done by the `return_search_json` function.

Now your search box should work nicely.

2.5. Managing Breweries

While this is implemented in the repository above, it is left as an exercise to the reader to work out some more details.

2.6. Wrapping Up

The tutorial presented an easy approach to start a web application with Couchbase Server as the underlying data source. If you want to dig a little bit deeper, the full source code on couchbaselabs on GitHub has more code to learn from. This may be extended and updated from time to time.

Of course, this is only the starting point for Couchbase, but together with the Getting Started Guide, you should now be well equipped to start exploring Couchbase Server on your own. Have fun working with Couchbase!

2.6.1. Food For Thought

There are some things still not implemented in the example; here is some food for thought.

- When deleting a brewery, ensure it has no beers dependent on it.
- Provide a search where one can query beers beloning to a given brewery 
- Handle concurrent updates to a beer and/or brewery 
- Implement a *like* feature, where one can like a beer or a brewery; likewise, they can unlike one as well!

Chapter 3. Using the APIs

Note

This section only gives an introduction to the APIs available. The actual API reference will feature more options for each of the APIs described here and will always be more up-to-date than the documentation here.

This section goes a level lower than the *Getting Started* guide and features the aspects of the APIs offered by the SDK.

3.1. Connecting

While this has been discussed extensively in previous sections, you can connect to a bucket using the simple `Couchbase.connect()` class method.

```
from couchbase import Couchbase

client = Couchbase.connect(bucket='default')
```

3.1.1. Multiple Nodes

Sometimes, it might be beneficial to let the client know beforehand about multiple nodes; for example if you have several nodes in the cluster and some may not be up; if only one node is passed, the client's constructor will raise an exception. You can pass multiple nodes as a list so that the constructor will try each node until it gets a successful connection (or the timeout is reached)

Using Multiple Nodes.

```
c = Couchbase.connect(
    bucket='default',
    host=['foo.com', 'bar.com', 'baz.com']
)
```

3.1.2. Timeouts

The client uses timeouts so that your application will not wait too long if the cluster is overloaded or there are connectivity issues. By default, this timeout value is 2.5 seconds.

You can adjust this value by setting it in the constructor

```
c = Couchbase.connect(bucket='default', timeout=5.5)
```

Or setting the `timeout` property

3.1.3. SASL Buckets

If your bucket is password protected, you can pass the SASL password using the `password` keyword parameter in the constructor

```
c = Couchbase.connect(bucket='default', password='s3cr3t')
```

3.1.4. Threads

This will be discussed later on in more detail, but the `Connection` object is fully thread-safe out of the box by default. You may tune some parameters which sacrifice thread-safety for performance.

3.2. API Return Values

Before we discuss the individual sections of the API, we'll discuss the common return value which is the `Result` object.

Typically subclasses of this object are returned appropriate for the operation executed.

All Result objects have the following properties

- `success`

  A boolean property indicating whether this operation was successful or not.

- `rc`

  This is the low level return code as received from the underlying `libcouchbase` layer. This is 0 on success and nonzero on failure. Typically this will be useful on operations in which `quiet` was set to `True`. Normally you'd use it like this

```
result = client.get("key", quiet=True)
if not result.success:
    print "Got error code", result.rc
```

- `__str__`

  While this isn't really a property, printing out the result object will yield interesting metadata to aid in debugging this particular operation.

3.3. Storing Data

3.3.1. Setting Values

These methods, if successful, set the value of their respective keys. If they fail, they will raise an exception (and are not affected by the `quiet` property).

These methods can accept a `format` property (which indicates the format in which the value will be stored on the server) as well as a `ttl` property which indicates the lifetime of the value; after the `ttl` is reached, the value is deleted from the server.

- `client.set(key, value, **kwargs)`

  Will set the key unconditionally

- `client.add(key, value, **kwargs)`

  Will set the key to the specified value, but only if the key does not already exists (an exception will be raised otherwise).

- `client.replace(key, value, **kwargs)`

  Will replace an existing key with a new value. This will raise an exception if the key does not already exist

3.3.2. Arithmetic and Counter Operations

These methods operate on 64 bit integer counters. They provide efficient mutation and retrieval of values. You may use these in place of the `set` family of methods when working with numeric values; for example:

Using `set`.

```
key = "counter"
```

```
try:
    result = c.get("counter")
    c.set(key, result.value + 1)
except KeyNotFoundError:
    c.add(key, 10)
```

Using `incr`.

```
key = "counter"
c.incr(key, initial=10)
```

These methods accept the `ttl` argument to set the expiration time for their values, as well as an `amount` value which indicates by what amount to modify their values. Additionally, an `initial` keyword argument is available to provide the default value for the counter if it does not yet exists. If an `initial` argument is not provided and the key does not exist, an exception is raised.

The value for the counter stored must either not exist (if `initial` is used) or should be a "Number", that is, a textual representation of an integer.

If using the default `FMT_JSON`, then your integers are already compliant.

If the existing value is not already a number, the server will raise an exception (specifically, a `DeltaBadvalError`).

Arithmetic methods return a `ValueResult` object (subclass of `Result`). The `value` property can be used to obtain the current value of the counter.

- `c.incr(key, amount=1, ttl=0)`

  Increments the value stored under the key.

- `c.decr(key, amount=1, ttl=0)`

  Decrements the value stored under the key. In this case, `amount` is how much to *subtract* from the key

3.3.3. Append and Prepend Operations

These operations act on the stored values and append or prepend additional data to it. They treat existing values as strings and such only work if the existing data stored is a string (i.e. `FMT_UTF8` or `FMT_BYTES`).

The `format` argument is still available, but the value must be either `FMT_UTF8` or `FMT_BYTES`. If not specified, it defaults to `FMT_UTF8`

Otherwise, they are part of the `set` family of methods

```
c.set("greeting", "Hello", format=FMT_UTF8)
c.append("greeting", " World!")
c.get("greeting").value == "Hello World!"
c.prepend("greeting", "Why, ")
c.get("greeting").value == "Why, Hello World!"
```

- `c.append(key, data_to_append, **kwargs)`

  Appends data to an existing value.

- `c.prepend(key, data_to_prepend, **kwargs)`

  Prepends data to an existing value

  Caution

  Ensure that you only append or prepend to values which were initially stored as `FMT_UTF8` or `FMT_BYTES`. It does not make sense to append to a *JSON* or *pickle* string.

Consider:

```
c.set("a_dict", { "key for" : "a dictionary" })
```

The key `a_dict` now looks like this on the server:

```
{"key for":"a dictionary"}
```

Now, prepend to it

```
c.prepend("a dict", "blah blah blah")
```

The value for `a_dict` looks like this now

```
blah blah blah{"key for":"a dictionary"}
```

Now, when you'll try to get it back, you'll see this happen:

```
>>> c.get("a_dict")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "couchbase/connection.py", line 325, in get
    return _Base.get(self, key, ttl, quiet)
  File "/usr/lib/python2.7/json/__init__.py", line 326, in loads
    return _default_decoder.decode(s)
  File "/usr/lib/python2.7/json/decoder.py", line 365, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
  File "/usr/lib/python2.7/json/decoder.py", line 383, in raw_decode
    raise ValueError("No JSON object could be decoded")
couchbase.exceptions.ValueFormatError: <Failed to decode bytes, Results=1, inner_cause=No JSON ob
```

Unfortunately, the SDK has no way to pre-emptively determine whether the existing value is a string or not, and the server does not enforce this.

3.3.4. Expiration Operations

This consists of a single method which is used to update the expiration time of a given key. It is passed two arguments, a key and an expiration time.

If the expiration time is greater than zero, the key receives the new expiration time (which is an offset in seconds, assuming it is smaller than `60*60*24*30` (i.e. a month) - if it is greater, it is considered to be a Unix timestamp).

If the expiration time is zero, then any existing expiration time is cleared and the value remains stored indefinitely (unless explicitly deleted or updated with expiration at a later time).

This is a lightweight means by which to ensure entities "stay alive" without the overhead of having to re-set their value or fetch them.

- `c.touch(key, ttl)`

  Update the given key with the specified `ttl`.

3.4. Deleting Data

- `client.delete(key, quiet=False)`

  Remove a key from the server. If `quiet` is specified, an exception is not raised if the key does not exist.

3.5. Retrieving Data

- `client.get(key, quiet=False, ttl=0)`

Retrieve a key from the server. If the key does not exist, an exception is raised if the key does not exist and `quiet` was set to False.

If `ttl` is specified, this also modifies, in-situ, the expiration time of the key when retrieving it. This is also known as *Get and Touch*

This returns a `ValueResult` object (subclass of `Result`) which may be used to obtain the actual value via the `value` property.

3.6. Locking Data/Ensuring Consistency

In production deployments, it is possible that you will have more than a single instance of your application trying to modify the same key. In this case a race condition happens in which a modification one instance has made is immediately overidden.

Consider this code:

```
def add_friend(user_id, friend):
    result = c.get("user_id-" + user_id)
    result.value['friends'][friend] = { 'added' : time.time() }
    c.set("user_id-" + user_id, result.value)
```

In this case, `friends` is a dictionary of friends the user has added, with the keys being the friend IDs, and the values being the time when they were added.

When the friend has been added to the dictionary, the document is stored again on the server.

Assume that two users add the same friend at the same time, in this case there is a race condition where one version of the friends dict ultimately wins.

Couchbase provides two means by which to solve for this problem. The first is called *Opportunistic Locking*, while the second is called *Pessimistic Locking*.

Both forms of locking involve using a *CAS* value. This value indicates the state of a document at a specific time. Whenever a document is modified, this value changes. The contents of this value are not significant to the application, however it can be used to ensure consistency. You may pass the *CAS* of the value as it is known to the application and have the server make the operation fail if the current (server-side) *CAS* value differs.

3.6.1. Opportunistic Locking

The opportunistic locking functionality can be employed by using the `cas` keyword argument to the `set` family of methods.

Note that the `cas` value itself may be obtained by inspecting the `cas` property of the `Result` object returned by any of the API functions.

In the previous example (i.e. `add_friend`), we can now modify it so that it handles concurrent modifications gracefully:

```
def add_friend(user_id, friend):

    while True:
        result = c.get("user_id-" + user_id)
        result.value['friends'][friend] = { 'added' : time.time() }

        try:
            c.set("user_id-" + user_id, result.value, cas=result.cas)
            break

        except KeyExistsError:
            print "It seems someone tried to modify our user at the same time!"
```

```
            print "Trying again"
```

This is called *opportunistic* locking, because if the *CAS* is not modified during the first loop, the operation succeeds without any additional steps.

3.6.2. Pessimistic Locking

Pessimistic locking is useful for highly contented resources; that is, if the key being accessed has a high likelihood of being contented. While this method may be more complex, it is much more efficient for such resources.

We can use pessimistic locking by employing the `lock` and `unlock` functions.

The `lock` method locks the key on the server for a specified amount of time. Once the key is locked, further attempts to access the key (without passing the proper CAS) will fail with a `TemporaryFailureError` exception until the key is either unlocked, or the lock timeout is reached.

- `c.lock(key, ttl=0)`

  This has the same behavior as `get` (i.e. it returns the value on the server), but the `ttl` argument now indicates how long the lock should be held for.

  By default, the server-side lock timeout is used (which is 15 seconds).

  Returns a `ValueResult`

- `c.unlock(key, cas)`

  Unlocks the key. The key must have been previously locked and must have been locked with the specified `cas`. The `cas` value can be obtained from the `Result` object's `cas` property

  Important

  Calling any of the `set` methods with a valid CAS will implicitly unlock the key, and thus make an explicit call to `unlock` unnecessary — calling `unlock` on a key that is not currently locked will raise an exception.

We can rewrite our `add_friend` example using the lock functions

```
def add_friend(user_id, friend):
    while True:
        try:
            result = c.lock("user_id-" + user_id)
            break

        except TemporaryFailError:
            # Someone else has locked the key..
            pass
    try:
        result.value['friends'][friend] = { 'added' : time.time() }
        c.set("user_id-" + user_id, result.value, cas=result.cas)

    except:
        # We want to unlock if anything happens, rather than waiting
        # for it to time out
        c.unlock(result.key, result.cas)

        # then, raise the exception
        raise
```

When To Use Optimistic Or Pessimistic Locking

Optimistic locking is more convenient and sometimes more familiar to users. Additionally, it does not require an explicit *unlock* phase.

> However, during a CAS mismatch, the full value is still sent to the server in the case of opportunistic locking. For highly contended resources this has impacts on network I/O, as the value must be sent multiple times before it is actually stored.
>
> Pessimistic locking does not retrieve its value unless the operation was successful, however.

3.7. Working With Views

This section will provide a bit more information on how to work with views from the Python SDK. If you are new to views, it is recommended you read the server documentation <link?> which covers the topic itself more extensively.

In order to use views, you must have already set up *design documents* containing one or more view queries you have defined. You can execute these queries from the Python SDK and retrieve their results.

You can define views either via the Couchbase Server web interface, or through the Python SDK (see <link> design document management functions).

Couchbase Server comes with two pre-defined sample buckets which can be installed from the "Sample Buckets" section in the "Settings" pane.

The basic interface for views is such

```
client.query(design_name, view_name)
```

Which returns an iterable object which yields `ViewRow` objects.

`ViewRow` objects are simple namedtuples with the following fields:

- `vr.key`

  The key emitted by the view's `map` function (i.e. first argument to `emit`

- `vr.value`

  The *value* emitted by the view's `map` function (i.e. second argument to `emit`)

- `vr.id`

  The document ID of this row. Can be passed to `get`, `set`, and such.

- `vr.doc`

  A `Result` object containing the actual document, if the `query` method was passed the `include_docs` directive (see later).

The object returned by `query` is a class which defines an `__iter__` (and thus does not have a `__len__` or `items()` method). You can convert it to a list by using *list comprehension*:

```
rows_as_list = [ c.query("beer", "brewery_beers") ]
```

You can also pass options to the `query` method. The list of available options are documented in the `Query` class in the API documentation.

```
from couchbase.views.params import Query

client.query(design_name, view_name,
             limit=3,
             mapkey_range = ["abbaye", "abbaye" + Query.STRING_RANGE_END],
             descending=True)
```

The `include_docs` directive may be used to fetch the documents along with each `ViewRow` object. Note that while it is possible to simply call `c.get(vr.id)`, the client handles the `include_docs` directive by actually performing a batched (`get_multi`) operation.

You can also pass options for the server itself to handle. These options may be passed as either an encoded query string, a list of key-value parameters, or a `Query` object.

Using encoded query strings.

```
client.query("beer", "brewery_beers", query="limit=3&skip=1&stale=false")
```

Note that this is the most efficient way to pass options as they do not need to be re-encoded for each invocation.

However, it is impossible for the SDK to verify the inputs and thus it is suggested you only use a raw string once your query has been refined and optimized.

Using key-value pairs.

```
client.query("beer", "brewery_beers", limit=3, skip=1, stale=False)
```

This allows simple and idiomatic construction of query options.

Using a Query object.

```
from couchbase.views.params import Query

q = Query
q.limit = 3
q.skip = 1
q.stale = False
client.query("beer", "brewery_beers", query=q)
```

The `Query` object makes it simple to programmatically construct a Query, and provides the most maintainable option. When using key-value pairs, the SDK actually converts them to a `Query` object before processing.

`Query` objects also have named properties, making query construction easy to integrate if using an IDE with code completion.

3.7.1. Common View Parameters

Here are some common parameters used for views. They are available either as keyword options to the `query` method, or as properties on the `Query` object

3.7.1.1. Server Parameters

- `mapkey_range = [ "start", "end" ]`

  Set the start and end key range for keys emitted by the `map` function

- `startkey = "start"`

  Set the start key

- `endkey = "end"`

  Set the end key

- `descending = True`

  Invert the default sort order

- `stale = False`

  Possible values are `True`, `False`, or the string `update_after`.

- `limit = 10`

  Limit the number of rows returned by the query

3.7.1.2. `query` Method Options

These are only available as options to the `query` method, and should not be used on the `Query` object.

- `include_docs = True`

  Fetch corresponding documents along with each row

- `streaming = True`

  Fetch results incrementally. Don't buffer all results in memory at once.

3.7.2. Pagination

Often, view results can be large. By default the client reads all the results into memory and then returns an iterator over that result set. You can change this behavior by specifying the `streaming` option to the `query` method. When used, results will be fetched incrementally.

Using `streaming` does not have any impact on how the rows are returned.

3.8. Design Document Management

The Python Couchbase SDK provides means by which you can manage design documents; including all phases of design document development. You can

- Create a development design
- Publish a development design to a production design
- Retrieve a design document
- Delete a design document

Note that all design creation methods take a `syncwait` argument, which is an optional amount of time to wait for the operation to be complete. By default the server (and thus the SDK) only **schedule** a design document operation. This means that if you try to use the view right after you created it, you may get an error as the operation has not yet completed. Using the `syncwait` parameter will poll for this many seconds - and either return successfully or raise an exception.

An additional argument which may be provided is the `use_devmode` parameter. If on, the name of the design will be prepended with `dev_` (if it does not already start with it).

All these operations return an `HttpResult` object which contains the decoded JSON payload in its `value` property.

- `c.design_create(name, design, use_devmode=True, syncwait=0)`

  Creates a new design document. `name` is the name of the design document (e.g. `"beer"`). `design` is either a Python dictionary representing the structure of the design or a valid string (i.e. encoded JSON) to be passed to the server.

- `c.design_get(name, use_devmode=True)`

Retrieves the design document.

- `c.design_publish(name, syncwait=0)`

  Converts a development-mode view into a production mode view. This is equivalent to pressing the **Publish** button on the web UI.

- `c.design_delete(name, use_devmode=True)`

  Delete a design document

Chapter 4. Advanced Usage

This covers advanced topics and builds on the *Using the APIs* section.

4.1. Batched (Bulk) Operations

Most API functions have both single and multi-key (batched) variants. The batched variant will be of the same name as the single-key variant, but have its method name appended with `_multi`.

The batched operations are significantly quicker and more efficient, especially when dealing with many small values, as they allow pipelining of requests and responses, saving on network latency.

Batched operations tend to accept an iterable of keys (or a dict of keys, depending on the method) and return a dictionary of the following format

```
c.foo_multi(["key1", "key2", "key3"])

{
    "key1" : FooResult(...),
    "key2" : FooResult(...),
    "key3" : FooResult(...)
}
```

4.1.1. Exceptions in Batched Operations

Sometimes a single key in a batched operation may fail, resulting in an exception. It is still possible to retrieve the full result set of the failed batched operation by using the `all_results` property of the thrown exception (assuming it is of type `CouchbaseError`)

```
c.set("foo", "foo value")

try:
    c.add_multi({
        "foo" : "foo value",
        "bar" : "bar value",
        "baz" : "baz value"
    })
except CouchbaseError as exc:
    for k, res in exc.all_results.items():
        if res.success:
            # Handle successful operation
        else:
            print "Key {0} failed with error code {1}".format(k, res.rc)
            print "Exception {0} would have been thrown".format(
                CouchbaseError.rc_to_exctype(res.rc))
```

4.2. Using With and Without Threads

The `Connection` object by default is thread safe. In order to do so, it uses internal locks and explicitly locks and unlocks the Python *GIL* to ensure that a fatal error is not thrown by the application.

The locking and unlocking has a slight performance impact, with the guarantee that things will not crash if an application is using threads.

If you are not using threads in your application (at all), you may pass the `unlock_gil=False` option to the `connect` method like so:

```
c = Couchbase.connect(unlock_gil=False, bucket='default')
```

This will disable all locking/unlocking (not to be confused with the `lock` and `unlock` features which operate on keys in the server) functionality. If your application does use threads, those threads will be **blocked** while the `Connection` object waits for the server to respond.

In addition to locking and unlocking the *GIL*, upon entry to each function the `Connection` object locks itself (using the equivalent of `Lock.acquire`) and unlocks itself once it leaves. This is to ensure that multiple threads are not using the same `Connection` object at once; and thus access is serialized.

You may disable this behavior with the following two options:

- Don't lock at all

  The `Connection` object is not locked at all. If your application will try to use the `Connection` object from multiple threads at once, strange errors may happen and your program will eventually core dump.

  If you're sure you're not going to use it from more than one thread, you can use the `lockmode = LOCKMODE_NONE` in the constructor

```
from couchbase import Couchbase, LOCKMODE_NONE
c = Couchbase.connect(bucket='default', lockmode=LOCKMODE_NONE)
```

- Throw an exception if concurrent access is detected

  This is helpful for debugging an application where multiple threads **should** not be accessing the `Connection` object (but for some reason, they are). You can use the `lockmode = LOCKMODE_EXC` for this

```
from couchbase import Couchbase, LOCKMODE_EXC
c = Couchbase.connect(bucket='default', lockmode=LOCKMODE_EXC)
```

The default lockmode is `couchbase.LOCKMODE_WAIT` which waits silently if concurrent access is detected

4.3. Custom Encodings/Conversions

While the Python SDK offers numerous options for converting your data to be suitable for storing on the server, it may sometimes not be enough. For this, the `Transcoder` interface is used.

The `Transcoder` interface allows you to define an object which is called with each value together with the `format` arguments passed to it.

4.3.1. Formats and Flags

The value passed for the `format` parameter is actually a flag which is stored on the server. Each key has a small amount of metadata which is stored with it along on the server. The Python SDK stores the `format` value to the metadata when you store a value (using `set`) and then reads it when retrieving the value (using `get`). If the flag is equal to `FMT_JSON` then it attempts to decode it as JSON; if the flag is equal to `FMT_PICKLE` then it attempts to decode it as Pickle, and so on.

4.3.2. Custom `Transcoder` Objects

You can write a custom transcoder which will allow *Zlib* compression; here's a snippet

```
import zlib

from couchbase.transcoder import Transcoder
from couchbase import FMT_MASK

# We'll define our own flag.
FMT_ZLIB = (FMT_MASK << 1) & ~FMT_MASK

class ZlibTranscoder(Transcoder):
    def encode_value(self, value, format):
        converted, flags = super(ZlibTranscoder, self).encode_value(value, format & FMT_MASK)
        if (format & FMT_ZLIB):
            flags |= FMT_ZLIB
            converted = zlib.compress(converted)
```

```
        return (converted, flags)

    def decode_value(self, value, flags):
        if (format & FMT_ZLIB):
            value = zlib.decompress(value)
            format &= FMT_MASK
        return super(ZlibTranscoder, self).decode_value(value, flags)
```

In the above example, the `ZlibTranscoder` class is defined as a subclass of the provided `couchbase.transcoder.Transcoder` class. The latter is a wrapper class which defaults to use the default conversion methods in the SDK (note that the library does not use any `Transcoder` object by default, but the provided one wraps the built-in converters.

For `encode_value` we are passed the user-specified value (which is any Python object) and a `format` value, which too can be any valid Python object (though the default transcoder only accepts the `FMT_JSON`, `FMT_UTF8`, `FMT_BYTES` and `FMT_PICKLE` values.

We define an additional format flag called `FMT_ZLIB`. We make this one higher than `FMT_MASK` (which is the bitmask for the built-in formatting flags).

In `encode_value` we first call our parent's `encode_value` (only passing the relevant bits of the `format`) and receive the converted value and output flags back (in reality, output flags will typically be the same as the format flags.

Then we convert the already-converted value and compress it as *zlib*. We then AND the flag with our `FMT_ZLIB` bit, and return it. The value and flag returned from the `encode_value` method are stored as is on the server.

We then do the converse when reading data back from the server in `decode_value`. In this method we are passed the value as it is stored on the server, along with the numeric flags as they are stored in the key's metadata. We check to see first whether there is any special `FMT_ZLIB` flag applied, and if so, decompress the data and strip those bits from the flag. Then we dispatch it to the default `decode_value` to handle any further encapulation formats.

This may all be used like so from Python

```
# assuming the ZlibTrancoder class is defined above

c = Couchbase.connect(transcoder=ZlibTranscoder(), bucket='default')
c.set("foo", "long value" * 1000, format=FMT_BYTES|FMT_ZLIB)
c.get("foo")
```

4.3.3. Bypassing Conversion

If you are having difficulties with reading some value from the server (possibly because it was stored using a different client with different flag semantics) then you may disable conversion (when retrieving) entirely by using the `Connection` object's `data_passthrough` property. This is a boolean, and when enabled does not deconvert the value (i.e. it does not call `decode_value` but simply interprets the value as a sequence of bytes and returns them as part of the `Result` object's `value` property

```
c.set("a_dict", {"foo":"bar"})
c.data_passthrough = True
c.get("a_dict").value == b'{"foo":"bar"}'
```

4.4. Logging and Debugging

This section will cover how to uncover bugs in your application (or in the SDK itself).

4.4.1. Components

To debug anything, it is necessary to be able to identify in which domain a problem is found. Specifically there are four components which participate in typical Couchbase operation

- Couchbase Server

  This is the server itself which stores your data. Errors can happen here if your data does not exist, or if there are connectivity issues with one or more nodes in the server. Note that while Couchbase Server is scalable and fault tolerant, there are naturally some conditions which would cause failures (for example, if all nodes are unreachable).

- libcouchbase

  This is the underlying layer which handles network communication and protocol handling between a client and a Couchbase node. Network connectivity issues tend to happen here.

- Python C Extension Layer

  This is the C code which provides the bulk of the SDK. It interfaces with the libcouchbase component, creates `Result` objects, performs input validation and encoding/decoding of keys and values

- Python Layer

  This is written in pure python. For simple key-value operations these normally just dispatch to the C layer. Most of the view option and row code is handled here as well, with the C layer just performing the lower level network handling.

4.4.2. Exception Handling

When something goes wrong, an exception of `CouchbaseError` is typically thrown. The exception object contains a lot of information which can be used to find out what went wrong.

```
from couchbase import Couchbase
from couchbase.exceptions import CouchbaseError

c = Couchbase.connect(bucket='default')
try:
    # Will fail because 'object' is not JSON-serializable
    c.set("key", object())
except CouchbaseError as e:
    print e
```

Printing the exception object will typically produce something like this:

```
# line breaks inserted for clarity

<Couldn't encode value,
    inner_cause=<object object at 0x7f873cf220d0> is not JSON serializable,
    C Source=(src/convert.c,131),
    OBJ=<object object at 0x7f873cf220d0>
>
```

The exception object consists of the following properties:

- `message`

  This is the message (if any) indicating what went wrong. This is always a string

  ```
  >>> e.message
  "Couldn't encode value"
  ```

- `inner_cause`

  If this exception was triggered by another exception, this field contains it. In the above example, we see the exception

  ```
  >>> e.inner_cause
  TypeError('<object object at 0x7f873cf220d0> is not JSON serializable',)
  ```

- `csrc_info`

If present, contains the source code information where the exception was raised. This is only present for exceptions raised from within the C extension

```
>>> e.csrc_info
('src/convert.c', 131)
```

- `objextra`

  Contains the Python object which likely caused the exception. If present, it means the object was of an invalid type or format.

```
>>> e.objextra
<object object at 0x7f873cf220d0>
```

4.4.3. Application Crashes

As this is a C extension, some fatal errors may result in an application crash. On Unix-based systems, these typically look like this:

```
python: src/callbacks.c:132: get_common_objects: Assertion `PyDict_Contains((PyObject*)*mres, hkey) == 0' failed.
Aborted
```

Or simply

```
Segmentation Fault
```

While the actual cause may be in the application code or in the SDK itself, there is often less information available in debugging it.

The SDK should never crash under normal circumstances, and any application crash ultimately indicates a bug in the SDK itself (invalid user input should result in a Python exception being thrown).

To better help us fix the SDK, a C *backtrace* is needed. To generate a helpful backtrace, Python should be available with debugging symbols (this can be done by installing `python-dbg` or `python-debuginfo` from your distribution. Likewise, `libcouchbase` itself should also be installed with debugging symbols (this can be done by installing `libcouchbase2-dbg` or `libcouchbase2-debuginfo` depending on your distribution).

You will also need `gdb` (this is also available on any distribution).

When you have the desired debugging symbols, invoke `gdb` as follows:

We assume `python` is a Python interpreter, and `crash.py` is a script which can trigger the crash.

```
shell> gdb --args python crash.py
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /usr/bin/python...Reading symbols from /usr/lib/debug/usr/bin/python2.7...done.
done.
```

This will bring you to the `gdb` prompt. Run the program by typing `r` and then pressing *enter*.

```
(gdb) r
Starting program: /usr/bin/python crash.py
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
python: src/callbacks.c:132: get_common_objects: Assertion `PyDict_Contains((PyObject*)*mres, hkey) == 0' failed.
```

```
Program received signal SIGABRT, Aborted.
0x00007ffff6fc9475 in *__GI_raise (sig=<optimized out>) at ../nptl/sysdeps/unix/sysv/linux/raise.c:64
64      ../nptl/sysdeps/unix/sysv/linux/raise.c: No such file or directory.
```

Debugging an already-running application

Often in the case of webservers, it is difficult to invoke the script directly. In this case, you need to debug an already-running application. This can be done with `gdb` by determining the process ID of the already-running process. In this case, you can attach `gdb` to the running process like so:

```
shell> gdb -p 29342
.....
(gdb) continue
```

Once `gdb` has attached, you can type `continue` (instead of `r`) to continue the application.

This shows us that an application crashed. When this happens, `gdb` will print the location of the crash. This is not enough, however as we need the full trace of the crash. To do this, type `bt` and then enter to obtain the trace:

```
(gdb) bt
#0  0x00007ffff6fc9475 in *__GI_raise (sig=<optimized out>)
    at ../nptl/sysdeps/unix/sysv/linux/raise.c:64
#1  0x00007ffff6fcc6f0 in *__GI_abort () at abort.c:92
#2  0x00007ffff6fc2621 in *__GI___assert_fail (assertion=assertion@entry=
    0x7ffff67f6f68 "PyDict_Contains((PyObject*)*mres, hkey) == 0",
    file=<optimized out>, file@entry=0x7ffff67f6e0d "src/callbacks.c",
    line=line@entry=132, function=function@entry=
    0x7ffff67f6fe0 "get_common_objects") at assert.c:81
#3  0x00007ffff67f000c in get_common_objects (cookie=<optimized out>,
    key=<optimized out>, nkey=<optimized out>, err=err@entry=LCB_KEY_ENOENT,
    conn=conn@entry=0x7fffffffd328, res=res@entry=0x7fffffffd330,
    restype=restype@entry=2, mres=mres@entry=0x7fffffffd338)
    at src/callbacks.c:132
#4  0x00007ffff67f0623 in get_callback (instance=<optimized out>,
    cookie=<optimized out>, err=LCB_KEY_ENOENT, resp=0x7fffffffd3e0)
    at src/callbacks.c:216
#5  0x00007ffff65cf861 in lcb_server_purge_implicit_responses ()
   from /sources/libcouchbase/inst/lib/libcouchbase.so.2
#6  0x00007ffff65d0f1b in lcb_proto_parse_single ()
   from /sources/libcouchbase/inst/lib/libcouchbase.so.2
#7  0x00007ffff65cfef5 in lcb_server_v0_event_handler ()
   from /sources/libcouchbase/inst/lib/libcouchbase.so.2
#8  0x00007ffff58b9ccc in event_base_loop ()
   from /usr/lib/x86_64-linux-gnu/libevent-2.0.so.5
#9  0x00007ffff65d50f0 in lcb_wait ()
---Type <return> to continue, or q <return> to quit---
```

Python traces can be rather long; continue pressing *enter* until the last line (`--Type <return>...`) is no longer present.

Once you have a backtrace, send the information (along with the script to reproduce, if possible) to your desired support venue.

Note

It is also possible to debug a crash using *Valgrind*, but the process is significantly more involved and requires a slightly modified build of Python. See the *Contibuting* section for more details.

Chapter 5. Contributing

This section contains means by which you can contribute to this SDK.

5.1. General Information

The latest source code for the Python SDK may be found on *github*. It is located at https://github.com/couchbase/couchbase-python-client.

If you wish to contribute to the C extension itself, it may be worthwhile using a debug build of Python.

5.1.1. Compiling Python From Source

You may skip this section if you do not intend to contribute to the C part of the SDK.

Note

The instructions here have been tested on Python 2.6.7 and Python 3.2.4. They will likely work for any version of Python.

In order to generate a debug build of python, you will need to compile it from source. To do this, you will need to modify some Python source files as instructed in the `Misc/README.valgrind` file within the Python source distribution.

Additionally, if you wish to have your Python be useful for installing other packages (for example, `nose`), you will need to have `pip` and `distribute` installed. These themselves depend on several core modules which may not be built by default on some systems.

The `Modules/Setup` file may be modified using the following diff as a guidline:

```
--- ../../tmp/Python-2.6.7/Modules/Setup.dist   2008-11-27 02:15:12.000000000 -0800
+++ Setup.dist  2013-05-15 15:58:30.559170619 -0700
@@ -162,7 +162,7 @@
 # it, depending on your system -- see the GNU readline instructions.
 # It's okay for this to be a shared library, too.

-#readline readline.c -lreadline -ltermcap
+readline readline.c -lreadline -ltermcap


 # Modules that should always be present (non UNIX dependent):
@@ -215,6 +215,7 @@
 #_ssl _ssl.c \
 #      -DUSE_SSL -I$(SSL)/include -I$(SSL)/include/openssl \
 #      -L$(SSL)/lib -lssl -lcrypto
+_ssl _ssl.c -DUSE_SSL -lssl -lcrypto

 # The crypt module is now disabled by default because it breaks builds
 # on many systems (where -lcrypt is needed), e.g. Linux (I believe).
@@ -248,14 +249,14 @@
 # Message-Digest Algorithm, described in RFC 1321.  The necessary files
 # md5.c and md5.h are included here.

-#_md5 md5module.c md5.c
+_md5 md5module.c md5.c


 # The _sha module implements the SHA checksum algorithms.
 # (NIST's Secure Hash Algorithms.)
-#_sha shamodule.c
-#_sha256 sha256module.c
-#_sha512 sha512module.c
+_sha shamodule.c
+_sha256 sha256module.c
+_sha512 sha512module.c
```

```
 # SGI IRIX specific modules -- off by default.
@@ -460,7 +461,7 @@
 # Andrew Kuchling's zlib module.
 # This require zlib 1.1.3 (or later).
 # See http://www.gzip.org/zlib/
-#zlib zlibmodule.c -I$(prefix)/include -L$(exec_prefix)/lib -lz
+zlib zlibmodule.c -I$(prefix)/include -L$(exec_prefix)/lib -lz

 # Interface to the Expat XML parser
 #
```

Note that on some distributions (specifically Debian) you may get a build failure when building the `ssl` module. If so, you likely need to modify the `Modules/_ssl.c` file like so:

```
--- ../../tmp/Python-2.6.7/Modules/_ssl.c       2010-08-03 11:50:32.000000000 -0700
+++ _ssl.c      2013-05-15 15:58:03.471170217 -0700
@@ -302,8 +302,6 @@
         self->ctx = SSL_CTX_new(TLSv1_method()); /* Set up context */
     else if (proto_version == PY_SSL_VERSION_SSL3)
         self->ctx = SSL_CTX_new(SSLv3_method()); /* Set up context */
-    else if (proto_version == PY_SSL_VERSION_SSL2)
-        self->ctx = SSL_CTX_new(SSLv2_method()); /* Set up context */
     else if (proto_version == PY_SSL_VERSION_SSL23)
         self->ctx = SSL_CTX_new(SSLv23_method()); /* Set up context */
     PySSL_END_ALLOW_THREADS
```

Once the source tree is prepared, you can do something like:

```
shell> ./configure --without-pymalloc --prefix=/source/pythons/py267
shell> make install
```

5.1.2. Running Tests

If you've made changes to the library, you need to run the test suite to ensure that nothing broke with your changes.

To run the tests, you need to have the `nose` package installed (this may also work with the `unittest` module as well, but is less tested).

Additionally, you need a real cluster to test against. The test may modify the buckets specified, so be sure not to point it to a production server(!).

Note that the views test may fail if you have made changes to the `beer-sample` bucket.

To tell the test about your cluster setup, copy the file `tests/test.ini.sample` to `tests/test.ini` and modify as needed.

To run the tests, simply do:

```
shell> nosetest -v
```

from within the root of the SDK source.

5.1.3. Building Docs

You will need `sphinx` and `numpydoc` installed. Simply do

```
shell> make -C docs html
```

Once done, the built HTML should be in `docs/build/html`, and you can begin browsing by opening `docs/build/html/index.html` in your browser.

5.2. Source Style Guidelines

For the Python code, a loose adherence to *PEP-8* should be used. For the C extension code, a fairly more strict adherence to *PEP-7* should be used.

Note

These rules are meant to be broken; this just reflects some guidelines to use.

In general:

- Use spaces, not tabs
- Lines should never be longer than 80 columns
- Code should be compatible with Python versions 2.6 up to the latest 3.x

Python-Specific:

- Doc strings should be readable by Sphinx
- Methods should not have more than three positional arguments
- Avoid using string literals in code

  If a new object makes use of a dictionary, consider converting this dictionary to a proper Python object, using a `namedtuple`, etc.

- Avoid dependencies not in Python's standard library

  Though you may add conditional functionality depending on whether a specific library is installed or not.

- Don't use threads

  While threads are a useful construct in application code, they do not belong in library code without good reason.

C-Specific:

- Use of `goto` is better than deeply nested blocks
- Return type and storage specifiers should be on their own line

  Thus:

```
static PyObject*
do_something(PyObject *self, PyObject *args, ...)
{
    /** ... **/
}
```

  Rather than

```
static PyObject *do_something(PyObject *self, PyObject *args)
{
    /** ... **/
}
```

- Code should compile with the following flags (for GCC or clang)

```
-std=c89 -pedantic -Wall -Wextra -Werror \
    -Wno-long-long -Wno-missing-field-initializers
```

- Non-static functions should have a `pycbc_` prefix
- Functions exposed as Python methods should be named as `pycbc_<Object>_<Method>`

  Where `<Object>` is the name of the class in the SDK (e.g. `Connection`) and `<Method>` is the name of the method (e.g. `<get>`), thus, `pycbc_Connection_get`

- Code should be portable to Win32

  Therefore, only include standard library headers and use `PyOS_*` functions when needed.

Appendix A. Release Notes

The following sections provide release notes for individual release versions of Couchbase Client Library Python. To browse or submit new issaues, see Couchbase Client Library Python Issues Tracker.

A.1. Release Notes for Couchbase Client Library Python 0.8.0 Beta (1 September 2012)

Known Issues in 0.8.0

- Exception is thrown on key not found errors with unified client.

```
try:
 bucket.get("key_that_does_not_exist")
except:
 #couchbase.exception.MemcachedError
```

- "id" values from view rows must be converted to strings to be used with Memcached API.

```
view = bucket.view("_design/beer/_view/by_name")
for row in view:
 id = row["id"].__str__()
 beer = bucket.get(id)
 #do something
```

- View queries on authenicated buckets are not currently supported.

A.2. Release Notes for Couchbase Client Library Python 0.7.1 Beta (6 August 2012)

This is the latest release of the Couchbase Python SDK. It is written from the ground up based on the Couchbase C library, libcouchbase.

This release is considered beta software, use it at your own risk; let us know if you run into any problems, so we can fix them.

New Features and Behaviour Changes in 0.7.1

- SDK now installable via python setup.py install from source or via pip install couchbase.

Fixes in 0.7.1

- Temporarily removing unimplemented multi-get until full implementation available. This will be re-addressed in PYCBC-49 in a future release

  Issues: PYCBC-49, PYCBC-49

A.3. Release Notes for Couchbase Client Library Python 0.7.0 Beta (6 August 2012)

This is the latest release of the Couchbase Python SDK. It is written from the ground up based on the Couchbase C library, libcouchbase.

This release is considered beta software, use it at your own risk; let us know if you run into any problems, so we can fix them.

New Features and Behaviour Changes in 0.7.0

- Introduced VBucketAwareClient which extends MemcachedClient with Membase/Couchbase specific features.
- SDK now requires Python 2.6.
- SDK can now handle server restarts/warmups. Can handle functioning Couchbase Server that is loading data from disk after restart.

Fixes in 0.7.0

- Set() now works with integer values; fixes PYCBC-15.

  Issues: PYCBC-15

- Deprecated `get_view` as it was a duplicate of `view_results`.
- Added memcached level `flush()` command to unify client with other SDKs. Please note this only works with 1.8.0 without changing settings. See the release notes for Couchbase 1.8.1 and 2.0.0 for how to enable memcached `flush()`.

  Warning

  This operation is deprecated as of the 1.8.1 Couchbase Server, to prevent accidental, detrimental data loss. Use of this operation should be done only with extreme caution, and most likely only for test databases as it will delete, item by item, every persisted record as well as destroy all cached data.

  Warning

  Third-party client testing tools may perform a `flush_all()` operation as part of their test scripts. Be aware of the scripts run by your testing tools and avoid triggering these test cases/operations unless you are certain they are being performed on your sample/test database.

  Inadvertent use of `flush_all()` on production databases, or other data stores you intend to use will result in permanent loss of data. Moreover the operation as applied to a large data store will take many hours to remove persisted records.

- Renamed VBucketAwareCouchbaseClient to CouchbaseClient.
- Can now create memcached buckets
- SDK can now created memcached buckets.
- Greater than 50% of SDK covered by unittests; fixes PYCBC-46.

  Issues: PYCBC-46

- Better handling of topology changes; fixes PYCBC-4.

  Issues: PYCBC-4, PYCBC-4

- `init_cluster` function has been removed.
- Globally, logging is no longer disabled; fixes PYCBC-31.

  Issues: PYCBC-31

- Set() now returns a proper status in the unified Couchbase() client 0.7.0.
- Added Apache License headers to all files
- Fixed .save() method; fixes MB-5609.

  Issues: MB-5609, MB-5609
- Deprecated Server() in favor of Couchbase() for the unified client name
- SDK now working with mixed clusters, including clusters with memcached type buckets.
- Deprecating getMulti for pep8-compliant multi-get.

Appendix B. Licenses

This documentation and associated software is subject to the following licenses.

B.1. Documentation License

This documentation in any form, software or printed matter, contains proprietary information that is the exclusive property of Couchbase. Your access to and use of this material is subject to the terms and conditions of your Couchbase Software License and Service Agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Couchbase without prior written consent of Couchbase or as specifically provided below. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Couchbase or its subsidiaries or affiliates.

Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Couchbase disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Couchbase. Couchbase and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

This documentation may provide access to or information on content, products, and services from third parties. Couchbase Inc. and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Couchbase Inc. and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.